The NSR Processor

E. Brunvand*
Department of Computer Science
University of Utah
Salt Lake City, UT, 84112

Abstract

The NSR (Non-Synchronous RISC) processor is a general-purpose computer structured as a collection of self-timed blocks that operate concurrently and communicate over bundled data channels in the style of micropipelines [3, 16]. These blocks correspond to standard synchronous pipeline stages such as Instruction Fetch, Instruction Decode, Execute, Memory and Register File, but each operates concurrently as a separate self-timed process. In addition to being internally self-timed, the units are decoupled through self-timed FIFO queues between each of the units which allows a high degree of overlap in instruction execution. Branches, jumps, and memory accesses are also decoupled through the use of additional FIFO queues which can hide the execution latency of these instructions. A prototype implementation of the NSR processor has been constructed using Actel FPGAs (Field Programmable Gate Arrays).

1 Introduction

As computer systems continue to grow in size and complexity, the challenges inherent simply in assembling the system pieces in a way that allows them to work together also grow. A major cause of the problems lies in the traditional synchronous design style in which all the system components are synchronized to a global clock signal. One solution is to use non-clocked *asynchronous* techniques or restricted versions of asynchrony known as *self-timed* [15]. Self-timed circuits avoid clock-related timing problems by enforcing a simple communication protocol between parts of the circuit. This protocol acts as a sort of local clock to synchronize pieces of the circuit, but does not rely on specific time intervals or extend to the entire circuit as a synchronous clock would. This local synchronization allows self-timed circuits to avoid many of the timing-related difficulties present in large synchronous systems. For example, simply distributing the clock signal throughout a large synchronous system can be a major source of difficulty. Clock skew is a major concern in a large system, and is becoming significant even within a single chip. At the chip level, more and more of the power budget is being used to distribute the clock signal and designing the clock distribution network can take a significant portion of the design time.

While there is a growing body of knowledge about how to build small asynchronous and self-timed systems [3, 16, 11, 5, 13, 12, 8], there are still very few real examples of large systems designed with these techniques [6, 7, 10]. To explore how self-timed techniques can be used in a larger system, and also to evaluate how these techniques might affect the basic architecture of a general purpose computer, we have designed a self-timed computer and built a prototype version using field programmable gate arrays (FPGAs).

The NSR (Non-Synchronous RISC[1]) processor is a general purpose computer structured as a collection of self-timed blocks. These blocks operate concurrently and cooperate by communicating with other blocks using self-timed communication protocols. The blocks that make up the NSR processor correspond to standard synchronous pipeline stages such as *Instruction Fetch, Instruction Decode, Execute (ALU), Memory Interface* and *Register File*, but each operates concurrently as a separate self-timed process. In addition to being internally self-timed, the units are decoupled through self-timed FIFO queues between each of the units which allows a high degree of overlap in instruction execution. The prototype NSR processor uses seven Actel field programmable gate arrays (FPGAs) with each of the pipeline stages using one or two of the FPGA chips. The processor is operational and is currently being used to gather information about the effectiveness of the self-timed architecture. This paper introduces the NSR processor in general terms. Further details about the NSR prototype may be found in another document [14].

*This work is supported in part by NSF award MIP-9111793

[1] Because the current instruction set has no explicit HALT instruction, NSR also stands for "Nantucket Sleigh Ride."

0-8186-1060-3425/93 $03.00 © 1993 IEEE

2 Self-Timed Systems

Self-timed circuits are a subset of a broad class of asynchronous circuits. Asynchronous circuits do not use a global clock for synchronization, instead they rely on the behavior and arrangement of the circuits to keep the signals proceeding in the correct sequence. In general these circuits are very difficult to design and debug without some additional structure to help the designer deal with the complexity. Traditional clocked synchronous systems are an example of one particular structure applied to circuit design to facilitate design and debugging. Important signals are latched into various registers on a particular edge of a special clock signal. Between clock signals information flows between the latches and must be stable at the input to the latches before the next clock signal. This structure allows the designer to rely on data values being asserted at a particular time in relation to this global clock signal.

