The N.I.G.E. Machine: an FPGA based micro-computer system for prototyping experimental scientific hardware

Andrew Read

May 2012

anding_eunding@yahoo.com

Abstract

This paper describes the N.I.G.E. Machine, a user-expandable micro-computer system that runs on an FPGA development board and is designed specifically for the rapid prototyping of experimental scientific hardware or other devices. The key components of the system include a stack-based softcore CPU optimized for embedded control, a FORTH software environment, and a flexible digital logic layer that interfaces the micro-computer components with the external environment. The system has been demonstrated on a Digilent Nexys 2 development board and in an example scientific experiment involving a light source and sensor.

1 Introduction

1.1 Overall concept

The N.I.G.E. Machine's primary intended application is as an electronic control and measurement unit for experimental scientific apparatus. The concept is to combine the merits of a traditional computer-based control system with the flexibility of Field Programmable Gate Arrays (FPGAs), and a rapid prototyping environment. This is done using a FORTH based, harmonized hardware-software system and a commodity FPGA development board.

FPGAs are integrated circuits with reconfigurable internal logic components and interconnects that can be repeatedly reprogrammed with fresh logic designs at the time of use. The functional capability of FPGAs is broadly equivalent to traditional integrated circuits, although operating parameters differ. Logic designs are written in a hardware description language such as VHDL or Verilog and then downloaded to the FPGA as a binary file after being synthesized by the development tools of the relevant FPGA manufacturer. Typically new logic designs are tested on special circuit boards (development boards) that host an FPGA alongside commonly used peripheral components such as external memory, switches, indicators, and connectors.

The N.I.G.E. Machine is a complete, user-expandable micro-computer system with extensive input/output (I/O) capabilities, hosted on a low-cost FPGA development board. It comprises (a) a general purpose, stack-based, 32-bit softcore CPU that has a number of optimizations for embedded control such as deterministic execution and rapid interrupt response time, (b) FORTH system software, and (c) a set of digital logic interface components including both peripherals for development use (keyboard, video) and also user-expandable peripherals for application specific interface logic (interrupt controller, hardware registers, and various I/O ports).

The softcore CPU and the other digital logic components are coded in VHDL[1] and the FORTH environment is coded in the assembly language of the softcore CPU. The N.I.G.E

Machine has been implemented and tested on a Digilent Nexys 2 development board with a Xilinx Spartan-3E XC3S1200E FPGA. With a keyboard and video monitor connected to the board, the system is a fully operational, stand-alone native FORTH environment from power on. A short video has been made to demonstrate[2].

1.2 Engineering approach

Typically there will be three layers of engineering in an application for prototyping an experimental scientific apparatus:

The physical layer: The scientific hardware, with actuators and sensors, driven by some electronic circuitry. The external electronic circuitry is connected to the N.I.G.E. Machine through the numerous expansion connectors available on the host FPGA development board, routing to free I/O pins on the FPGA.

The digital logic interface layer: Factory or custom (written in VHDL) digital logic interface peripherals directly interface with the external electronic circuitry and provide I/O functionality for the micro-computer system. The use of application-specific, custom digital logic interface peripherals permits greater scope for high-speed pre-processing of external signals and for processing multiple external signals in parallel than is possible with typical fixed-logic I/O ports. In addition, complete flexibility is permitted in the interface specification and design. The digital logic interface components are connected into the micro-computer architecture largely through hardware registers and external interrupts.

The micro-computer layer: The micro-computer system, built on the softcore CPU and the FORTH system software, is the platform both for the interactive prototyping of the external hardware and the custom digital logic interface, and also for running the final operating software.

