The NSR Processor Prototype

William F. Richardson and Erik Brunvand

UUCS-92-029

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

August 14, 1992

Abstract

The NSR (Non-Synchronous RISC) processor is a general purpose processor structured as a collection of self-timed units that operate concurrently and communicate over bundled data channels in the style of micropipelines. These units correspond to standard synchronous pipeline stages such as Instruction Fetch, Instruction Decode, Execute, 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. Branches, jumps, and memory accesses are also decoupled through the use of additional FIFO queues which can hide the execution latency of these instructions. The prototype implementation of the NSR has been constructed using Actel FPGAs (Field Programmable Gate Arrays).

This research was sponsored in part by NSF award MIP-9111793

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. For example, simply distributing the clock signal throughout a large synchronous system can be a major source of complication. Clock skew is a serious 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, while designing the clock distribution network can take a significant portion of the design time. One solution is to use non-clocked *asynchronous* techniques or restricted versions of asynchrony known as *self-timed* [8].

1.1 Self-Timed Systems

Self-timed circuits are a subset of a broad class of asynchronous circuits. General asynchronous circuits do not use a global clock for synchronization, but instead 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.

1.2 Communication Protocol

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

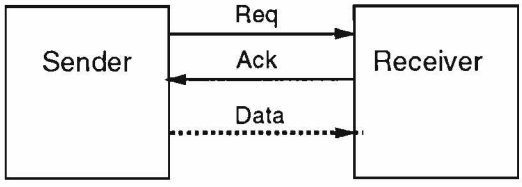

Figure 1: A Bundled Data Interface

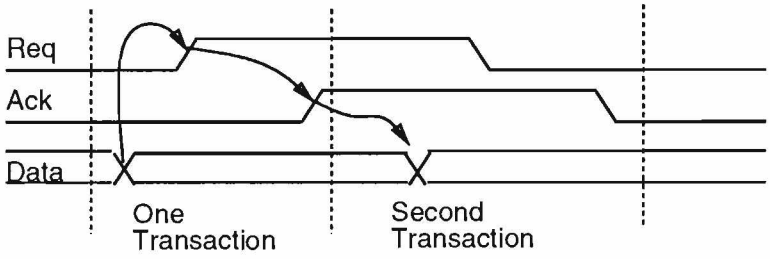

Figure 2: Two-Phase Bundled Transition Signaling

This procedure defines the operation of the modules which follows the common idea 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.

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 [8, 4] uses transitions on signal wires to communicate the request and acknowledge 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 [8]. 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.

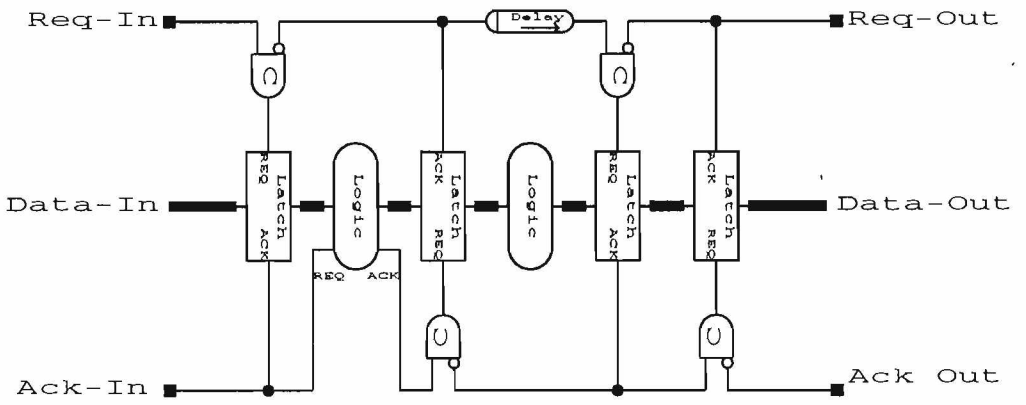

Figure 3: A Micropipeline FIFO Buffer

A self-timed FIFO buffer has a particularly simple implementation 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. If the processing 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. This type of FIFO is also known as a *micropipeline* [9].

2 NSR Architecture

The NSR (Non-Synchronous RISC[1]) processor prototype is structured as a collection of self-timed units which operate concurrently and cooperate by communicating with other units using self-timed communication protocols.

First-in first-out (FIFO) buffers play an extremely important role in the implementation of the NSR. 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. The overall architecture of the NSR is inspired by the synchronous WM [10] and PIPE [7] processors which also use FIFO queues extensively.

The units that make up the NSR processor correspond to standard synchronous pipeline stages, and consist of Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Register File (RF), and Memory Interface (MEM) as shown in Figure 4. Each unit operates concurrently as a separate self-timed process. In addition to being internally self-timed, the units are decoupled

[1] Because the current implementation has no explicit HALT instruction and no interrupt mechanism, NSR originally stood for "Nantucket Sleigh Ride."

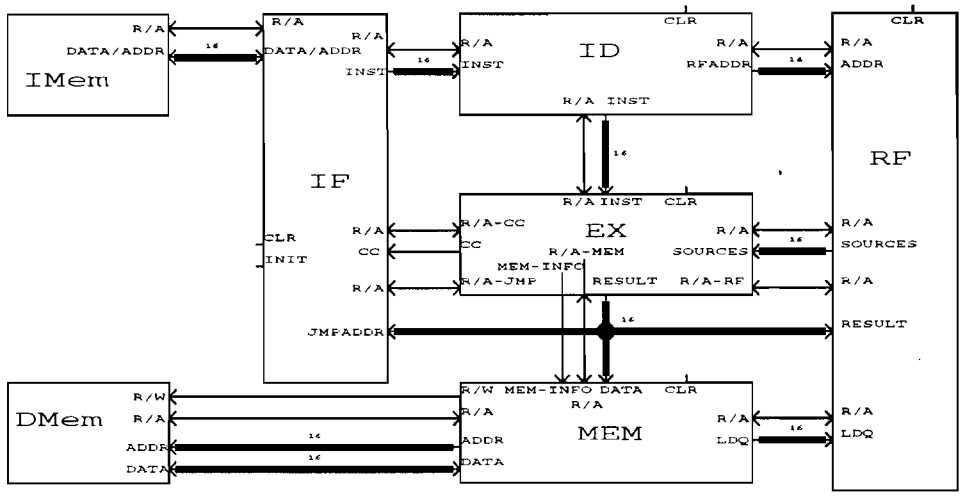

Figure 4: The NSR processor

through self-timed FIFO queues between each adjacent unit, which allows a high degree of overlap in instruction execution.

