ASYNCHRONOUS MICROPROCESSORS

Jurij Šilc
Computer Systems Department, Jožef Stefan Institute, Ljubljana, Slovenia
email: jurij.silc@ijs.si
AND
Borut Robič
Faculty of Computer and Information Science, University of Ljubljana, Slovenia
email: borut.robic@fri.uni-lj.si

Keywords: asynchronous logic, asynchronous processor, self-timed processor

Edited by: Rudi Murn
Received: April 6, 1999 **Revised:** May 19, 1999 **Accepted:** May 25, 1999

The asynchronous processors attack clock-related timing problems by asynchronous (or self-timed) design techniques. Asynchronous processors remove the internal clock. Instead of a single central clock that keeps the chip's functional units in step, all parts of an asynchronous processor (e.g., the arithmetic units, the branch units, etc.) work at their own pace, negotiating with each other whenever data needs to be passed between them. In this paper, several projects are presented, two of these – the Superscalar Asynchronous Low-Power Processor (SCALP) and AMULET – are presented in more detail.

1 Introduction

Conventional synchronous architectures are based on global clocking whereby global synchronization signals control the rate at which different elements operate. For example, all functional units operate in lockstep under the control of a central clock [16].

With progress of time and improvement of technology, clocks get faster, the chips have higher circuit density and the wires get finer. As a result, it becomes increasingly difficult to ensure that all parts of the processor are ticking along in step with each other. Even though the electrical clock pulses are travelling at a substantial fraction of the speed of light, the delays in getting from one side of a small piece of silicon to the other can be enough to throw the chip's operation out of synchronization. Even if the clock were injected optically to avoid the wire delays, the signals issued as a result of the clock would still have to propagate along wires in time for the next clock pulse, and a similar problem would remain. For example, the 1997 National Technology Roadmap for Semiconductors [23] forecasts that CMOS technology will reach a point where the switching delay for a single gate will be close to 10 ps while a single chip area will be nearly 7.5 cm^2. It will take 30 clock cycles for the electric signal to cross such a chip. Moreover, the interchip clock skew already represents a major problem.

The clock-related timing problems have been recently attacked by *asynchronous* (or *self-timed*) design techniques. These asynchronous processors do away with the idea of having a single central clock keeping the chip's functional units in step. Instead, each module of the processor – for example, the arithmetic units, the branch units, etc. – all work at their own pace, negotiating with each other whenever data needs to be passed between them. The communication protocol synchronizes the modules involved in the communication and allows data to be shared between them.

Without a global clock, asynchronous systems enjoy [19]:

- Data-dependent cycle time rather than worst case cycle time: The conventionally clocked chip has to be slowed down so that the most sluggish function does not get left behind. To deal with this problem one can either use some extra circuitry to try to speed up these slow special cases, or alternatively just accept it and slow everything down to take account of the lowest common denominator. Either way the result is that resources are wasted or the chip's speed is determined by an instruction that may hardly ever be executed. In the asynchronous approach the chip only becomes more sluggish when a tricky operation is encountered.
- Potential for low power consumption: The conventional processors are becoming increasingly power consuming. For example, DEC's Alpha and the IBM/Motorola PowerPC 620 emit around 20 W to 30 W in normal operation. If we were to continue to use 5 V supplies, we could expect by the end of 1999 a 0.1 micron processor dissipating 2 kW. Reducing the supply to 3 V (or 2 V) would only reduce the power dissipation to 660 W

