DSS: Applying Asynchronous Techniques to Architectures Exploiting ILP at Compile Time

Wei Shi, Zhiying Wang, Hongguang Ren, Ting Cao*, Wei Chen, Bo Su, and Hongyi Lu
*School of Computer, *Department of computer science*
*National University of Defense Technology, *Australian National University*
{*shiwei,zywang,ren,chenwei,subo,hylu*}*@nudt.edu.cn *ting.cao@anu.edu.au*

***Abstract*—Embedded application environments require both high performance and low power. Architectures exploiting instruction-level parallelism (ILP) at compile time, such as very long instruction word (VLIW) and transport triggered architecture (TTA), may satisfy the requirements. They can be further enhanced by using asynchronous circuits to significantly reduce power consumption. As such, we are interested in asynchronous processors with architectures exploiting ILP at compile time. However, most of the current asynchronous processors are based on RISC-like architectures. When designing asynchronous VLIW or TTA processors, the distribution of control introduces some serious problems, and errors may occur because of the variable latencies of operations. This paper investigates the asynchronous processor with architecture exploiting ILP at compile time. In order to overcome these problems, we propose a data source selecting (DSS) scheme to guarantee instructions run correctly on asynchronous VLIW and TTA processors. Concretely, an asynchronous pipelined processor based on TTA is designed. The micro-architecture of the proposed asynchronous TTA processor is presented and an asynchronous processor named Tengyue is implemented using 180nm technology. The experimental results, for a range of benchmarks and working modes, show that the implemented asynchronous TTA processor with DSS scheme support runs correctly and power dissipation is reduced to about 43% to 65% of the equivalent synchronous processor.**

I. INTRODUCTION

As the complexity of embedded applications is growing rapidly, systems in the embedded domain are required to be of high performance to meet real-time requirements. Exploiting ILP is an attractive approach to satisfying the high performance requirements, and two approaches are typically used: traditional CPU, such as superscalar processor, can exploit the ILP at run time, and we abbreviate this type of architectures to EIRT (exploiting ILP at run time) architectures; while others, such as VLIW and TTA based processors, exploit the ILP at compile time and they are abbreviated to EICT (exploiting ILP at compile time) architectures in this paper. Furthermore, low power consumption is also required for mobile embedded environments. In order to improve energy-efficiency, a lot of techniques are proposed to overcome the power problem at different levels ranging from circuits to applications [1].

The clock related components have become the largest power consuming part in a processor, and these components include the clock generator, the clock distribution tree, clock drivers, latches and the clock loading due to all the clocked elements [2]. Friedman [3] reported that the clock distribution network consumed more than 25% of the total power in many applications. Approximately 70% of the power is burned by the clock distribution and latches in the POWER 4 processor [4]. In the Alpha 21264, 32% of the chip power is attributed to the global clock network [5]. In embedded SuperH™ processor core, the clock-tree and Flip-Flops take 35% of the total power [6]. In the VLIW processor core of TMS320C6416T, the clock distribution network contribution percentage to the total power consumption is about 76% for some simple algorithms [7]. Instead of being driven periodically by a global clock, asynchronous circuits work on demand and stop when not needed. For this reason, a design utilizing asynchronous circuits produces an effectively power-efficient architecture. Moreover, an asynchronous circuit exhibits average-case performance rather than the worst-case performance of a synchronous circuit.

In the past two decades, many asynchronous processors have emerged. The first asynchronous processor was designed at Caltech in 1988 [8], and it was followed by FAM (fully asynchronous microprocessor) [9], NSR (non-synchronous RISC) [10], STRiP (self-timed RISC processor) [11], Amulet [12], TITAC [13], MiniMIPS [14], ARM996HS [15], and so on. Almost all of the above asynchronous processors are based on RISC architectures. Nowadays, VLIW and TTA are being widely used in the embedded domain for their hardware simplicity [16], [17]. However, the design methods used for asynchronous RISC-like architectures cannot be directly applied to asynchronous EICT architectures.

In EICT architectures, data and control hazards are overcome at compile time. The compiler knows the latency of each operation, and the latency information is used to schedule instructions. Processors run the scheduled program instruction by instruction and need not to do dependence detection at run time. However, EICT architectures cannot tolerate variable latencies of operations. When we implement an EICT architecture based processor using asynchronous circuits, the centralized control becomes distributed because the global clock is replaced by local handshake signals. The naive replacement of the global clock in a synchronous processor with local handshake signals to produce an asynchronous processor may result in incorrect behaviors. By removing the global clock, it is no longer possible to use the cycle count to measure the variable latencies of operations. That is to say, there has always been a conflict between the indefinite

978-1-4244-8935-0/10/$26.00 ©2010 IEEE

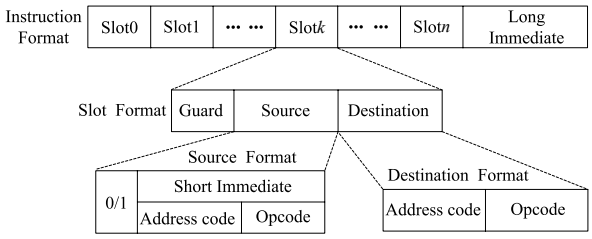

Fig. 1. TTA long instruction format.