The prototype implementation of the NSR is a 16-bit pipelined RISC processor. It has 16 general purpose registers, uses a triadic register addressing scheme, and provides decoupled branches and memory accesses. There are separate busses for instructions and data, although the memory address space is shared. The hardware consists of seven Actel field programmable gate arrays (FPGAs) with each of the pipeline stages using one or two of the FPGA chips.

2.1 Instruction Set

There are several considerations which influence the choice of instruction set. Largely due to space considerations and pin limitations, the NSR is limited to 16-bit data paths. In order to facilitate the decoupling of functional units within the NSR, a triadic register addressing method is used. To keep the instruction pipelining as simple as possible, each complete instruction consists of a single 16-bit word. These factors limit the opcodes available to the NSR to a small set. The sixteen instructions chosen are the minimum needed to provide general purpose computations while maintaining some flexibility and ease of use. An early proposal included additional opcodes for arithmetic carry operations, immediate operands, byte addressing, and interrupt handling, but was judged to be too complex for a first attempt. Figure 5 shows the current instruction set of the NSR.

Encoding	Mnemonic		Action
1111 -Rd- -Ra- -Rb-	STA	Rd,Ra,Rb	Rd,AQ (Store) ← Ra + Rb
1110 -Rd- -Ra- -Rb-	LDA	Rd,Ra,Rb	Rd,AQ (Load) ← Ra + Rb
1101 -Rd- -Ra- -Rb-	SJMP	Rd,Ra,Rb	Rd,Jmp-Queue ← Ra + Rb
1100 -Rd- -Ra- -Rb-	ADD	Rd,Ra,Rb	Rd ← Ra + Rb
1011 -Rd- -Ra- -Rb-	XNOR	Rd,Ra,Rb	Rd ← Ra XNOR Rb
1010 -Rd- -Ra- -Rb-	XOR	Rd,Ra,Rb	Rd ← Ra XOR Rb
1001 -Rd- -Ra- -Rb-	OR	Rd,Ra,Rb	Rd ← Ra OR Rb
1000 -Rd- -Ra- -Rb-	AND	Rd,Ra,Rb	Rd ← Ra AND Rb
0111 -Rd- -offset-	MVPC	Rd,offset	Rd ← PC + offset
0110 -Rd- 0100 -Rb-	SHRA	Rd,Rb	Rd ← shift right arithmetic Rb
0110 -Rd- 0010 -Rb-	SHRL	Rd,Rb	Rd ← shift right logical Rb
0110 -Rd- 0001 -Rb-	SHLL	Rd,Rb	Rd ← shift left logical Rb
0101 11xx -Ra- -Rb-	SNE	Ra, Rb	CC-Queue ← (Ra ≠ Rb)
0101 10xx -Ra- -Rb-	SGE	Ra, Rb	CC-Queue ← (Ra ≥ Rb)
0101 01xx -Ra- -Rb-	SGT	Ra, Rb	CC-Queue ← (Ra > Rb)
0101 00xx -Ra- -Rb-	SEQ	Ra, Rb	CC-Queue ← (Ra = Rb)
0100 -Rd- -Ra- -Rb-	SUB	Rd,Ra,Rb	Rd ← Ra − Rb
0011 -Rd- -value-	MVIL	Rd,value	Rd.h ← 00, Rd.l ← value
0010 -Rd- -value-	MVIH	Rd,value	Rd.h ← value, Rd.l ← 00
0001 —offset—	BCND	offset	if CC-Queue then PC ← PC + offset
0000 xxxx xxxx xxxx	JMP		PC ← Jmp-Queue

Figure 5: NSR Instruction Set

2.2 Control Flow

All control flow decisions are made by the Instruction Fetch unit based on conditions set up in advance by the Execution unit. Conditional branch (`BCND`) instructions and jump (`JMP`) instructions are handled and consumed entirely by the IF unit and do not proceed any further though the NSR pipeline. The semantic convention used is that branches implement an offset relative to the program counter (PC) while jumps are made to a specific address.

`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 advance by the Execute unit and stored in a FIFO queue between the Execute unit and Instruction Fetch unit.

Note that the arithmetic instructions do *not* set the condition bit. These CC bits are set only 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.

Each `BCND` instruction consumes one CC bit from the CC queue in order to make the branch decision. Thus, the CC bits generated by the Execute unit and used in the Instruction Fetch unit must obey a one-to-one producer-consumer relationship.

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 the contents of two registers with the SJMP instruction and sending the result to a FIFO queue. The Instruction Fetch stage, upon seeing a JMP instruction, dequeues an address from the Jmp-Quene and uses it to update the value of the PC. The jump addresses and JMP instructions must also 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, in which case the Instruction Fetch unit will wait forever for the jump address to show up in the queue.

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, 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.

2.3 Memory Access

The memory address space consists of 65536 16-bit words, addressed sequentially from 0x0000 to 0xFFFF. For this prototype version of the NSR, the smallest (and indeed only) addressable memory element of the NSR is a 16-bit word.