Self-timed circuits apply a different type of structure to circuit design. Rather than let signals flow through the circuit whenever they are able as with an unstructured asynchronous circuit, or require that the entire system be synchronized to a single global timing signal as with clocked systems, self-timed circuits avoid clock-related timing problems by enforcing a simple communication protocol between circuit elements. This is quite different from traditional synchronous signaling conventions where signal events occur at specific times and may remain asserted for specific time intervals. In self-timed systems it is important only that the correct sequence of signals be maintained. The timing of these signals is an issue of performance that can be handled separately.

Recently, asynchronous circuits in general, and self-timed circuits in particular, are experiencing renewed interest by systems designers [3, 16, 11, 5, 13, 12, 8]. The self-timed techniques being explored, although distinct in many ways, share the common property of local, rather than global, synchronization.

Self-timed protocols are often defined in terms of a pair of signals that request or initiate an action, and acknowledge that the requested action has been completed. One module, the sender, sends a request event (*Req*) to another module, the receiver. Once the receiver has completed the requested action, it sends an acknowledge event (*Ack*) back to the sender to complete the transaction.

This procedure defines the operation of the modules which follows the simple principle of passing a token of some sort back and forth between two participants. Imagine that a single token is owned by the sending module. To issue a request event it passes that token to the receiver. When the receiver is finished with its processing it produces an acknowledge event by passing that token back to the sender. The sequence of events in this communication transaction is an alternating sequence of request and acknowledge events. The sequence of events in a communication transaction is called the protocol. In this case the protocol is simply for request and acknowledge to alternate, although in general a protocol may be much more complicated and involve many interface signals.

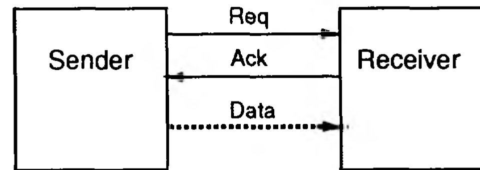

Figure 1: A Bundled Data Interface

Although testing and debugging of unstructured asynchronous circuits can be very difficult indeed, debugging self-timed circuits is facilitated because of the communication structure imposed on such circuits. One major debugging aid is the ability to stop the circuit at any point of communication by simply holding up the outgoing request signal. With the system paused, circuit information may be sensed or loaded using any of a number of standard techniques. Scan paths, for example, can be used to sense and load the data values internal to the circuit. In the absence of a global clock, individual request signals may also be used as a "clocking" signal for standard synchronous logic analyzers. Debugging such circuits does not seem to be dramatically more difficult than with synchronous circuits, but variations on standard techniques for designing testable circuits must be employed.

Although self-timed circuits can be designed in a variety of ways, the circuits used to build the NSR processor use *two-phase transition signalling* for control and a bundled protocol for data paths. Two-phase transition signalling [15, 3] uses transitions on signal wires to communicate the *Req* and *Ack* events described previously. Only the transitions are meaningful; a transition from low to high is the same as a transition from high to low and the particular state, high or low, of each wire is not important.

A bundled data path uses a single set of control wires to indicate the validity of a *bundle* of data wires. This requires that the data bundle and the control wires be constructed such that the value on the data bundle is stable at the receiver before a signal appears on the control wire. This condition is similar to, but weaker than, the equipotential constraint [15]. Two modules connected with a bundled data path are shown in Figure 1 and a timing diagram showing the sequence of the signal transitions using two-phase transition signalling is shown in Figure 2.