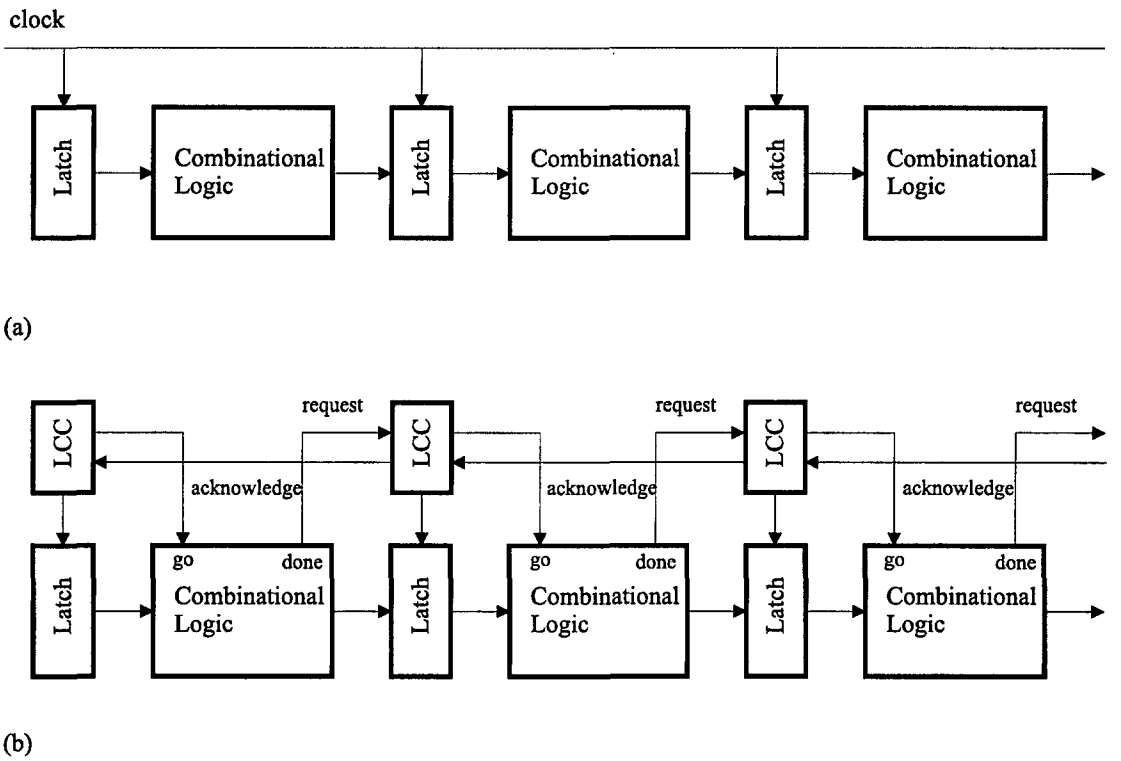

Figure 1: A simple pipeline (a) synchronous (b) asynchronous

(or 330 W). One of the reasons is that many of the logic gates switch their states simply because they are being driven by the clock, and not because they are doing any useful work. Removing the clock in asynchronous processors also removes the unnecessary power consumption as CMOS gates only dissipate energy when they are switching.

- Ease of modular composition, i.e., circuits can be assembled as plug-and-play.
- Optimization of frequent operations while rare operations can spend more time.
- No need for clock alignment at the interfaces.
- Timing fault-tolerance.

There are also several shortcomings to the asynchronous approach:

- clock-based computers are easier to build than asynchronous;
- it is easier to verify a synchronous design due to its deterministic operation (by comparison, verifying an asynchronous design, with each part working at its own pace, is difficult).

2 Asynchronous Logic

Virtually all digital design today is based on a synchronous approach whereby each subsystem is a clocked finite state machine that changes its states on the edges of a regular global clock. Such a system behaves in a discrete and deterministic way, provided the delays are managed so that the flip-flop setup and hold times are met under all conditions.

As a contrast, in asynchronous design, there is no clock to govern the timing of state changes. Subsystems exchange information at mutually negotiated times with no external timing regulation. An asynchronous pipeline, such as *micropipelines* [18], manages the flow of data according to the state of the next and previous pipeline stages. In a synchronous pipeline, if a stage is late in completing operation of the combinatorial circuit, the entire pipeline delayed by an amount of time equal to the clock period. For an asynchronous pipeline, however, if a stage is late by an duration ω, the entire pipeline is delayed just by ω. Also in contrast to synchronous pipelines, an asynchronous pipeline controller only judges the state of adjoining stages. Therefore decentralized control is possible [19].

Figure 1a shows the structure of a synchronous pipeline with latches and combinational logic blocks. All latches are controlled by a single global clock signal and operate simultaneously.