Memory access on the NSR is 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 transfers data between the NSR and 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). Neither operation takes place until the corresponding address has been placed into the Address Queue (AQ) The memory access queues 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 initi-

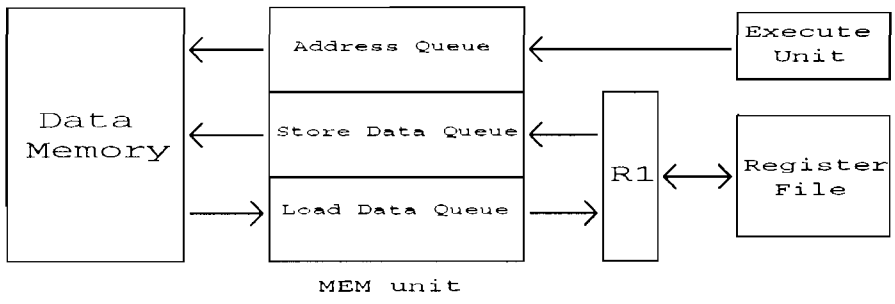

Figure 6: NSR Memory Queues

ated 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. As with control flow operations, no explicit control of the pipeline is needed to generate possible stall cycles.

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 also 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 compilers to avoid this problem.

System Unit	Chips Used	Logic Modules	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	538 each	98%
Memory Interface	2 Actel 1010A	277 each	94%

Figure 7: NSR FPGA Implementation

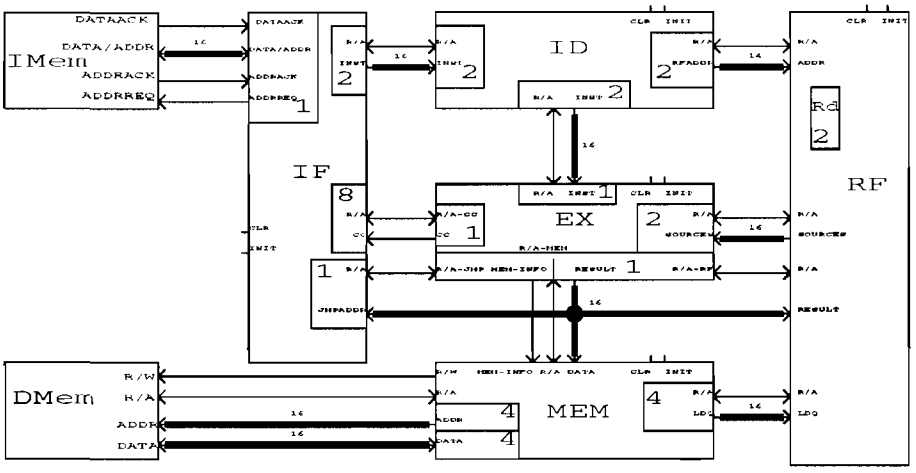

Figure 8: FIFO Queue Lengths

3 Prototype Implementation

The separate functional units 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 [3, 2]. The individual units of the NSR are designed to behave as pipeline stages that also process the information that flows through them [5, 4]. 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 Figure 7. The NSR processor is connected to a standard PC clone to allow programs to be loaded into the NSR's memory and data to be retrieved to the PC for analysis.

The individual units of the NSR are connected with self-timed FIFO pipelines to provide a higher degree of overlap in instruction execution. The length of each FIFO is not a factor in insuring correct operation, but may become significant in improving the throughput of the processor. The actual length of each FIFO was determined by the amount of space left over on each FPGA after the essential functions were implemented. The block diagram of the NSR functional units along with the length and location of the connecting FIFOs is shown in Figure 8.

3.1 Instruction Fetch

The Instruction Fetch unit is responsible for maintaining the program counter (PC), fetching instructions from memory, and passing executable instructions on to the next stage of the NSR to be decoded and executed. As described earlier, the IF unit detects and handles all control flow instructions directly. The IF unit reads instructions from the NSR memory. There are then several actions which may be taken by the IF unit. If the opcode corresponds to a jump instruction, an address is taken from the Jmp-Queue and the program counter is loaded with that address. If the opcode is a conditional branch, a bit is taken from the CC-Queue, and if the bit indicates that a branch should occur a 12-bit signed offset obtained from the opcode is added to the program counter. Otherwise, the program counter is incremented by one to fetch the next sequential instruction. The jump and branch instructions go no further through the NSR. All other opcodes are passed on unchanged to the Instruction Decode unit for further action. This action is similar to the concept of "squashing" instructions, often found in synchronous processors. However, the NSR does not convert the branch and jump instructions into NO-OPs, but instead removes them completely from the main processor pipeline.

There is one additional opcode which is recognized by the IF unit for special handling. Since the program counter is only stored in the IF unit, and all other units operate independently, there would normally be no way to obtain a "current" instruction address for use as a return address in a subroutine. To deal with this case, the `MVPC` instruction causes the IF unit to send the Program Counter value to the next stage. Before passing the PC value on, an 8-bit signed offset is extracted from the `MVPC` opcode and added to the current PC value. The `MVPC` instruction is passed on to the ID unit, and is followed immediately afterwards by the modified PC value. The current PC value held in the IF unit is then incremented normally. The `MVPC` instruction does not alter the program counter value used to fetch the next instruction, but allows the programmer to obtain the modified address to be placed in a register for later use. It is the responsibility of the ID unit to recognize the `MVPC` instruction and handle the subsequent address accordingly.

The prototype board on which the NSR is built has only a single memory address space, but the NSR has separate logical paths for instructions and data. In order to share the access to the memory, a simple round-robin arbiter is used. The IF unit must take turns with the MEM unit when accessing the memory. In addition, due to pin restrictions, the interface to the instruction

Bits:	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Usage:	Destination				Source A				V	Source B				V

Figure 9: Register Usage Encoding

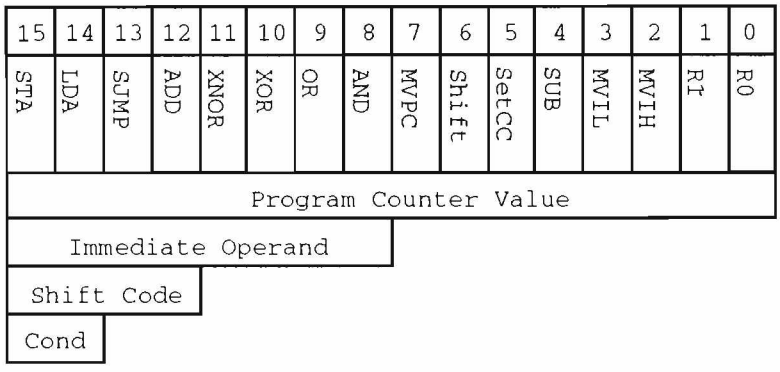

Figure 10: Execution Unit Operation Encoding

memory uses a 16-bit multiplexed bus, which carries both the address and the opcodes. The Instruction Fetch unit must provide the proper signals to fetch instructions from memory, and must obey the protocols established to ensure that the data memory unit can also access the system memory. Details on the memory arbitration will be found in section 3.6.