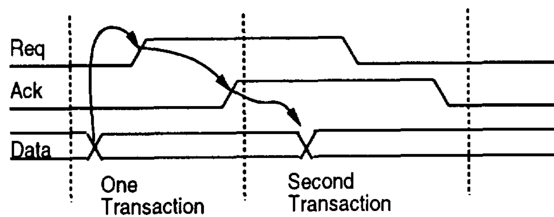

Figure 2: Bundled Transition Signaling

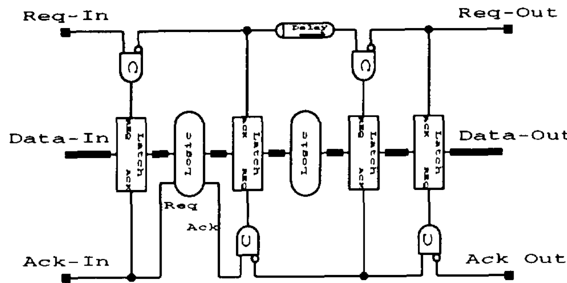

Figure 3: A micropipeline FIFO buffer

2.1 Micropipelines

First-in first-out (FIFO) buffers play an extremely important role in the NSR prototype. In fact, one way to look at the architecture of the NSR processor is as a large FIFO buffer that also modifies the data passing through it according to some rules. A self-timed FIFO buffer has a particularly simple circuit realization using the two-phase bundled protocol. The circuit in Figure 3 is an example of a FIFO buffer of this type with processing between two of the stages. This type of FIFO is also known as a *micropipeline* [16].

The *Req* and *Ack* signals in this circuit are transitions, and the data are contained in bundles. The controlling gates are C-elements. Notice that there is logic between the first two stages of the FIFO. If this logic is not internally self-timed and able to generate a completion signal, a delay must be added that models the delay of the data through that logic as shown in the figure. If no processing is present between the stages, as seen in the right two stages in the figure, the pipeline is a simple FIFO buffer.

3 NSR Architecture

The main blocks that make up the NSR processor are shown in Figure 4. These blocks are organized in the same way as the circuit in Figure 3. Each stage of the processor accepts data from its input, processes it in accordance with its function, and passes the result to the output in FIFO order. The thick lines in Figure 4 are bundled data paths, and the thin lines are pairs of request-acknowledge wires. Not shown in this figure are the FIFO queues that exist between each of the blocks. These queues decouple the stages of the NSR processor so that the occasional slow instruction does not hold up the entire machine.

Each of these blocks operates concurrently and performs a task roughly equivalent to its synchronous counterpart. The overall architecture of the NSR is inspired by the synchronous WM [17] and PIPE [9] processors that also use FIFO queues extensively.

A quick overview of the operation of the machine reveals typical instruction pipeline operation. The Instruction Fetch (IF) stage reads instructions from the Instruction Memory (IMem) and, unless they are branch or jump instructions, passes them to the Instruction Decode (ID) stage. The ID stage sends register addresses to the Register File (RF) and decoded instruction information to the Execute (EX) stage. The RF uses those addresses to send operands to the EX stage which performs some ALU operation. The result is written back to the RF. If the instruction computes addresses or data bound for the Data Memory (DMem), the result may also go to the Memory Interface (MEM), and if condition code (CC) bits or jump addresses are computed, the result may also go back to the IF stage.

Note that unlike a synchronous pipeline, if an instruction does not need the services of a particular pipeline stage, it need not pass through that stage. Branches and jumps are handled in the Instruction Fetch stage and never passed to the Instruction Decoder and are thus never seen by the rest of the pipeline. Instructions that do not deal with memory are never seen by the Memory interface, and instructions that do not write back to the Register File are not required to communicate with that stage.

3.1 NSR Instruction Set

The prototype NSR processor is a 16-bit machine and implements the simple instruction set shown in Figure 5. Most of the instructions are typical. The NSR is a load-store machine with all arithmetic and logic instructions operating only on registers. There are 16 general purpose 16-bit registers with register R0 tied to 0. Writing into register R0 has no effect. These operations are encoded as three-addresses instructions. Branches are relative to the current program counter (PC), and jumps use absolute addresses. Three different shift instructions implement logical right and left shift, and arithmetic right shift. Move-immediate instructions allow data in the instruction word to be loaded directly into the NSR's registers. The MVPC instruction allows the current PC value, plus a sign-extended offset contained in the instruction, to be moved to a regiater.