An asynchronous implementation of the pipeline is shown in Fig. 1b. The latches and the combinational logic block are the same as in the synchronous pipeline. The timing, however, is controlled differently. Each latch has an associated *latch control circuit* (LCC)

which opens and closes the latch in response to *request* signals from the previous stage and *acknowledge* signals from the following stage. There are a few key features which describe most current approaches:

- *Delay-insensitive* vs *speed-independent* design: Delay-insensitive designs make no assumptions about delays within the system. That is, any gate or interconnection may take an arbitrary time to propagate a signal. Speed-independent systems are tolerable to variations in gate speeds but assume instantaneous transmissions along wires.
- *Dual-rail encoding* vs *data bundling* communication protocol: In dual rail encoded data, each Boolean is implemented as two wires. This allows the value and the timing information to be communicated for each data bit. Bundled data, on the other hand, has one wire for each data bit and a separate wire to indicate the timing.
- *Level* vs *transition* encoding: Level-sensitive circuits typically represent a logic one by a high voltage and a logic zero by a low voltage. In transition signaling, only changes in the level of signals are taken into account.

Delay-insensitive circuits with dual-rail communication and encoding with transition signaling proved to be ideal for automatic transformation into a silicon layout, as the delays introduced by the layout compiler cannot affect the functionality. The most popular form in recent years has been dual-rail encoding with level-sensitive signaling. Delay insensitivity is achieved at the cost of more power dissipation than with transition signaling. The advantage of this approach over transition signaling is that the logic processing elements can be much simpler. A well-known form of delay-insensitive circuit with bundled data communication and encoding with transition signaling is the *micropipelined* approach, which was proposed in [18] and adopted in the AMULET project (see below).

3 Microprocessors

A number of asynchronous microprocessors have been proposed or built recently. The processors described can be divided broadly into two categories:

- Those that were built using a conservative timing model, suitable for formal synthesis or verification, but with a simple architecture. Among these are CAP, TITAC, ST-RISC, ARISC, and ASPRO-216.
- Those that were built with a less cautious timing model using an informal design approach, but with a more ambitious architecture. These include the AMULET processors, NSR, Fred, CPP, Hades, ECSTAC, STRiP, and SCALP.

Table 3 summarizes these characteristics.

Let us describe the architecture and the asynchronous design of asynchronous superscalar processors SCALP and AMULET and, only briefly, some other projects [7, 22].

3.1 Superscalar Asynchronous Low-Power Processor

The first asynchronous superscalar processor was designed in 1996 at the University of Manchester [7]. The processor was named SCALP, for Superscalar Asynchronous Low-Power Processor. SCALP's main architectural innovation is its lack of a global register file and its result forwarding network. Most SCALP instructions do not specify the source of their operands and destination of their results by means of register numbers. Instead, the idea of *explicit forwarding* was introduced whereby each instruction specifies the destination of its result. That destination is the input to another functional unit consuming the value. Instructions do not specify the source of their operands at all; they implicitly use the values provided for them by the preceding instructions.

Figure 2 shows the organization of the SCALP processor. SCALP does have a register file; it constitutes one of the functional units. It is accessed only by read and write instructions which transfer data to and from other functional units by means of the explicit forwarding mechanism. Several instructions are fetched from memory at a time. Each instruction has a *functional unit identifier*, which is a small number of easily decoded bits that indicate which functional unit will execute the instruction. The instructions are statically allocated to functional units. If there is more than one functional unit capable of executing a particular instruction, one must be chosen by the compiler. This simplifies the instruction issuer and is essential to the explicit forwarding mechanism. The instruction issuer is responsible for distributing the instructions to the various functional units on the basis of the functional unit identifier. Each functional unit has a number of input queues: one for instructions and one for each of its possible operands. An instruction begins execution once it and all of its necessary operands have arrived at the functional unit. The functional unit sends the result, along with the destination address, to the result-routing network. This places the result into the appropriate input queue of another functional unit.

There are some similarities between the SCALP approach and dataflow computing. In particular, it is possible to describe SCALP programs by means of dataflow graphs. Nevertheless, the flow of control in SCALP is determined by a conventional control-flow mechanism, not a dataflow mechanism.