3.2 Instruction Decode

The ID unit has two responsibilities. It takes executable instructions from the IF unit and informs both the Execute unit and the Register File of what actions they must take. The Register File may provide two, one, or no register contents to the Execution unit for consumption. It must also be told whether to expect a result from the EX unit and to what register it should route that result. There are multistage FIFO queues between the ID, RF, and EX units. It is not necessary that all instructions be synchronized, but each unit must know how many operands and results are needed, and what to do with them when they arrive.

The source and destination information is encoded and sent to the Register File using the format shown in Figure 9. Since register R0 cannot be overwritten, it is often used as a destination when the result of an instruction does not need to be saved. The destination field is therefore all zeros when the result is not to be placed in a register. The source registers cannot be encoded in this way, since register R0 is a valid source, so they are encoded with a bit indicating their validity.

The EX unit knows how many operands each instruction requires, but it must be told what instruction to perform and where to send the results. The results may be sent to the Register File, to the Memory unit as an address or

as data, placed in the Jmp-Queue, or simply discarded. It is also possible to route the results to more than one of these destinations. In the case of a `MVPC` instruction, the subsequent PC address is passed unchanged through the ID and EX units and routed to the Register File.

There are sixteen possible opcodes in the NSR instruction set, so we can indicate the instruction to the EX unit by dedicating one bit of the 16-bit ID-EX communication path to each opcode class. There are two bits left over corresponding to the `JMP` and `BCND` instructions, which are never seen by the ID unit. These two bits are used to indicate whether the result is to be sent to the MEM unit (R1), or discarded (R0). This encoding is shown in Figure 10.

3.3 Execute Unit

The Execute unit is told what operation to perform by the ID unit, accepts the correct number of operands from either the Register File or—in the case of a `MVPC` instruction—from the ID unit, performs the operation, and routes the results to the correct places. The actual operations are standard mathematical and logical operations (see Figure 5). Zero, one, or two operands are provided by the Register File as instructed by the ID unit, but the EX unit doesn't know or care from which registers the operands come. There is only a single 16-bit path for source operands, so if two operands are required, they are presented in sequence. This decision was made due to the limited pin count of the FPGAs.

The ID unit also tells the EX unit how to route the result. The result may be presented to the RF unit to be written into a register, or it may be sent to the MEM unit as an address or as data. The result may also be discarded, as is common when the operand has a side-effect. This is often the case with the `SJMP` instruction, for example, which loads a value into the Jmp-Queue. With side-effects, it is possible that the results may be distributed to more than one destination. For example, the instruction `STA r1,r2,r3` would add the contents of registers R2 and R3, and put the result in both the Store Address Queue (as a side effect of the `STA`) and the Store Data Queue (because the destination register is R1) and would thereby initiate a memory write operation.

3.4 Register File

There are sixteen 16-bit registers available for use, numbered R0 through R15. Any of these registers may be used as source or destination for any instruction. There are three classes of registers, however. Registers R0, R14, and R15 are hardwired to the constant values of zero, one, and negative one, respectively. Writing to these registers has no effect on their contents. Registers R2 through R13 are normal general purpose registers. In the original design, R14 and R15 were also general purpose registers, but due to space restrictions they were hardwired to contant values for the NSR prototype. Register R1 is not a register at all, but is actually the access point for the memory FIFO queues.