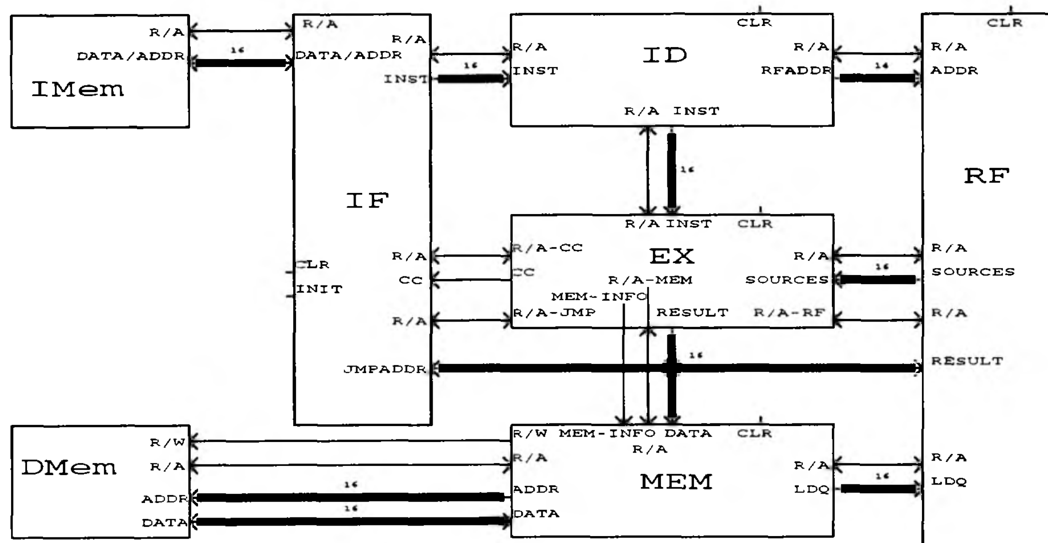

Figure 4: NSR Architecture Block Diagram

Mnemonic	Encoding	Action
ADD Rd,Ra,Rb	1100 -Rd- -Ra- -Rb-	Rd ← Ra + Rb
SJMP Rd,Ra,Rb	1101 -Rd- -Ra- -Rb-	Rd, Jmp-Queue ← Ra + Rb
LDA Rd,Ra,Rb	1110 -Rd- -Ra- -Rb-	Rd, AQ(load) ← Ra + Rb
STA Rd,Ra,Rb	1111 -Rd- -Ra- -Rb-	Rd, AQ(store) ← Ra + Rb
SUB Rd,Ra,Rb	0100 -Rd- -Ra- -Rb-	Rd ← Ra - Rb
AND Rd,Ra,Rb	1000 -Rd- -Ra- -Rb-	Rd ← Ra AND Rb
OR Rd,Ra,Rb	1001 -Rd- -Ra- -Rb-	Rd ← Ra OR Rb
XOR Rd,Ra,Rb	1010 -Rd- -Ra- -Rb-	Rd ← Ra XOR Rb
XNOR Rd,Ra,Rb	1011 -Rd- -Ra- -Rb-	Rd ← Ra XNOR Rb
BCND offset	0001 —offset—	If cc, PC ← PC + offset
JMP	0000 xxxx xxxx xxxx	PC ← Jmp-Queue
MVIH Rd,value	0010 -Rd- –value–	Rd.H ← value, Rd.L=0
MVIL Rd,value	0011 -Rd- –value–	Rd.L ← value, Rd.H=0
Scond Ra, Rb	0101 cond -Ra- -Rb-	CC-Queue ← cond bit
MVPC Rd	0111 -Rd- –offset–	Rd ← PC + offset
SHcode Rd,Rb	0110 -Rd- code -Rb-	Rd ← shifted Rb

Figure 5: NSR Instruction Set (16-bit Prototype)

The interesting parts of the NSR's instruction set involve the decoupling of the branches and jumps, and load and store instructions to the memory. These aspects of the instruction set are also inspired by the WM machine [17].