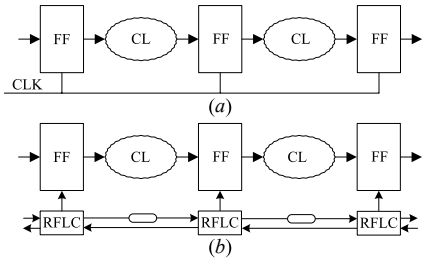

Fig. 2. Synchronous circuit and its asynchronous equivalent.

latencies of asynchronous circuits and the accurate cycle count of operations demanded by the compiler. This paper proposes a DSS scheme to guarantee programs run correctly on asynchronous EICT architectures. In the DSS scheme, the results of operations are buffered and then correct results are picked as inputs of the corresponding operations. In order to elaborate the DSS scheme, the design of an asynchronous pipelined processor based on TTA employing the DSS scheme is presented.

The paper is organized as follows. Section II introduces the TTA and asynchronous circuit design method. Section III presents the asynchronous TTA with DSS scheme. The detail implementation of the asynchronous TTA is explained in Section IV. The delay, area and power consumption of the proposed asynchronous TTA processor are compared to its synchronous counterpart in Section V. Conclusions are given in Section VI.

II. Preliminaries

A. The Synchronous Transport Triggered Architecture

In this paper, we take TTA as an example of EICT architectures to elaborate the DSS scheme. This is because TTA can be regarded as an extreme example of EICT architectures. Actually, some hardware mechanisms such as simplified scoreboard may also be used by VLIW in some situations, but there are not any hardware mechanisms to do dependence detection and hazard elimination in TTA.

TTA proposed by H. Corporaal [18] can be viewed as a superset of the traditional VLIW architecture. However, TTA has unique characteristics compared to conventional processor architectures. One of the main features of a TTA processor is that all the actions are actually side-effects of transporting data from one place to another. When the data is written into a special register, called a trigger register, execution is triggered. Consequently, a TTA processor only has *move* instructions. A long instruction in TTA usually consists of several slots, each of which contains a short *move* transport instruction. These *move* instructions are dispatched to control data transports through the interconnection network and different functional units will be triggered. All the functional units work in parallel which will significantly improve performance. An example of the format of a TTA instruction is shown in Fig. 1, which contains several *move* slots and a long immediate number slot. For each *move* slot, there are 3 sub-domains: guard domain which specifies the condition for data transport; source domain, identifying the source of the data; destination domain, which specifies the target of the transport.

There are usually 5 kinds of components in a TTA based processor. Those components are the instruction fetch unit, instruction decode unit, interconnection network, register files and functional units. All of the register files and functional units are connected by the interconnection network through sockets. Data transports are performed through these sockets, and these sockets are controlled by the decode unit. Each functional unit contains one or more operand registers, a unique trigger register, and one or more result registers. When data is written into the trigger register, the functional unit is triggered to perform computation using the data in both operand registers and trigger register, and then output results are written into the result registers. The pipeline of TTA usually contains 4 stages: instruction fetch (IF), instruction decode (ID), transport (MV) and execution (EXE). The EXE stage can be further divided into several pipeline stages.

B. Asynchronous Design Flow

Synchronous-to-asynchronous circuit conversion methodology [19], which converts synchronous circuits into asynchronous ones using existing EDA tools and flows, is an efficient design approach for asynchronous circuits. Furthermore, the de-synchronization method proposed in [20] is very popular. In this paper, we use a simple design flow which is similar to the de-synchronization methodology to implement a TTA based asynchronous processor. In our design flow, the global clock network is replaced by local handshake circuits, and the Flip-Flops in data paths commonly remain the same as the original synchronous processor. The local controller of a control path uses redundant four-phase latch control (RFLC) protocol which is combined with 2 simple four-phase latch controllers. The synchronous circuit and its asynchronous equivalent are shown in Fig. 2(a) and Fig. 2(b) respectively.

III. Asynchronous Transport Triggered Architecture

A. Problems of the Asynchronous Design for TTA

Data dependence in a program limits the parallelism we can exploit. As such, dependence detection and hazard elimination are considered critical for all the pipelined architectures. Several approaches such as dynamic scheduling, dynamic branch prediction, loop unrolling, and compiler pipeline scheduling have been employed in synchronous pipelines [21].