The Register File unit is implemented in two FPGAs because of space and

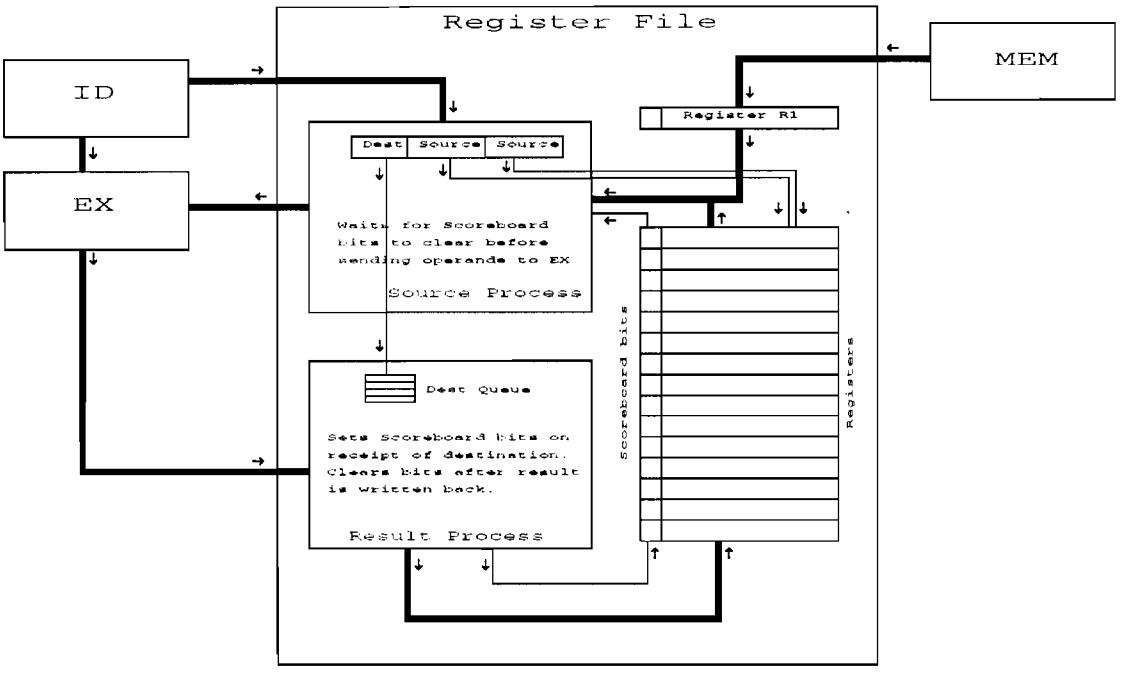

Figure 11: Register File Unit Processes

pin limitations. Both FPGAs are identical, and each contains an eight-bit slice of each of the registers. There is a mode pin which is used to designate one of the parts as the Master section to insure that both units will operate together. Logically, the Register File consists of two parallel processes. The "source" process provides source operands to the EX unit, while the "result" process writes the EX unit results back to the appropriate registers. The two processes coordinate their actions with the use of a FIFO queue containing the register number of the destination and with a scoreboard bit for each register. The scoreboard bit ensures that a register which has already been named as a destination is not used as a source until the results are available.

The source process begins operation when the usage information is received from the ID unit (see section 3.2). Three operations are performed in sequence to provide two sources and a destination. If the first source is a valid register address then a mux selects the appropriate register contents and scoreboard bit for examination. If the scoreboard bit is set, the output transition signal is delayed until the scoreboard bit is cleared. When the scoreboard bit is cleared, the register contents are provided to the EX unit. However, if the source register is R1, then the scoreboard bit is not checked, the source request is sent directly to the MEM unit, and the process waits until the MEM results are provided, then passes them on to the EX unit. When the source has been acknowledged by the EX unit, the same process is repeated for the

second source register. If either source register is not required, the requesting process is skipped. When all sources have been provided and acknowledged, the destination is dealt with.

If the destination is R0, no action is needed and the next register usage information is requested from the ID unit. Otherwise, the scoreboard bit for the destination register is set, unless it is already set in which case the process waits until it is cleared before setting it. The destination register number is enqueued in a FIFO to await future results from the EX unit, and the next register usage information is requested.

If an operand is requested from a register which is already being used for a destination (as indicated by the scoreboard bit), the operand is not provided until the results have been written back. FIFO queues buffering the ID usage information, the destination register, and the EX results help to hide varations in execution times.

The result process begins when a destination register address is placed in the destination FIFO. The process waits for results to be sent from the EX unit. When they arrive, the destination register number is popped off the destination FIFO, the results are written to the appropriate register and the corresponding scoreboard bit is cleared.

There is no direct correspondence between the number of source operands provided by the RF unit and the number of results written back. The ID unit coordinates the sourcing and disposition of data between the RF unit and the EX unit. In addition, results from the EX unit which are destined for R1 go directly to the MEM unit, without passing through the register file. Only when R1 is used as a source does data from memory pass through the RF unit, and even then the address must be sent from the EX unit directly to the MEM unit to initiate the read cycle. Figure 11 shows the structure of the Register File.

3.5 Memory Interface

The data memory (MEM) unit handles requests to read and write from the system memory. This unit contains the three queues needed to handle memory interfaces (see figure 6). Since there is only one Address Queue, all memory accesses are sequential, allowing decoupled memory access to take place without the possibility of read-after-write errors.

When a `LDA` or `STA` instruction is executed, an address is composed and then placed in the Address Queue (AQ), along with a bit indicating whether the address corresponds to a load or store operation. Load addresses which reach the front of the AQ initiate a load operation. The address is placed on the external bus, and the appropriate control signals are generated to fetch the memory contents. The MEM unit must convert the two-phase transitions that the NSR uses into the four-phase level-sensitive signals expected by the system SRAM. A simple external delay line is used to provide the request/acknowledge handshaking expected by the NSR. When the memory contents arrive on the data bus following the read signal, they are enqueued in the Load Data Queue, the FIFO queue which is used by the register file to satisfy read requests from

register R1.

Memory writes operate in a similar fashion. They are initiated when both a store address and a store data value reach the head of the Address Queue and the Store Data Queue, respectively. At that point, both the address and data are placed on the external busses, and the correct signals are generated to produce a write cycle on the system memory. Although the programmer uses register R1 as a destination in order to write to memory, the data to be written never actually passes through the register file, but instead comes directly from the EX unit. Recognizing that R1 is a destination and producing the correct routing signals is one of the functions of the ID unit.

The MEM unit is implemented in two FPGAs, consisting of both a master and a slave chip. These chips are very similar, but the master coordinates the actions of the two, and handles the interface to the memory arbiter. Each chip contains eight bits of the 16-bit data queues, with the master chip also containing the Load/Store bit of the Address Queue.

3.6 Memory Arbitration

The IF unit and the MEM unit must take turns accessing the single system memory. To accomplish this, a round-robin arbiter is built into parts of both units. This arbiter passes a token between the two units, and only the unit which has the token is allowed to access the memory. Usually, the IF stage performs most of the memory accesses. Between each instruction fetch, the token is passed to the MEM unit. If there are no pending data reads or writes, the token is simply returned to the IF unit. If the MEM unit wishes to access memory, it keeps the token until one read or write operation has been completed. The design of the arbiter is shown in Figure 12.

The arbiter starts when a single transition occurs on the `INIT` line thereby inserting a token into the loop. The token circulates until one of the processes requests it with a transition on its `REQ-MEM-PROC` line. This causes the token to be diverted until the memory process is finished. Notice that because of the way the Q-select module samples its probe input, the token may pass by twice before the request is recognized. The first pass samples the probe input, and the token is diverted on the second pass. This arbiter design is not very efficient, but it uses only two pins on each chip, and it is fair.