3.1.1 Decoupled Control Flow

All flow control decisions are made by the Instruction Fetch unit based on conditions set up in advance by the Execution unit. For example, BCND instructions are recognized by the Instruction Fetch unit and cause the program counter to either be incremented by one (branch not taken), or to be updated by adding a signed constant present in the opcode (branch taken). The decision to take the branch or not is made based on a condition code (CC) bit. This CC bit is computed in the Execute unit and stored in a FIFO queue between the Execute unit and Instruction Fetch unit. The CC bits generated by the Execute unit (Scond instructions) and used in the Instruction Fetch unit (BCND instructions) must obey a one-to-one producer-consumer relationship.

Note that the arithmetic instructions do *not* set the condition bit. These CC bits are only set by the explicit condition code setting instructions. These instructions compare the values contained in a pair of registers and set the condition code based on the result of that comparison. The prototype NSR processor implements EQ, NEQ, GT, and GE comparisons.

Jump instructions are also handled in the Instruction Fetch unit. In this case, the target address is computed by the Execute unit in advance by adding two registers (the SJMP instruction). This address is sent to a FIFO queue and eventually consumed by a JMP instruction. The Instruction Fetch stage, upon seeing a JMP instruction, dequeues an address and uses that to update the value of the PC. Note that the SJMP instruction is exactly the same as an ADD instruction except that it also causes the result of the addition to be queued in the jump-address queue. Again, the jump addresses and JMP instructions must obey the producer-consumer relationship. One easy way to halt the NSR processor in a deadlock is to issue a JMP instruction before any SJMP instruction. The Instruction Fetch unit will wait forever for the jump addresses to show up in the queue. Fortunately, a compiler is not likely to make such a mistake.

The effect of the decoupling of the branch and jump instructions is similar to the common idea of delay slots. However, rather than using a fixed number of delay slots,

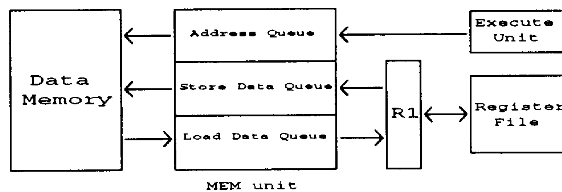

Figure 6: NSR Memory Queues

the programmer is free to put any number of instructions between, for example, the SNE instruction and the BCND that uses the generated condition code. If many instructions are issued between these two then the condition code will be waiting when the BCND is executed and there will be no stalling of the pipeline and no delay. If, on the other hand, the SNE is followed directly by the BCND, then the Instruction Fetch stage will simply wait for the condition code to be produced before proceeding with the branch. Note that since all the stages are self-timed, no explicit control of the pipeline is required to implement this possible stall and no NO-OP instructions are required to fill the delay slots.

3.1.2 Decoupled Memory Access

Memory access on the NSR is also decoupled through FIFO queues. There are, in fact, no standard load and store instructions in the NSR instruction set. Instead, memory addresses are computed and sent to the Memory Interface which processes the requests and queues up the results. An LDA instruction is exactly like an ADD instruction with the result also sent to the Memory Interface as an address to load from. The result of an STA instruction is likewise considered an address in which to store data.

The programmer manipulates data to and from the memory by accessing register R1, a special register which is actually connected to queues to and from the memory. When the program reads from register R1 (R1 is the source register for some operation) the result is data from memory out of the Load Data Queue (LDQ), and when the program stores into register R1 (R1 is the destination register of some operation), that data gets queued up to be stored into memory through the Store Data Queue (SDQ). These queues and another queue for the addresses (AQ) are shown in Figure 6.