There are some hardware mechanisms to deal with hazards in RISC-like architectures, and ILP can be exploited at run time. Chang [22] presented methods of dependence detection and hazard elimination for asynchronous RISC processors. Unlike traditional RISC architectures, dependence detection and hazard elimination for EICT architectures are mainly performed by compilers. Based on accurate delay information (in cycles), the compiler knows when instructions in synchronous pipelines are executed and when the results of those instructions can be used. Thus, the hardware just needs to run the program in pre-scheduled manner. To avoid any hazards, the compilation process for TTA addresses the following three issues. Firstly, instructions which definitely will be executed are inserted into the branch delay slots to overcome the control hazard. Secondly, *move* instructions which have the same destination will be scheduled into different long instructions to avoid a structure hazard. Lastly, in order to avoid a data hazard, the compiler arranges instructions according to the execution cycles of each functional unit. In asynchronous TTA, since there is no global clock, it is unreasonable to define the delay of an asynchronous functional unit by cycle count. As a result, the behavior of the instructions running on an asynchronous TTA processor would not be the same as the behavior that the compiler expects, i.e. the uncertainty of the actual execution time of the functional units and the certainty of the originally compiled codes may lead to incorrect results in an asynchronous TTA.

In an asynchronous TTA processor, the global *lock* signal is another problem we must solve. When a data cache miss occurs, the TTA processor core is usually locked by the global *lock* signal to preserve existing states, and since no global clock exists in an asynchronous TTA this mechanism is no longer possible. As an example, Fig. 3(a) shows the behavior of several instructions on a synchronous TTA. Instruction 1 will trigger functional unit n, while instruction 2 will trigger functional unit n and the load/store unit. The result of the instruction 1 on functional unit n and the data loaded by the load/store unit will be used by instruction 5. Functional unit n is implemented in a 3 stage pipeline. In this example, we assume that the data which is wanted by the load/store unit is not in the data cache, and the whole processor core will be locked until the data is retrieved from memory. While in an asynchronous TTA processor, the whole processor core can not be locked for the lack of a global clock reference. Instructions after the instruction 5 can be blocked, but instructions before instruction 5 will continue. When the data is read from memory, the block will be eliminated. However, the result *r1* of instruction 1 on functional unit n has already been replaced by the result *r2* of instruction 2 at the time of instruction 5 entering MV stage, and errors occur. The corresponding behavior of the asynchronous processor core is shown in Fig. 3(b).

B. Asynchronous TTA

According to the analysis in Section III.A, the asynchronous behavior of an EICT architecture may lead to an erroneous result. In order to solve this problem, a novel approach called DSS scheme is proposed. The newly proposed scheme is responsible for selecting correct results for asynchronous EICT architectures to guarantee that instructions run correctly on the asynchronous processor. Fig. 4 shows the improved asynchronous TTA block diagram which adopts the DSS scheme.

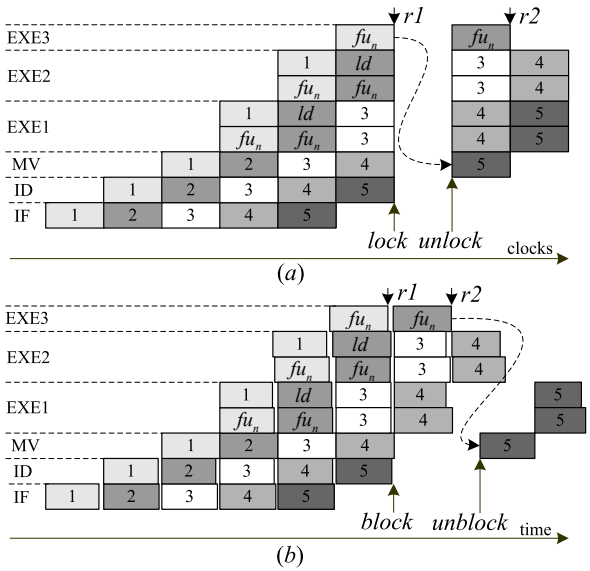

Fig. 3. Behaviors of synchronous and asynchronous TTA.

In IF stage, instructions are fetched from instruction cache. In ID stage, instructions are decoded. The value of condition domain, which controls the execution of the transport, is decided by the result of comparison unit. In MV stage, the results of functional units and data from the register file are delivered to destination registers. In EXE stage, different functional units implemented in asynchronous pipelines execute the operations and calculate the results. However, results of asynchronous functional units may be produced earlier or later than expected. This is because there is not a global reference clock, and it will cause decoding errors in ID stage and also the transporting of incorrect data to destination registers in MV stage. When the data is produced later, the delay of the delay elements in the ID and MV stages can be increased to avoid errors. Actually, in a synchronous TTA, the same method has been used to guarantee the correctness, i.e. extending the global clock cycle to make sure that all the functional units can finish the work in time. When the data is produced earlier, the situation becomes more complicated, and the DSS scheme is used in this circumstance. Firstly, we append an instruction index for each instruction to be executed in asynchronous pipelines. Secondly, a new stage is added at the end of each asynchronous functional unit to latch results. Lastly, DSS circuits are added to the ID and MV stages. A result buffer is used to protect the earlier result from being overwritten by the latter one before it has been accessed. Using instruction indexes, DSS circuits can select the correct results from the result buffers of functional units and send them to destination registers. More details are presented in Section IV.