4 Performance and Evaluation

The Protozone$_{tm}$ prototype board produced by Standford University contains memory, logic, and connections to communicate with a standard PC clone. The NSR occupies the development space on this board, using wirewrapping sockets for the FPGAs. The NSR prototype is shown in Figure 13.

Debugging the NSR was remarkably simple. Each chip was thoroughly simulated with unit delays as part of the design process. Once the functionality was correct, the design was placed and routed on the Actel parts, the more realistic delays back-annotated, and the simulations were repeated. The main

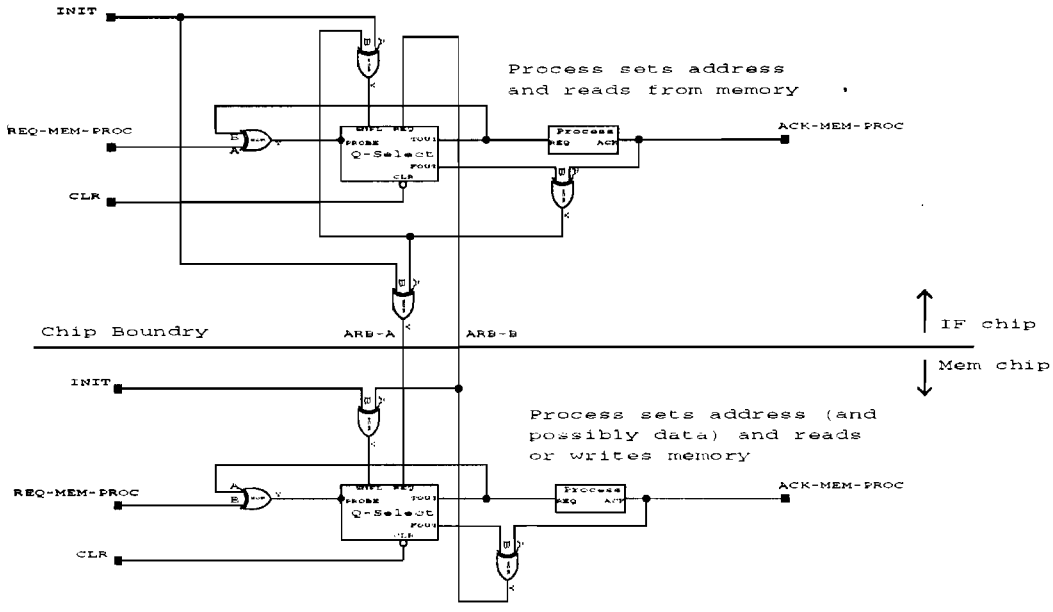

Figure 12: Round-Robin Arbiter

problem with this method was that the Actel tools do not provide a way to specify the bundling constraint in the routing process, and so delays had to be added by hand in some cases by inserting buffers in the control paths. To further complicate matters, the delay changes each time another place-and-route is performed because of the relatively high resistance of the Actel antifuses. It would be very useful if routing tools were available which allowed the designer to specify that the delay of a request line be greater than the delay of the associated data lines.

Some circuitry was added to each chip for debugging purposes. Each unit has an extra gate attached to the main incoming request signal which can indefinitely delay the request transitions from the connected units. These gates are controlled by switches installed on the protoboard along with the rest of the NSR, allowing instructions to be stepped through the pipeline one by one. In addition, several LEDs are used to monitor the state of the request and acknowledge lines between chips, so that when the NSR is deadlocked, there is some indication as to which unit is causing the problem. Since the request and acknowledge transitions occur in pairs, starting from a zero level, any mismatch of requests and acknowledges can be easily detected. Additional lights are used to monitor the state of the memory arbiter.