The Memory Interface uses the information in these queues to perform memory cycles. When a load address is at the head of the AQ, a read cycle is initiated and the resulting data are placed in the LDQ. When a store address is at the head of the AQ, and there are data at the head of the SDQ, a store cycle is initiated and those data are stored to memory. Because the memory operations are decoupled, several requests may be queued before they are needed. For example, by placing an LDA instruction several instructions in advance of the instruction that requires the memory contents, the memory access latency is hidden. Again, this is similar to delayed loads with the advantage that any number (including zero) instructions may be executed between the initiation of the load and the use of the loaded data.

Note that each time an instruction uses register R1 as a source, it dequeues one word from the LDQ. This means that a different value may be received each time R1 is accessed. For example, if two LDA instructions have been issued previously, then the instruction ADD R2,R1,R1 will add the two values loaded from memory and store the result in R2. In fact, if an address has been queued with an STA instruction, the instruction ADD R1,R1,R1 will add two values from memory and store the result back to another memory location.

Interleaved STA and LDA instructions may be used without concern. Although the LDQ and SDQ are independent, there is only one Address Queue. In addition to enqueuing the address, a bit is enqueued which indicates whether the address is for a write or read operation. By sharing the AQ, read-after-write hazards are avoided. However, the unwary programmer can easily deadlock the NSR processor by issuing an instruction that uses R1 as a source before queuing up an address using an LDA instruction. The processor will stop and wait for the result from memory that will never arrive. Note that it is a simple matter for a compiler to avoid problems of this sort.

3.2 NSR Functional Units

Details of the individual blocks of the NSR are as follows:

Instruction Fetch The Instruction Fetch unit reads instructions from the instruction memory (IMem) and passes them to the Instruction Decode unit. It also holds the program counter (PC), and therefore processes branch and jump instructions directly. Jump addresses are generated in the Execute unit and passed to the Instruction Fetch unit through a single-place FIFO queue. Condition codes are likewise computed in the Execute unit and passed to the Instruction Fetch unit through a FIFO queue, this time 8 places deep.

Instruction Decode The Instruction Decode unit takes the instruction from the Instruction Fetch unit (through a FIFO queue) and decodes information for both the Register File and Execute units. The Register File receives register address information, and the Execute unit receives decoded instruction bits.

Execute Unit The Execute unit receives its instructions in pre-decoded form from the Instruction Decoder. It uses that information, and the necessary operands from the Register File, to produce a result. This result can be routed back to the Register File, to the Memory unit, or, in the case of computing a jump address or a condition code, to the Instruction Fetch unit.

The Execute unit uses a carry-completion-sensing adder in its ALU [3]. This form of adder senses when the addition is complete by looking at the carries at each stage of the adder and then generates a completion signal. This allows the time to complete an addition to depend on the data being added. A simple ripple-carry version of this adder will complete quickly on average and only slow down in the rare case that the carry must ripple through many of the bits. Because the NSR is self-timed, such variations in processing speed are not only handled gracefully, but encouraged as they allow the machine to run closer to average case speed than worst case speed.

Register File The Register File receives register address information from the Instruction Decoder. It passes operands to the Execute unit and receives results from the Execute unit. It also receives data from the Memory Interface through the Load Data Queue (LDQ). Data loaded from the data memory (DMem) are available by reading a special register, R1, in the Register File. When this register is accessed, the Register File requests data from the Memory unit. Data written to R1 are sent to the Store Data Queue (SDQ) in the Memory Interface to be stored in data memory.

To prevent write-after-read hazards in the Register File, each register has a single tag bit that tells when a write is pending on that register. Consider the instruction sequence ADD R2,R3,R4 followed directly by SUB R5,R6,R2. Because there are queues between the RF and EX units, it would be possible for the old value in register R2 to be queued up as an argument to the SUB instruction before the result of the ADD instruction is written back to the Register File. To prevent this, a bit is set for each destination address seen by the Register File. A register may not be read as long as this bit is set.