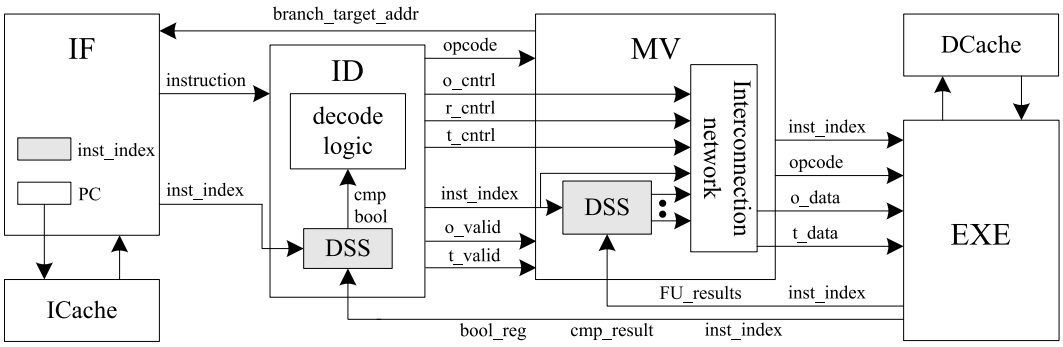

Fig. 4. Asynchronous TTA block diagram.

IV. Logic and Microarchitecture Design

A. Asynchronous TTA Pipeline Structure

The pipeline structure of an asynchronous TTA processor is shown in Fig. 5. The whole pipeline can be divided into several smaller scale pipelines, and each pipeline can be implemented using the asynchronous circuit design method described in Section II.B. One of the small-scale pipelines contains three stages which are IF, ID, MV and the structure is shown in the left part of Fig. 5. Because the branch target address in MV stage is fed back to IF stage, there exists a feedback signal in the control path from MV to IF. Each functional unit in EXE stage can be regarded as an individual pipeline, and it is very easy to design its asynchronous equivalent. In Fig. 5, two blocks are added between the MV and EXE stage. In EXE stage only functional units with valid *trigger* signals are activated. Thus, *trigger* signals are integrated into the control path to avoid every functional unit being activated, and redundant operations are removed. The corresponding circuit structure is shown in shaded block 2. In a TTA processor, there is an instruction which is called dummy instruction, and when the dummy instruction is performed none of the functional units will be triggered. In this situation, an acknowledge signal must be generated to make the processor proceed. The acknowledge signal generating circuit is shown in shaded block 1.

The results of functional units are usually used by the logic in ID and MV stages, and the DSS scheme is added to guarantee the correctness of the asynchronous processor. For this reason, the structure of functional units and combinational logic in ID and MV stages must be modified. The newly modified structure will be explained in Section IV.B and IV.C.

B. Modified Functional units

The asynchronous functional unit pipeline is shown in Fig. 6(a). To implement the DSS scheme, the data path of the functional unit is revised. For each intermediate register, an instruction index *i* and a data usability tag *v* are added. A new stage for buffering results is also added to the end of each functional unit. Instruction index indicates which instruction produced the intermediate result, and data usability tag indicates whether the data is suitable to be transported through the network. The results of each functional unit are firstly latched