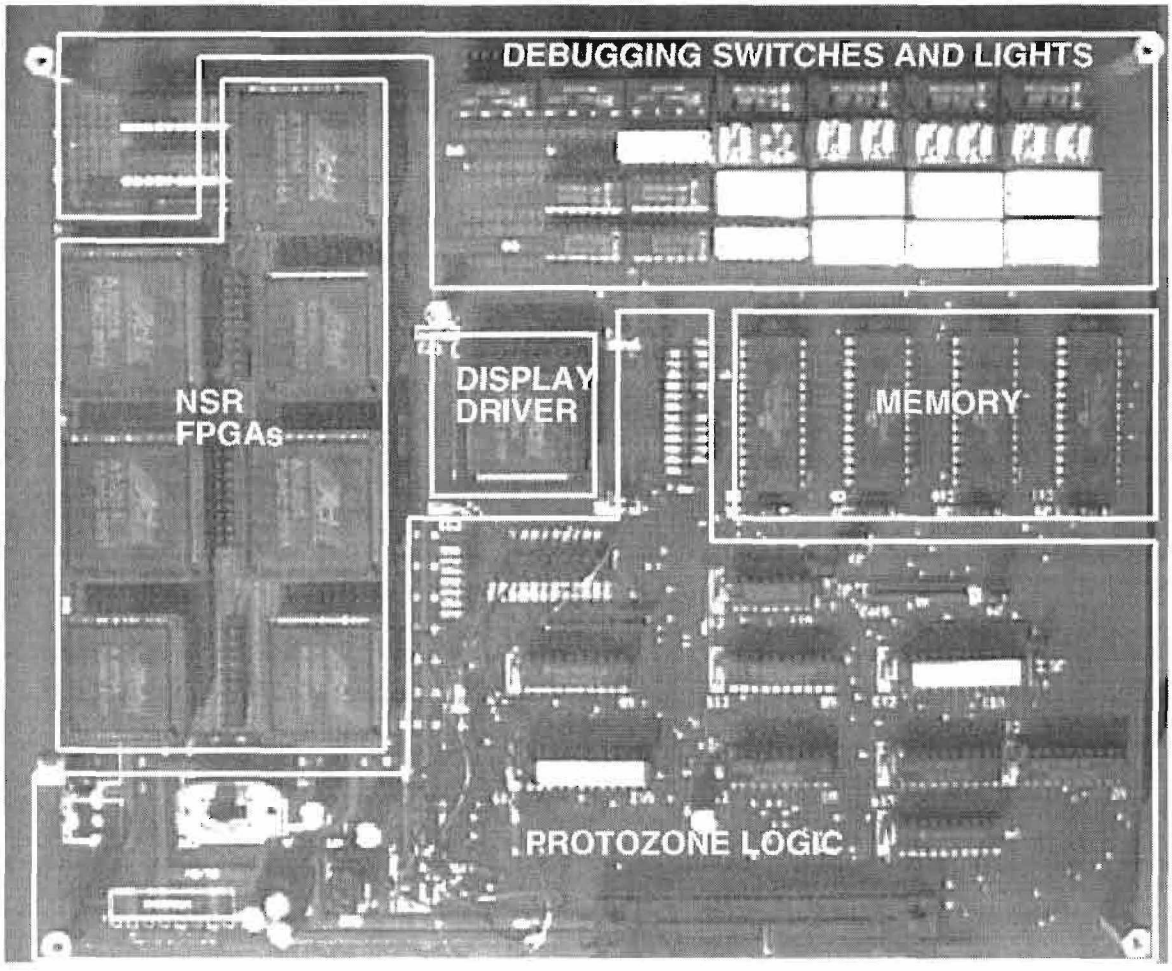

Figure 13: The NSR Prototype

Another useful debugging aid is a bus monitor. Since the NSR can be stopped temporarily by holding up the request signals between chips, the data placed on the inter-chip buses can be examined. A driver and encoder for a set of four seven-segment LEDs was built into an FPGA, and a ribbon cable and plug was used to monitor the bus contents. This is very useful in determining whether the correct data was being transmitted between units.

These switches and lights are useful in getting the NSR to communicate among its component parts. To insure correct operation, traces of the inter-chip buses can be made using a standard logic analyzer. Although the NSR uses transitions instead of levels to indicate when the data is valid, by using two channels of the logic analyzer on the same bus and triggering one on the rising edge of the request line and the other on the falling edge, a complete trace of the bus activity can be obtained.

4.1 Software tools

Communicating with the NSR is fairly simple. The protoboard memory is mapped directly into unused memory space on the PC, as byte addresses D000:0000 to D000:FFFF. This addresses only 64 Kbytes, but another 64 Kbytes is available by changing a bit on a specific I/O port. The NSR sees the two bytes with the same PC address as a single 16-bit word. Either the NSR or the PC can access the protoboard memory, but not both. Access is moderated by a toggle switch. Several simple programs have been written to transfer data between the NSR memory and the PC. A load utility takes a simple text file describing the address and contents of the NSR memory and loads the memory with the appropriate values. An unload utility reverses the process.

4.1.1 Assembler

A simple assembler has been written to convert the NSR assembly language instructions into data which can be loaded into the NSR memory and executed. The assembler allows for labels, symbols, data constants and relative offsets, but does not produce object files which can be linked with others. The output of the assembler is a text file containing the address and data of each affected NSR memory location. The NSR always begins execution at location 0x0000.

A nice ability of the assembler is to add some additional opcodes for instructions not implemented directly by the NSR. The NSR only tests for GT, GE, EQ, and NE conditions. The LE and LT conditions are implemented by the assembler as GE and GT tests, with the operands reversed. For example, the instruction `SLE r2,r3` would be implemented as `SGE r3,r2` instead. In the same way, the `NOT r2,r3` instruction is assembled as `XNOR r2,r0,r3`.

One of the first programs to be run on the NSR was a simple test to generate Fibbonacci numbers. The source is shown in Figure 14, which provides a good example of the assembly language for the NSR.

```
;       this is a test to generate Fibbonacci numbers

text    .EQU    0x0100
data    .EQU    0x0200          ; write results here

limit   .EQU    22              ; only do 22 of em

start:  seq     r0,r0           ; jump to main
        bcnd    main

        .ORG    text
main:   mvih    r2,hi8(data)
        mvil    r3,lo8(data)
        or      r2,r2,r3        ; r2 points to output place

        mvil    r3,0x1          ; we'll repeat r5 = r4 + r3
        mvil    r4,0x1          ; and shift 'em down.
        mvil    r10,lo8(limit)  ; just do a few
        xor     r8,r0,r0        ; init count register

        sta     r0,r2,r0        ; gonna write one
        or      r1,r3,r0        ; write r3
        add     r2,r2,r14       ; increment r2

        sta     r0,r2,r0        ; gonna write one
        or      r1,r4,r0        ; write r4
        add     r2,r2,r14       ; increment r2

loop:   add     r5,r4,r3        ; r5 = r4 + r3
        sta     r0,r2,r0        ; gonna write one
        or      r1,r5,r0        ; write r5
        add     r2,r2,r14       ; increment r2

        or      r3,r4,r0        ; copy r4 to r3
        or      r4,r5,r0        ; copy r5 to r4

        add     r8,r8,r14       ; r8++
        sle     r8,r10          ; branch if r8 < r10
        bcnd    loop            ; take branch

        jmp                     ; die
```

Figure 14: Fibbonacci Program Source

4.1.2 Simulator

To aid in the development of test programs for the NSR while the prototype was still being developed, a simulator was written. In addition to speeding program development, the simulator was also useful in suggesting changes to the instruction set. At first, the `MVPC` instruction did not add anything to the program counter value, requiring the programmer to dedicate a register to fixing up the PC value before using it. After writing a few programs and running them on the simulator, it was realized that this was awkward, and the modification to the instruction set was made.

The simulator is based on an implementation of the C-Threads library routines [6], and has been built on both Sun SPARC and Hewlett-Packard workstations. The simulator consists of C-Threads libraries which are machine-dependent, plus the actual NSR simulation code written in C. Separate threads are used for each functional unit of the NSR, with two threads required for the Register File. The functional units of the simulator communicate over pipelines implemented with semaphores.