Memory Interface The Memory Interface to the NSR contains the FIFO queues shown in Figure 6. The LDA and SDA instructions queue up addresses into an Address Queue (AQ), and the data to be loaded or stored is also queued up by reading or writing to a special register. When the Memory Interface has both an address, and, in the case of a write, the necessary data, a memory access is initiated. If the operation is a load, the data from memory is queued up and is available by reading a special register. This memory queue organization is similar to the WM machine [17], and the PIPE processor [9]. Our version is shown in Figure 6.

Table 1: NSR FPGA Implementation

System Piece	Chips Used	Logic Modules	Percent Utilization
Instruction Fetch	1 Actel 1020A	547	100%
Instruction Decode	1 Actel 1010A	287	97%
Execute	1 Actel 1020A	518	95%
Register File	2 Actel 1020A	1076	98%
Memory Interface	2 Actel 1010A	554	94%

4 NSR FPGA Implementation

The processes that implement the separate pieces of the prototype NSR processor are each implemented using Actel FPGAs. The two-phase transition control modules and bundled data modules have been assembled from a library of macros designed to be used with the Actel parts [2, 1]. The individual units of the NSR are designed to behave as pipeline stages that also process the information that flows through them [4, 3]. These parts were designed and implemented by students in a graduate seminar on VLSI architecture using the Workview suite of schematic capture and simulation tools from ViewLogic.

The resulting FPGAs have been assembled as a wire-wrapped prototype for testing and evaluation. The number of Actel FPGA chips used to implement each of the parts of the NSR and the utilization of those chips are shown in Table 1. The NSR processor is connected to a standard PC to allow programs to be loaded into the NSR's memory and data to be retrieved to the PC for analysis.

Although the 16-bit prototype of the NSR processor has 16 general purpose registers, there was not enough space on the FPGAs to implement all of these registers. Register R0 is already defined to be tied to 0, and for the FPGA prototype, register R14 is tied to 1 and register R15 is tied to -1. Recall that register R1 is actually a special purpose register that acts as the interface to the memory. This leaves registers R2-R13 as general purpose 16-bit registers.

Because the entire processor is self-timed, stopping at any stage of the execution is possible simply by delaying a control transition. To use this feature as an aid in testing the machine, we installed switches that block the outgoing *Req* signal from each pipeline stage of the NSR processor. We also installed lights on the *Req* and *Ack* signals between each stage. This allows us to hold up instructions at the output of any stage, or to single-step instructions through

the machine while monitoring the state of the communication at each bundled data interface. This turned out to be quite useful during debugging of the prototype.

Memory presented another problem. Because the processor is self-timed, it expects to send out a memory request, and receive and acknowledge when the data is available. Commercial memory chips unfortunately do not provide such an acknowledge signal. We use digital delay chips to delay the outgoing request transition long enough to account for the delay through the memory chips. This delayed request becomes the acknowledge to the NSR processor. This allows us to use standard static RAM chips in the NSR prototype.

The FPGAs for the prototype were finished in Autumn Quarter 1991. The prototype board was wire-wrapped in Winter Quarter 1992, and the processor was assembled, tested, and debugged in the first part of Spring Quarter 1992. The NSR prototype is fully functional and test programs are now being written and run to evaluate the architecture. Performance results are difficult to express because the number of instructions per second will depend on the mix of instructions being executed. Preliminary results for this FPGA-based prototype indicate that the best case speed is on the order of 1.3 MIPS, although this is a rather meaningless performance metric. The FPGAs themselves are major culprits in the relatively slow instruction times. Measurements indicate that a 16-bit addition (from the time the operands are presented at the input to the chip, until the completion acknowledgment is generated as the chip output) takes between 225ns and 320ns depending on the length of the carry in the adder. This time is not completely due to the adder as it includes the delay of latching the input data, completing the addition, and latching the result in the output register.