Processor	Design Style	ISA	Organization
CAP	4-phase, dual-rail delay-insensitive	own 16-bit RISC-like	fetch-execute pipeline
FAM	4-phase, dual-rail delay-insensitive	own RISC-like	pipelined
STRiP	variable clock synchronous	MIPS-X	pipelined forwarding
ST-RISC	dual-rail delay-insensitive	own	fetch-execute pipeline
NSR	2-phase bundled data	own 16-bit RISC-like	pipelined no forwarding decoupled branch & load/store
CFPP	2-phase bundled data	SPARC	pipelined multiple execution stages single issue result pipeline forwarding using counter-flow
AMULET1	2-phase bundled data	ARM	pipelined no forwarding
TITAC-1	2-phase, dual-rail quasi delay-insensitive	own 8-bit	nonpipelined
Fred	2-phase bundled data	based on MC 88100	pipelined multiple functional units single issue no forwarding decoupled branch & load/store
Hades	unspecified	own	pipelined multiple functional units multiple issue forwarding
ECSTAC	fundamental mode	own variable length	pipelined no forwarding
AMULET2e	4-phase bundled data	ARM	pipelined forwarding
SCALP	4-phase bundled data	own	pipelined multiple functional units multiple issue explicit forwarding
TITAC-2	2-phase, dual-rail scalable delay-insensitive	own 32-bit	pipelined multiple functional units
AMULET3i	4-phase bundled data	ARM	pipelined branch prediction out-of-order completion unrestricted forwarding
ARISC	4-phase bundled data	MIPS-II, MIPS16	pipelined multiple functional units
ASPRO-216	4-phase, multi-rail quasi delay-insensitive	own 16-bit	pipelined out-of-order completion

Table 1: Recent asynchronous microprocessors

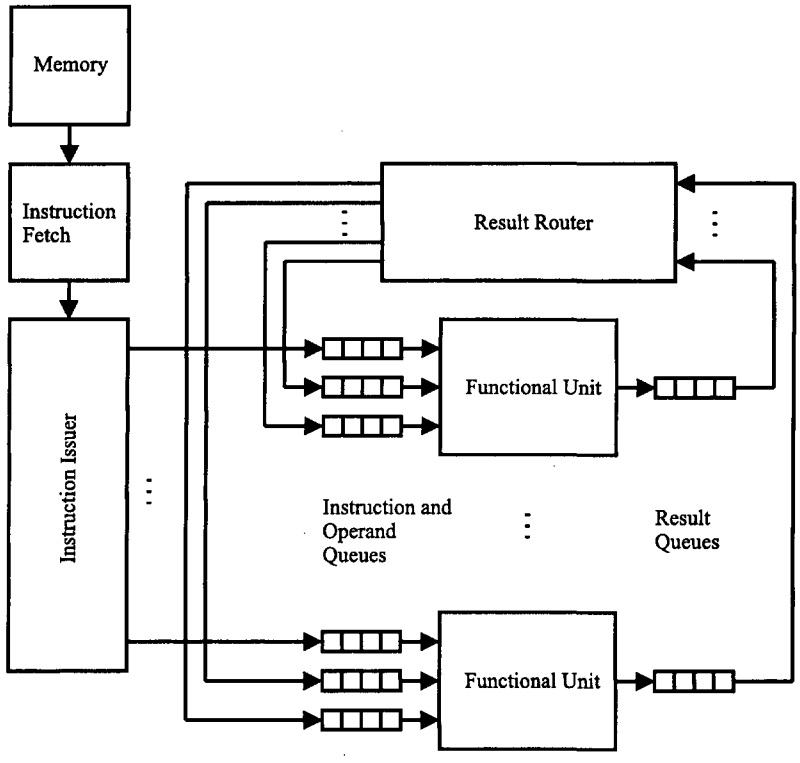

Figure 2: SCALP overall organization

3.2 AMULET

At the University of Manchester, several asynchronous processors called AMULET1, AMULET2, and AMULET3 were implemented [8, 9, 10, 20].