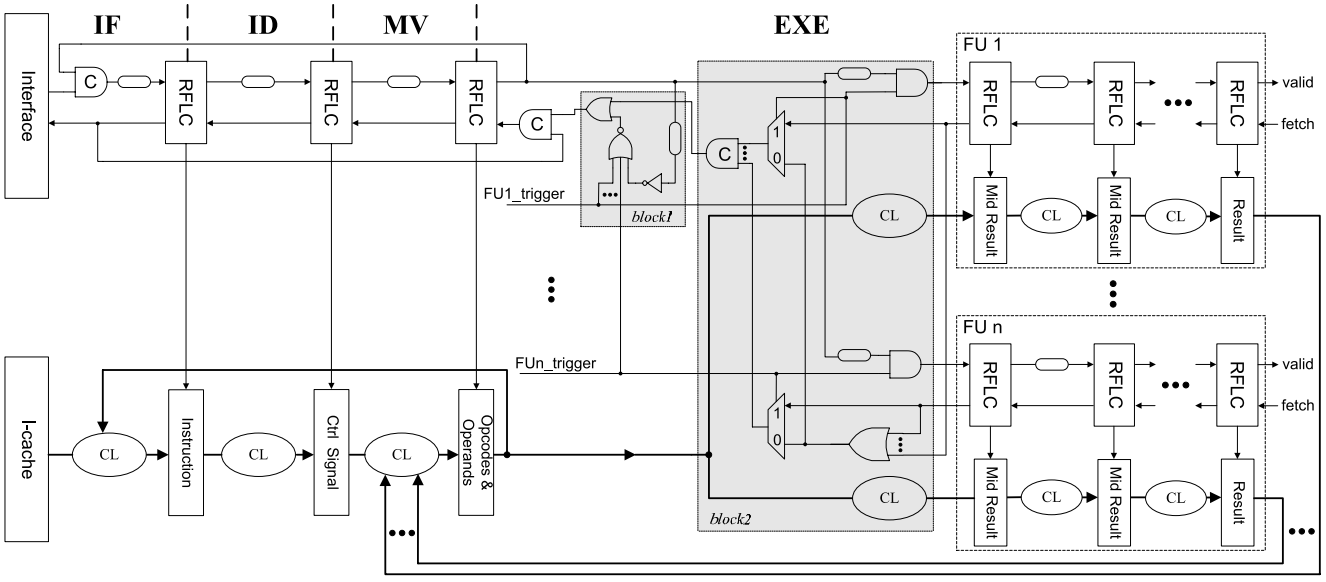

Fig. 5. Pipeline structure of asynchronous TTA.

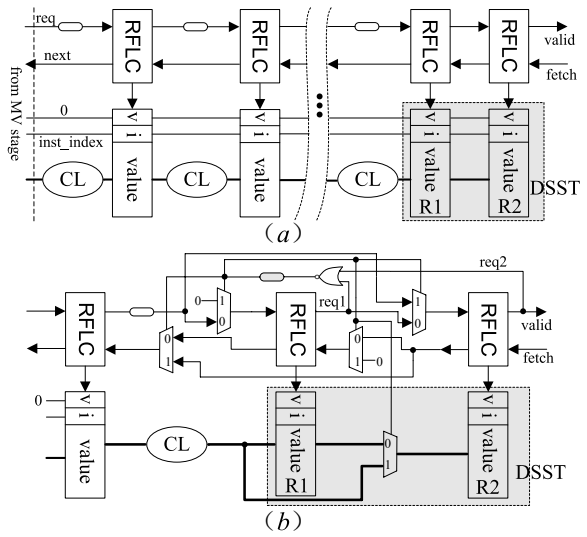

Fig. 6. Pipeline structure of a functional unit and the modified structure.

to register *R1*, and if *R2* is empty then the result in *R1* will be delivered to *R2*. In this case, the process of transferring the result from register *R1* to *R2* surely will increase the execution time. In order to solve this problem, we propose a modified pipeline structure as shown in Fig. 6(b). The output request signals also indicate the validity of the value in the corresponding registers. If the signal *req1* and *req2* in Fig. 6(b) are both low, indicating that values in both registers *R1* and *R2* are invalid, then the result can move directly to *R2* without passing through *R1* temporarily. This will decrease execution time significantly. To improve performance further, the fine-grain asynchronous pipeline such as LDA (locally distributed asynchronous pipelines) [23] can also be used to implement the result buffer. In an asynchronous TTA, a data source selecting table (DSST) containing results of all the functional units and their related information is created. In ID and MV stages of an asynchronous TTA, the correct data are selected from the DSST and then used. Each item of the table is related to a functional unit, comprising execution cycles (*n*) of the corresponding functional unit implemented in synchronous circuits, value of the results (*value*), index (*i*) of the instruction running in the functional unit and validity tag of the results (*req*). When a new instruction enters the pipeline an index number is assigned to the instruction, and the index number is generated by an instruction index counter in IF stage of the processor. We encode the instruction index according to the maximum length of the TTA pipeline, i.e. if the maximum length of the pipeline is *N*, then we might encode the instruction index by a binary number of $\lceil \log_2 N \rceil$ bits.

C. Data Source Selecting Scheme

The DSS scheme is used to ensure that the asynchronous TTA pipeline runs in the correct order and the ID and MV stages always select correct data sources. This approach is based on the hardware structure described in Section IV.B. The structure of DSS scheme is shown in Fig. 7. When a new result is produced, it will be latched in the result register

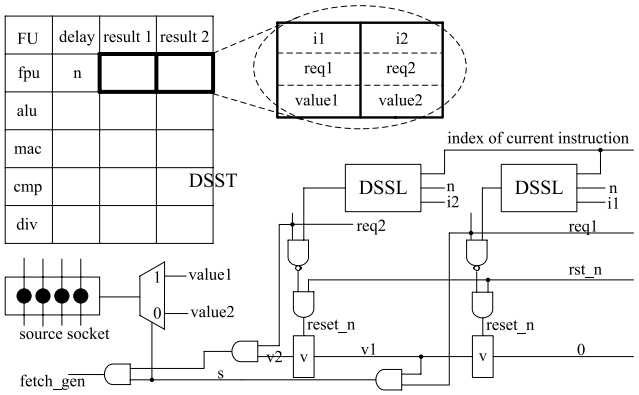

Fig. 7. Data source selecting scheme in MV stage.

buffers as shown in the DSST. The data source selecting logic (DSSL) calculates the usability of the current result, according to the information in the DSST, for the ID or MV stage. The DSSL works as follows: firstly, a difference is calculated by subtracting the index of the current instruction (*I*) with the index stored in the result buffer (*i*); secondly the difference is compared with the delay of the functional unit (*n*); and lastly if the difference value is bigger than the delay, then the value in the result register is usable for the ID or MV stage, otherwise the result is unusable. The difference value can be calculated by the formula (*I*-*i*+*N*) mod *N*.