5 Conclusions

Using a library of self-timed modules, a prototype of a self-timed general purpose processor was constructed. Further details about the prototype may be found in another document [14]. We are very pleased with the results of this work and plan to continue to investigate the potential of large self-timed systems. This processor has many novel features and the FPGA implementation is being used to gain experience with the architecture before starting to build a larger 32-bit version of the processor in semicustom CMOS. Major additions to the next version of this processor, in addition to the increase in word width, will include an interrupt structure, and I/O subsystems to allow the processor to communicate with other, more standard, peripheral devices.

We have found that field programmable gate arrays are an excellent medium for fast inexpensive system prototyping provided the necessary circuit primitives can be implemented. The Actel FPGA, and the self-timed library, have proven to be a very useful and flexible tool.

6 Acknowledgments

Many thanks to my students in the autumn 1991 VLSI Architecture class for designing and implementing the FPGA components that are the NSR processor. They are: IF stage - John Hurdle, Lüli Josepheson, ID - Ajay Khoche, Bill Richardson, Michael Stephenson, RF - Kent Bunker, V. Chandramouli, Dewey Jones, EX - Corby Bacco, Prabhat Jain, Cliff Miller, MEM - Madhu Penugonda, Marshall Soares. Thanks also to Nick Michell and Bill Richardson for helping define the NSR architecture.

References

[1] Erik Brunvand. A cell set for self-timed design using actel FPGAs. Technical Report UUCS–91–013, University of Utah, 1991.

[2] Erik Brunvand. Implementing self-timed systems with FPGAs. In W. R. Moore and W. Luk, editors, *FPGAs*, chapter 6.2, pages 312–323. Abingdon EE&CS Books, 1991.

[3] Erik Brunvand. *Translating Concurrent Communicating Programs into Asynchronous Circuits.* PhD thesis, Carnegie Mellon University, 1991. Available as Technical Report CMU-CS-91-198.

[4] Erik Brunvand and Robert F. Sproull. Translating concurrent programs into delay-insensitive circuits. In *ICCAD-89*, pages 262–265. IEEE, November 1989.

[5] S. Burns. Automated compilation of concurrent programs into self-timed circuits. Master's thesis, Caltech, 1987.

[6] Wesley A. Clark. Macromodular computer systems. In *Spring Joint Computer Conference*. AFIPS, April 1967.

[7] A.L. Davis. The architecture and system method for DDM1: A recursively structured data-driven machine. In *5th Annual Symp. on Computer Architecture*, April 1978.

[8] Jo C. Ebergen. *Translating Programs into Delay-Insensitive Circuits.* PhD thesis, Technische Universiteit Eindhoven, 1987.

[9] J. R. Goodman, J Hsieh, K Liou, A. R. Pleszkun, P. B. Schechter, and H. C. Young. PIPE: A VLSI decoupled architecture. In *12th Annual International Symposium on Computer Architecture*, pages 20–27. IEEE Computer Society, June 1985.

[10] Alain Martin, Steven Burns, T.K. Lee, Drazen Borkovic, and Pieter Hazewindus. The design of an asynchronous microprocessor. In *Proc. Cal Tech Conference on VLSI*, 1989.

[11] Alain J. Martin. Compiling communicating processes into delay insensitive circuits. *Distributed Computing*, 1(3), 1986.

[12] T. H.-Y. Meng, R. W. Broderson, and D. G. Messerschmitt. Design of clock-free asynchronous systems for real-time signal processing. In *ICASSP-89*, pages 2532–5. IEEE, May 1989.

[13] Cees Niessen, C.H. (Kees) van Berkel, Martin Rem, and Ronald W.J.J. Saeijs. VLSI programming and silicon compilation; a novel approach from Philips research. In *ICCD*, Rye Brook, NY, October 1988.

[14] William F. Richardson and Erik L. Brunvand. The NSR processor prototype. Technical Report UUCS-92-029, University of Utah, 1992.

[15] C. L. Seitz. System timing. In *Mead and Conway, Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.

[16] Ivan Sutherland. Micropipelines. *CACM*, 32(6), 1989.

[17] Wm. A. Wulf. The WM computer architecture. *Computer Architecture News*, 16(1), March 1988.