AMULET1 was the first asynchronous implementation of a commercially important instruction set architecture (ARM's instruction set architecture version 3 used in the ARM6 processor) [8], and appeared in early 1993 (see Fig. 3). It was designed using a 2-phase bundled data design style, with a 5-stage pipeline and no result forwarding. An interlocking was used to stall instructions at the register read stage until their operands had been written by previous instructions. After being fetched, a branch instruction had to pass through ten pipeline or FIFO stages before the target address was sent to memory. This resulted in large numbers of prefetched instructions that were discarded and a significant number of bubbles. As a result, the pipeline throughput was low. AMULET1 permitted out-of-order completion of load instructions relative to other instructions. AMULET1 was about 70 % the speed of a 20 MHz ARM6 processor, but with faster simple operations (e.g., with three times faster multiplication).

In October 1996, the AMULET2 processor was designed [9], based on the ARM instruction set architecture version 4. The processor used a 4-phase bundled data design style because this was believed to have benefits in terms of speed, size, and power relative to AMULET1. The processor had a slightly shorter pipeline than AMULET1 and employed both forwarding and branch prediction. It also incorporated limited forwarding by employing a last-result register at the output of the ALU, and forwarding mechanisms to use the result of a load instruction in a following instruction. A more sophisticated register-interlocking mechanism was used to remove write-after-write hazards. The AMULET2e chip consisted of 454 000 transistors including a 4 kbyte fully associative cache. The synchronous equivalent ARM810 used almost twice as many transistors, but also an 8 kbyte cache. With its 40 MIPS, the AMULET2 was 3.2 times faster than AMULET1, and with half the performance of a 75 MHz ARM810.

The next in the AMULET line, AMULET3, is expected to be a commercial product in 1999 [10]. It is expected that the key feature of this microprocessor will be a reorder buffer capable of solving the problems of result forwarding and exception handling in an asynchronous pipeline. This will allow a high degree of flexibility in operation, such as out-of-order completion, whilst avoiding read-after-write hazards (by stalling until the relevant value appears) and write-after-write hazards (averted by the reorder buffer).

3.3 Caltech Asynchronous Processor

The Caltech Asynchronous Processor (CAP) [11] was built in the late 1980s at California Institute of Tech-