If the value in the result register is usable and corresponding *req* in the control path is high, then the usability tag *v* will be reset to high. If there is only one usable result, this value will be selected and sent to destination through the network. If the two results are both usable, then the latest result will be selected. In the mean time, the older result is no longer needed, so we can move it out of result buffers. The signal *s* in Fig. 7 indicates which result should be sent through the network. If *s* is high, then register *R1* carries the latest result and *value1* is connected to the network as the selected data source, otherwise *value2* is selected as the data source. The signal *fetch_gen* in Fig. 7 is used to generate a pulse for the fetch port of the asynchronous functional unit. Thus, the older result can be removed from the result buffers and the space is provided for the next result.

V. Experiments and Evaluation

A. Asynchronous Processor Implementation

We selected a set of multimedia kernels from the Berkeley multimedia kernel library (BMKL) [24] and a multimedia application oriented embedded processor named Tengyue was implemented to validate ideas proposed. Tengyue works as a co-processor and the host processor can access Tengyue through the adapter interface. There are two processor cores in Tengyue, a synchronous one called Syn-TTA and an asynchronous one called Asyn-TTA. For the purpose of comparison, these two cores are configured with same functional units. The process of design space exploration and cost estimation of Tengyue has been described in [25]. In each core, there are 4

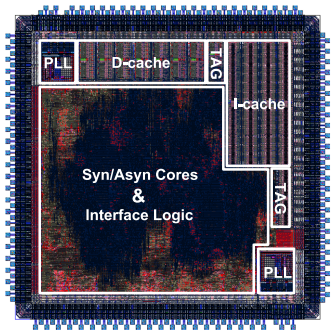

Fig. 8. Layout of Tengyue.

ALUs, 4 multiply-accumulate (MAC) units, 2 floating point units, 1 integer divide functional unit, 1 floating point divide functional unit, 2 load/store units, 2 comparison units, 1 I/O unit and register files. All the functional units and register files in each core are connected by an interconnection network consisting of 8 transport buses.

Tengyue is implemented using the standard cell based semi-custom synchronous design flow and the asynchronous design flow described in Section II.B. The layout of Tengyue is shown in Fig. 8, and the area is 4.89mm×4.89mm.

B. *Performance Analysis*

In the processor core Syn-TTA, the clock cycle is usually set according to the maximum delay of all the stages at worst-case condition. As a result, the synchronous cycle is 4.64 ns. In the Asyn-TTA, the delay of IF, ID and MV stages are set to be the maximum delay of all the stages to guarantee correctness, while the delay of every stage of functional units are set according to actual execution time of the corresponding combinational logic. At worst-case condition, the cycle of asynchronous pipeline is about 5.34ns. The performance overhead is due to the return-to-zero attribute of the four phase handshake protocol.

However, the asynchronous circuit has the ability to operate at their average speed. Andrikos [26] assumed that the performance of asynchronous pipelines obey the normal distribution between the best-case and worst-case condition according to actual running environments of the processor. At the best-case condition, the cycle of Asyn-TTA is about 2.49ns. In a word, the cycle range of Asyn-TTA swings between 2.49ns and 5.34 ns. Fig. 9 shows the probability of Asyn-TTA having better performance than Syn-TTA. In Fig. 9, three different standard deviations are used (σ=0.4, σ=0.5 and σ=0.6). We conclude that at more than 90% of cases, the cycle of the asynchronous pipeline is shorter than the cycle of the synchronous pipeline.

Table 1. Area Comparison of Syn-TTA and Asyn-TTA (μm^2).

	Attributes	Syn-TTA	Asyn-TTA	+%
logic synthesis	cell number	151713	163225	7.59
	comb. logic	2651649	2799977	5.59
	seq. logic	765388	768772	0.44
	cell area	3417037	3568749	4.44
layout	cell area	3661355	3780376	3.25

C. *Area Comparison*

Both the areas for Syn-TTA and Asyn-TTA are shown in Table 1. After logic synthesis, the area of Syn-TTA is 3.42 mm^2, and the area of Asyn-TTA is 3.57mm^2. Asyn-TTA is about 4.44% larger than the Syn-TTA. The extra area cost of Asyn-TTA is mainly due to the control path circuits. Additionally, optimizing asynchronous functional units will increase the data paths and thus increase the area cost. In the process of place and route, some buffer circuits must be inserted to optimize the timing, resulting in an increase in area. In Syn-TTA, an entire clock network must be created, which takes up significant area. For Asyn-TTA, we just need to optimize the local clock network, so the increased area is relatively smaller. Finally, the area of Asyn-TTA is 3.25% larger than Syn-TTA after layout.