Although the simulator is accurate in that it produces the same results as the NSR when running the same programs, it does not attempt to model the performance of the NSR. Some discussion is underway to determine whether modifying the simulator to do so would be worthwhile.

4.2 Speed

The speed of the NSR varies depending on the program it is running, but the best performance to date has fallen between 1.10 and 1.34 MIPS. This is relatively slow, but is not surprising since the Actel devices are relatively slow and all design decisions were made to minimize the number of gates and/or pins needed from the FPGAs. Speed optimizations were not considered for the prototype.

Performance was measured by placing a series of exactly 1000 instructions in a large loop and then executing that loop 65536 times, after which the NSR was deadlocked. Execution time was measured by placing an oscilloscope probe on the request signal between the IF and ID, and measuring the duration of the active signal with a stopwatch. Typical execution times were on the order of one minute, with repeatability within half a second, for an accuracy of within 1 percent. A variety of test programs have been run with the results listed in Figure 16.

4.3 Power Consumption

To measure power consumption, the power supply traces on the prototype printed cicuit board were cut and rerouted to isolate the NSR FPGAs from the rest of the protoboard. An ammeter was place in series with the FPGA power supply, and current drain was measured while various programs were running, and also while the NSR was idle or deadlocked (Figure 15). All measurements were made with the LED bus display and driver removed, although it made

NSR State	Current Drain
CLR = 0, RESET = 0	45.3 mA
Deadlocked on JMP, Arb ON	52.7 mA
Deadlocked on JMP, Arb OFF	45.3 mA
Running `BCND` loop	53.1 mA
Running `SJMP` loop	40.8 mA

Figure 15: NSR Current Drain

very little difference.

As expected, the current drain was higher when data memory was accessed, since the MEM unit draws less current when idle. Correspondingly, programs which only branched and did not use any registers used less current. A heavier drain was also noted when the operands of instructions had a larger number of ones in their binary representation.

The standby current for a typical Actel Act-1 series FPGA should be around 3 mA with a maximum of 10 mA [1], if all outputs are unloaded. The measured standby current for the seven FPGAs comprising the NSR was 45.3 mA. Surprisingly, the current drain was actually less while running some particular programs than when deadlocked. We currently have no explanation for this behavior, except to note that the perversity of the universe tends toward a maximum.

5 Conclusions and thoughts on Fred

Plans are being made for the development of a 32-bit self-timed processor which would incorporate several architectural changes and improvements when compared with the NSR. For obscure reasons[2] this processor will be called Fred. With Fred, we hope to develop a processor capable of acting as the main component of a standalone computer system. If time and resources permit, we would like to be able to build a Unix platform with it.

Accordingly, there must be several architectural changes. We plan to provide for 8-, 16-, and 32-bit memory accesses, I/O ports, hardware and software interrupts, and increased parallelism with additional arithmetic or logical units. It might also be desirable to separate loads from stores, allowing out-of-order memory access if needed. Much time will need to be devoted to speed issues.

From a programming standpoint, there are many improvements to the instruction set which would be desirable in a 32-bit version of the NSR. These include adding a carry bit for multiple precision arithmetic, providing for immediate operands in several instructions, allowing for additional classes of instructions, and providing software exceptions. The addition of a protected mode of operation for system security would be useful also.

Of course, we may do something completely different.

[2] Apparently, Erik just likes the name.

Program	Contents	Seconds	MIPS	milliAmps
ADD0.S	add r9,r0,r0	52	1.27	63.0
ADD1.S	add r9,r14,r15	52	1.27	93.3

ADD2.S	add r9,r0,r0
	add r10,r0,r0
	add r11,r0,r0
	add r12,r0,r0
		52	1.27	65.3

ADD3.S	add r0,r0,r0	52	1.27	57.4
ADD4.S	add r0,r0,r15	52	1.27	95.7
ADD5.S	add r9,r0,r15	52	1.27	102.2
OR0.S	or r0,r0,r0	51	1.29	52.5
OR1.S	or r0,r0,r15	51	1.29	68.3
SEQ0.S	seq r0,r0 bcnd +1	56	1.18	56.9
SEQ1.S	seq r0,r15 bcnd +1	56	1.18	45.0
JMP0.S r9 = PC	sjmp r9,r9,r8 jmp	49	1.34	62.4
MVPCJMP.S	mvpc r9,+3 sjmp r0,r0,r9 jmp	57	1.16	73.1
LDA0.S	lda r0,r0,r0 or r0,r0,r1	59	1.12	66.4
STA1.S	sta r1,r0,r15	60	1.10	105.3
STA2.S	sta r0,r0,r15 or r1,r0,r15	55	1.20	106.4
LDASTA2.S	lda r0,r0,r15 sta r1,r0,r1	58	1.14	117.0
LDASTA3.S	lda r0,r0,r15 or r9,r0,r1 sta r1,r0,r9	55	1.20	108.7

MEM1.S
r9 = −1, r8 = 1
	lda r10,r9,r8
	lda r10,r9,r8
	sta r11,r9,r8
	add r1,r1,r1
		54	1.22	98.2

MEM1A.S	lda r10,r14,r15
	lda r10,r14,15
	sta r11,r14,15
	add r1,r1,r1
		55	1.20	100.0

MEM0.S
r9 = r8 = 0
	lda r10,r9,r8
	lda r10,r9,r8
	sta r11,r9,r8
	add r1,r1,r1
		54	1.22	75.0

MEM0A.S	lda r10,r0,r0
	lda r10,r0,r0
	sta r11,r0,r0
	add r1,r1,r1
		54	1.22	71.7

Figure 16: Performance and Current Drain

References

1. Actel Corporation. *ACT Family Field Programmable Gate Array Databook*, March 1991.

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

3. 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.

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

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

6. Eric C. Cooper and Richard P. Draves. C threads. Department of Computer Science, Carnegie Mellon University, September 1990.

7. 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.

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

9. Ivan Sutherland. Micropipelines. *CACM*, 32(6), 1989.

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