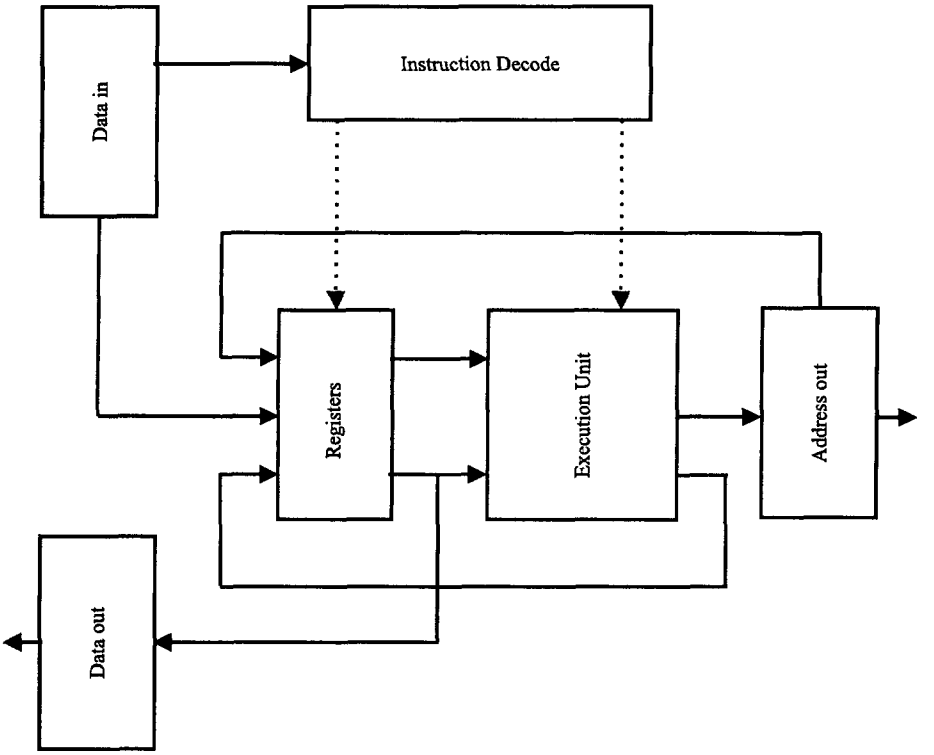

Figure 3: The AMULET1 organization

nology. The circuit design was delay-insensitive with dual-rail encoded communication. The processor featured a RISC-like load/store instruction set with 16 registers. A number of concurrent processes were responsible for instruction fetch, operand read, ALU operate, etc. The processor was implemented in a 1.6 micron CMOS process, and operated at 18 MIPS at room temperate and 5 V. The circuit continued to function at very low supply voltages, with optimum energy per operation at around 2 V. It was also tested in liquid nitrogen at 77 K when its performance reached 30 MIPS. More recently, GaAs version of CAP has been implemented [21].

3.4 Fully Asynchronous Microprocessor

The 32-bit Fully Asynchronous Microprocessor (FAM) [3] developed in the early 1990s at the Korean Institute of Science and Technology and the Tokyo Institute of Technology, was a dual-rail asynchronous processor with a RISC-like load/store instruction set. It had a 4-stage pipeline but register read, ALU, and register write occurred in a single stage eliminating the need for any forwarding. FAM, like CAP, is experimental.

3.5 Self-Timed RISC Processor

A Self-Timed RISC Processor (STRiP) [5] was built at Stanford University. Its instruction set was that of the MIPS-X processor. STRiP is included here even though it has a global clock signal and could be considered synchronous. It is unusual in that the speed of the global clock is dynamically variable in response to the instructions being executed, giving much of the advantage of an asynchronous system. The performance of STRiP was typically twice that of an equivalent synchronous processor. By maintaining global synchrony STRiP was able to implement forwarding in the same simple way as synchronous processors. For a 2 micron technology, the designers reported a 63 MHz effective clock frequency and a 62.5 MIPS performance rating.

3.6 ST-RISC

ST-RISC [4] was an architecture from the Israel Institute of Technology. It was delay-insensitive with a dual-rail datapath. ST-RISC had a 2-stage pipeline (fetch and execute) and a 3-address-register-based instruction set.

3.7 Nonsynchronous RISC

The Nonsynchronous RISC (NSR) processor [1] was built using FPGA technology at the University of Utah in 1993. It was implemented using a 2-phase bundled data protocol. NSR was a pipelined processor with pipeline stages separated by FIFO queues. The idea of the FIFO queues is that they decouple the pipeline stages so that an instruction that spends a long time in

one stage need not hold up any following instructions. The disadvantage of this approach is that the latency of the queues themselves is significant and, because of the dependencies within a processor pipeline, the increase in overall latency is detrimental. NSR used a locking mechanism to stall instructions that need operands produced by previous instructions. There was no forwarding mechanism. NSR had a 16-bit datapath and 16-bit instructions. The instructions included three 4-bit register specifiers and a 4-bit opcode. Some aspects of its instruction set were specialized for the asynchronous implementation: branch, load, and store instructions made the FIFOs that interconnected the functional units visible to the programmer. Conditional branch instructions were decoupled from the ALU that executed comparison instructions by a Boolean FIFO. Computed branch instructions also used a FIFO to store computed branch addresses. Load instructions had two parts. One instruction specifies a load address. A subsequent instruction used the load result by reading from a special register `r1`. There could be an arbitrary separation between the two instructions, and it was possible to have several load operations outstanding at one time. Store instructions worked similarly by writing the store data to `r1`.

3.8 Counterflow Pipeline Processor

The Counterflow Pipeline Processor (CFPP) [17], developed in 1994 at Sun Microsystems, was based on extensions of the techniques proposed in [18]. The CFPP executed SPARC instructions. Its novel contribution was the result forwarding in an asynchronous pipeline. CFPP had two pipelines. In one pipeline, instructions flowed upwards; in the other results flowed downwards. As instructions flowed upwards they watched out for results of previous instructions that they had to use as operands flowing downwards. If they spotted any such operands, they captured them; otherwise, they eventually received a value that flowed down from the register file which was at the top of the pipelines. When an instruction obtained all of its operands it continued to flow upwards until it found a pipeline stage where it could compute the result. Once computed, the result was injected into the result pipeline for use by any following dependent instructions and was also carried forward in the instruction pipeline to be written into the register file. The counterflow pipeline neatly solved the problem of result forwarding. This particular CFPP was not fabricated.