D. *Power Consumption*

In synchronous circuits, the clock network requires significant power whether or not any computations are performed. However, control paths of asynchronous circuits consume dynamic power only when they are active. In EICT architectures, functional unit resources are abundant, but they may be heavily underutilized due to the limited parallelism of some programs and the narrow width of the interconnection network. In this situation, there will always be some functional units at idle state, which provides the opportunities for asynchronous circuit to save power. Furthermore, if the whole computational core is idle, then asynchronous circuits will dissipate much

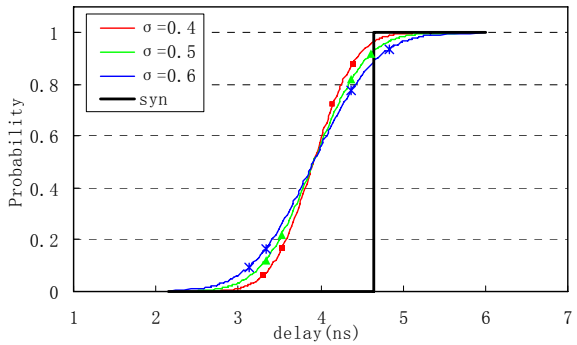

Fig. 9. Performance comparison of Syn-TTA and Asyn-TTA.

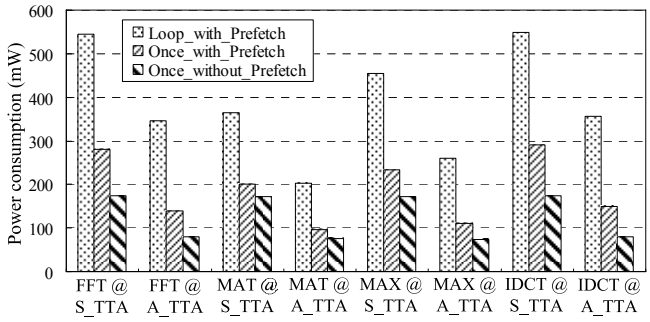

Fig. 10. Power comparison of Syn-TTA and Asyn-TTA.

less power compared to synchronous circuits.

To quantify the power advantage of the Asyn-TTA, we use PrimePower to simulate the power dissipation of the Syn-TTA and Asyn-TTA respectively. The two processor cores are simulated under the same environment, i.e. they are both under the typical case conditions and the cycle of Tengyue is 5ns. Several kernels randomly chosen form BMKL were used to evaluate power dissipation, and simulation results are shown in Fig. 10. There are 3 test modes used for the power simulations: mode I, the program is prefetched into the instruction cache before the processor is started and the program is performed repeatedly; mode II, the program is prefetched and performed only once; and mode III, the program is not prefetched and performed once. In mode I, the power of the asynchronous core takes about 55%~65% of the synchronous core; while in mode II and mode III, the percentage is reduced to about 47%~52% and 43%~47% respectively. This is because the cache miss in the later two modes may make the TTA core idle, and asynchronous circuits exhibit their low power advantage.

VI. Conclusions

High performance and low power are both important requirements in embedded application environments. The design of asynchronous processors with EICT architectures is an efficient way to satisfy the above requirements. However, the previously introduced asynchronous processors are almost all based on RISC-like architectures. This paper investigates asynchronous processor structures with EICT architectures, and an asynchronous processor based on transport triggered architecture is presented as an example. Using the DSS scheme, asynchronous techniques can be applied to the EICT architectures. The experiments show that the designed asynchronous processor can operate correctly and the proposed scheme works effectively. The asynchronous TTA processor achieves better performance and lower power consumption with a little more area overhead. In particular, the more time the asynchronous processor is idle, the greater the power saving compared to the synchronous equivalent.

Acknowledgment

The authors gratefully acknowledge the contribution of Dr. John Zigman at Australian National University and the reviewers for their valuable comments. This work is supported by the National Basic Research Program of China under grand 2007CB310901 and the National Natural Science Foundation of China under grand 60773024, 60903039.

References

[1] V. Venkatachalam, and M. Franz, "Power reduction techniques for microprocessor systems," *ACM Computing Surveys*, vol. 37, no. 3, pp. 195–237, Mar. 2005.

[2] V. Tiwari, D. Singh, S. Rajgopal, G. Mehta, R. Patel, and F. Baez, "Reducing power in high-performance microprocessros," in *Proc. of the ACM/IEEE conf. on Design Automation Conference*, 1998, pp. 732–737.

[3] E. G. Friedman, "Clock distribution networks in synchronous digital integrated circuits," *Proceedings of the IEEE*, vol. 89, no. 5, pp. 665–692, May 2001.

[4] C. J. Anderson, J. Petrovick, J. M. Keaty, J. Warnock, G. Nussbaum, J. M. Tendier, *et al*, "Physical design of a fourth-generation POWER GHz microprocessor," in *IEEE Int. Solid-State Circuits Conf.*, 2001, pp. 232–233.

[5] M. K. Gowan, L. L. Biro, and D. B. Jackson, "Power considerations in the design of the Alpha 21264 microprocessor," in *Proc. of the ACM/IEEE Design Automation Conf.*, 1998, pp. 726–731.

[6] T. Yamada, M. Abe, Y. Nitta, and K. Ogura, "Low-power design of 90-nm SuperH^{TM} processor core," in *Proc. of the IEEE Int. Conf. on Computer Design*, 2005, pp. 258–263.

[7] M. E. A. Ibrahim, M. Rupp, and H. A. H. Fahmy, "Power estimation methodology for VLIW digital signal processors," in *Proc. of the Asilomar Conf. on Signals, Systems, and Computers*, 2008, pp. 26–29.

[8] A. J. Martin, S. M. Burns, T. K. Lee, D. Borkovic, and P.J. Hzaewindus, "The design of an asynchronous microprocessor," in *Proc. of the Decennial Caltech Conf. on VLSI*, 1989, pp. 351–373.

[9] K. R. Cho, K, Okura, and K. Asada, "Design of a 32-bit fully asynchronous microprocessor (FAM)," in *Proc. of the Midwest Symp. on Circuits and Systems*, 1992, pp. 1500–1503.

[10] E. Brunvand, "The NSR processor," in *Proc. of the Annu. Hawaii Int. Conf. on System Sciences*, 1993, pp. 428–435.

[11] M. E. Dean, "STRiP: A Self-Timed RISC Processor," PhD thesis, Dept. of Computer Science, Stanford Univ., CA, USA, 1992.

[12] J. V. Woods, P. Day, S. B. Furber, J. D. Garside, N. C. Paver and S. Temple, "AMULET1: an asynchronous ARM microprocessor," *IEEE Trans. on Computers*, vol. 46, no.4, pp. 385–398, Apr. 1997.

[13] T. Nanya, Y. Ueno, H. Kagotani, M.Kuwako, and A.Takamura, "TITAC: design of a quasi-delay-insensitive microprocessor," *IEEE Design and Test of Computers*, vol. 11, no. 3, Mar. 1994, pp. 50–63.

[14] A. J. Martin, A. Lines, R. Manohar, M. Nystrom, P. Penzes, *et al*, "The design of an asynchronous MIPS R3000 micro-processor," in *Proc. Conf. Advanced Research in VLSI*, 1997, pp. 164–181.

[15] A. Bink and R. York, "ARM996HS: the first licensable, clockless 32-bit processor core," *IEEE Micro*, vol. 27, no. 2, Mar.-Apr. 2007, pp. 58–68.

[16] A. C. S. Beck and L. Carro, "A VLIW low power java processor for embedded applications," in *Proc. of the Symp. on Integrated Circuits and System Design*, 2004, pp. 157–162.

[17] P. Hamalainen, J. Heikkinen, M. Hannikainen, and T. D. Hamalainen, "Design of transport triggered architecture processors for wireless encryption," in *Proc. of the Euromicro conf. on Digital System Design*, 2005, pp. 144–152.

[18] H. Corporaal, *Microprocessor architecture: from VLIW to TTA*. West Sussex, England: John Wiley & Sons Ltd, 1998.

[19] M. Simlastik, V. Stopjakova, "Automated synchronous-to-asynchronous circuits conversion: a survey," in *Int. Workshop on Power and Timing Modeling, Optimization and Simulation*, 2008, pp. 348–358.

[20] J. Cortadella, A. Kondratyev, S. Member, L. Lavagno, and C. P. Sotriou, "Desynchronization: synthesis of asynchronous circuits from synchronous specifications," IEEE Trans. on Computer-Aided Design, vol. 25, no. 10, Oct. 2006, pp. 1904–1921.

[21] J. L. Hennessy, and D. A. Patterson, *Computer architecture: a quantitative approach, third edition*. San Francisco, CA: Morgan Kaufmann Publishers, 2003.

[22] M. C. Chang, and D. S. Shiau, "Design of an asynchronous pipelined processor," in *Int. Conf. on Communications, Circuits and System*, 2008, pp. 1226–1229.

[23] C. Choy, J. Butas, J. Povazanec, and C. Chan, "A fine-grain asynchronous pipeline reaching the synchronous speed," in *Proc. of the conf. on ASIC*, 2001, pp. 547–550.

[24] N. Slingerland, and A. J. Smith, "Measuring the performance of multimedia instruction sets," *IEEE Trans. on Computers*, vol. 51, no. 11, pp. 1317–1332, Nov. 2002.

[25] M. Lai, J. Guo, Z. Zhang, and Z. Wang, "Using an automated approach to explore and design a high-efficiency processor element for the multimedia domain," in *Proc. of Int. Conf. on Complex, Intelligent and Software Intensive Systems*, 2008, pp.613–618.

[26] N. Andrikos, L. Lavagno, D. Pandini, C. P. Sotiriou, "A fully-automated desynchronization flow for synchronous circuits," in *Proc. of the ACM/IEEE Design Automation Conf.*, 2007, pp. 982–985.