3.9 TITAC

TITAC-1 [13] was a simple 8-bit asynchronous processor built at the Tokyo Institute of Technology. It was based on the quasi delay-insensitive (QDI) timing model (where additional assumptions are introduced into the delay-insensitive model) and so had to use dual-rail encoding communication. This resulted in about twice as many gates in the datapath compared to an equivalent synchronous datapath. Architecturally, TITAC-1 was very straightforward with no pipelining and a simple accumulator-based instruction set.

In 1997, a 32-bit asynchronous microprocessor TITAC-2 [19] was built whose instruction set architecture was based on the MIPS R2000. It uses a scalable delay-insensitive (SDI) model, which unlike the QDI model, assumes that the relative delay ratio between any two components is bounded. SDI circuits can run faster than equivalent QDI ones. The measured performance of TITAC-2 was 52.3 MIPS using the Dhrystone benchmark.

3.10 Fred

Fred [15] is a development of NSR and also built at the University of Utah. Like NSR, Fred is implemented using 2-phase data bundling. It is modeled using VHDL. Fred extends the NSR to have a 32-bit datapath and 32-bit instructions, based on the Motorola 88100 instruction set. Fred has multiple functional units. Instructions from the functional units can complete out of order. However, the instruction issuer can only issue one instruction at a time, and the register file is only able to provide operands for one instruction at a time. This allows for a relatively straightforward instruction issue and a precise exception mechanism, but limits the attainable level of parallelism. There is no forwarding mechanism; instructions are stalled at the instruction issuer until their operands have been written to the register file. There is no out-of-order issue. Like the NSR, Fred uses programmer-visible FIFO queues to implement decoupled load/store and branch instructions. This arrangement has the possibility of deadlock if the program tries to read from an empty queue or write to a full one. Fred chooses to detect this condition at the instruction issuer and generate an exception.

3.11 Hades

Hades [6] is a proposed superscalar asynchronous processor from the University of Hertfordshire. It is in many ways similar to a conventional synchronous superscalar processor; it has a global register file, forwarding, and a complex (though in-order) instruction issue. Its forwarding mechanism uses a scoreboard to keep track of which result is available and from where.

3.12 ECSTAC

ECSTAC [12] is an asynchronous microprocessor designed at the University of Adelaide. ECSTAC is im-

plemented using fundamental mode control circuits. It is deeply pipelined with a complex variable-length instruction format. It has 8-bit registers and ALU. The variable-length instructions and the mismatch between the address size and the datapath width made the design more complex and slower. There is no forwarding mechanism within the datapath, and a register interlocking scheme is used to stall instructions until their operands are available.

3.13 ARISC

A joint project between the Technical University of Denmark and LSI Logic Denmark resulted in ARISC [2], which is an asyncronous re-implementation of the LSI Logic's TinyRISC TR4101 embedded microprocessor. Four-phase bundled data protocol is used throughout the entire design in combination with a normally opaque latch controller. All logic is implemented using static logic standard cells. In 0.35 micron CMOS technology ARISC performance is 74 MIPS (with power efficiency 635 MIPS/W), while the performance of the 83 MHz TR4101 is 48 MIPS (539 MIPS/W).

3.14 ASPRO-216

ASPRO [14] is a CMOS standard-cell QDI microprocessor which is being developed at the ENST Bretagne and CNET Grenoble. It is based on a simple RISC architecture with 16 general purpose registers. ASPRO-216 is a scalar processor, which extensively uses an overlapping pipelined execution scheme involving asynchronous units. The processor issues instructions in order and may complete their execution out of order. In 0.25 micron technology the expected peak performance is 200 MIPS with 0.5 W power dissipation.

4 Conclusions

Power consumption in VLSI chips may be a significant issue for two reasons. First, to optimize battery life, portable equipment such as lap-top computers and mobile telephones demand low power consumption. Second, high performance processors now dissipate enough power to make chip cooling a problem in an office environment. Thus, it has been predicted that asynchronous techniques will find their way into certain niches, in particular, embedded applications where the work required is extremely burst-intensive or where power-saving requirements make the approach attractive. Clocked chips with some asynchronous parts may also be expected. The asynchronous processor paradigm has the potential to solve the clocking problems in large processor chips. As a result, several universities and microprocessor manufacturers are actively investigating new asynchronous processor architectures.

References

[1] E. Brunvand: The NSR processor. *Proc. 26th Annual Hawaii International Conference on System Sciences*, (1993), pp. 428–435.

[2] K.T. Christensen et al.: The design an asynchronous TinyRISC TR4101 microprocessor core. *Proc. 4th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, (1998).

[3] K.-R. Cho, K. Okura, K. Asada: Design of a 32-bit fully asynchronous microprocessor (FAM). *Proc. 35th Midwest Symposium on Circuits and Systems*, (1992).

[4] I. David, R. Ginosar, M. Yoeli: Self-timed architecture of a reduced instruction set computer. *Proc. IFIP Working Conference on Asynchronous Design Methodologies*, (1993).

[5] M.E. Dean: Strip: A self-timed RISC processor. Technical Report CSL-TR-92-543, Stanford University, 1992.

[6] C.J. Elston et al.: Hades – Towards the design of an asynchronous superscalar processor. *Proc. 2nd Working Conference on Asynchronous Design Methodologies*, (1995), pp. 200–209.

[7] P.B. Endecot: SCALP: A superscalar asynchronous low-power processor. Ph.D. Thesis, University of Manchester, 1996.

[8] S.B. Furber et al.: A micropipelined ARM. *Proc. FIP TC10/WG 10.5 International Conference on Very Large Scale Integration*, (September, 1993), pp. 5.4.1–5.4.10.

[9] S.B. Furber et al.: AMULET2e. *Proc. EMSYS'96 - OMI 6th Annual Conference*, (September, 1996).

[10] S.B. Furber, J.D. Garside, D.A. Gilbert: AMULET3: A high-performance self-timed ARM microprocessor. *Proc. IEEE International Conference on Computer Design*, (October 1998).

[11] A.J. Martin et al.: The design of an asynchronous microprocessor. *Proc. Decennial Caltech Conference on VLSI*, (1989), pp. 351–373.

[12] S.V. Morton, S.S. Appleton, M.J. Liebelt: ECSTAC: A fast asynchronous microprocessor. *Proc. 2nd Working Conference on Asynchronous Design Methodologies*, (1995), pp. 180–189.

[13] T. Nanya et al.: TITAC: Design of a quasy-delay-insensitive microprocessor. *IEEE Design and Test of Comp.*, **11**(2):50–63, 1994.

[14] M. Renaudin, P. Vivet, F. Robin: ASPO-216: A standard-cell Q.D.I. 16-bit RISC asynchronous microprocessor. *Proc. 4th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, (1998).

[15] W.F. Richardson, E. Brunvand: Fred: An architecture for a self-timed decoupled computer. Technical Report UUCS-95-008, University of Utah, 1995.

[16] J. Šilc, B. Robič, T. Ungerer: *Processor Architecture - From Dataflow to Superscalar and Beyond.* Springer-Verlag, Berlin, 1999.

[17] R.F. Sproull, I.E. Sutherland, C.E. Molnar: Counterflow pipeline processor architecture. *IEEE Design and Test of Comp.*, **11**(3), 1994.

[18] I.E. Sutherland: Micropipelines. *Comm. ACM* **32**(6):720–738, 1989.

[19] A. Takamura et al.: TITAC-2: An asyncronous 32-bit microprocessor based on scalable-delay-insensitive model. *Proc. IEEE International Conference on Computer Design*, (October 1997), pp. 288–294.

[20] G. Theodoropoulos, J.V. Wood: Building parallel distributed models for asynchronous computer architecture. *Proc. World Transputer Congress*, (September 1994), pp. 285–301.

[21] J.A. Tierno et al.: A 100-MIPS GaAs asynchronous microprocessor. *IEEE Design and Test of Comp.*, **11**(2):43–49, 1994.

[22] T. Werner, V. Akella: Asynchronous processor survey. *Computer*, **30**(11):67–71, 1997.

[23] The National Technology Roadmap for Semiconductors. SEMATECH Inc., Austin, TX, 1997.