1.3 Comparison to alternatives

Other platforms have existed for some time that also combine elements of this engineering approach, for example:

- The use of a digital logic layer interfacing with the scientific apparatus parallels that of the CMS and ATLAS detectors at CERN. In the case of these detectors, the volume of measurement data and the speed with which it is generated necessitates a digital logic layer ahead of the computer layer to identify potentially interesting events and filter out the remainder before further processing. The CERN detectors are very high performance designs and extremely expensive.
- Numerous commodity, easy-to-use microcontrollers, such as the Atmel ATMEGA or Parallax Propeller provide a CPU with direct access to digital I/O ports alongside a selection of fixed-function hardware resources such as counters and timers. FORTH is sometimes implemented on such microcontroller platforms.
- Standard commercial or open source FPGA softcores are available that can be configured directly by the FPGA development tools, for example the Xilinx MicroBlaze[15].
- A number of small softcores have been designed specifically to execute FORTH[3, 4, 5, 6, 7, 8]. Several aspects of the J1[3] have directly inspired this project.

The N.I.G.E. Machine does not intend to compete head-on with any one of these alternatives, but rather offer something novel in the way that it brings different aspects together to create a flexible platform particularly focused on enabling rapid, small-scale, scientific research and development. In particular:

- The N.I.G.E. Machine works with FPGA development boards that are easy-to-use and affordable, so they are within the range of small labs and individuals.
- The N.I.GE. Machine is a fully integrated micro-computer system with video and keyboard facilities (rather than a purely embedded system).
- No complex or slow-to-use tool chain is required for software development. FORTH is instantly available at power-on.
- Rather than than relying on generic, fixed-function hardware resources, the N.I.G.E. Machine's digital logic interface layer can be custom built and optimized for each new application.
- The N.I.G.E. Machine's stack-based softcore is a full-featured, general purpose CPU that includes functionality such as interrupts, flexible memory access, and debugging facilities.

1.4 Scope of this paper

The primary aim of this paper is to describe the new softcore CPU and this is the main focus of section 2, "Methods". The paper also aims to illustrate the novel hardware development platform offered to users of the N.I.G.E. Machine and the use of FORTH for the development and testing of experimental apparatus. These are the main focus of the example application described in section 4, "Application discussion".

2 Methods

2.1 Key design objectives

In order to meet the goals outlines above, key design objectives for the softcore CPU were set out under the headings of (1) platform, (2) real-time control, and (3) CPU performance. These objectives and the strategies devised to meet them are as follows.

2.1.1 Platform objectives

Rapid prototyping To shorten application and software development time for users it was decided to (a) not require the use of an external tool-chain for programming and (b) have an interpreter available at power on. An interpreter allows the user to experiment directly with the electronic apparatus and also test small routines for bottom-up software development. BASIC was considered, but FORTH was chosen because it provides both an interpreter and also a compiler which can produce executables that are almost as fast as directly assembled machine code. For a new computer system, FORTH also has the advantage that it can be implemented quite easily in assembly language.

Custom digital logic interface layer The scope for creating custom digital logic interfaces in VHDL for each new application differentiates the N.I.G.E. Machine from a conventional microcontroller platform with resident FORTH, and opens up new design possibilities. The system can offer this firstly because it is built with soft-logic in an FPGA and secondly because the key micro-controller interface components (principally the memory-mapped hardware register module and the prioritized hardware interrupt controller) have been made user-expandable.

Stand-alone usage Alongside the CPU softcore and FORTH, a full set of peripheral modules (e.g. keyboard, video, other I/O) have been developed to create a complete micro-computer system for stand-alone use.

2.1.2 Real-time control objectives

Fast interrupt response time Fast interrupt response time facilitates the high frequency, low-latency processing of external signals[9]. One of the key handicaps to interrupt processing is the need to save the state of a large register set. Sympathetic with the choice of FORTH for the system software, the softcore CPU is stack based and so no save/restore of registers is required for interrupt (or subroutine) processing[11]. The N.I.G.E. Machine's typical interrupt response time is only 2 cycles to branch to the interrupt vector table followed by 3 cycles to branch through the vector to the interrupt routine itself.

Deterministic execution Deterministic execution[9] is the certainty that a given set of instructions will execute in a given number of CPU clock cycles regardless of state. Conversely, without deterministic execution, jitter is the deviation of an expected periodic signal from true periodicity. Avoiding jitter in electronic interfaces is essential for precise control and measurement. The instruction set and CPU control unit have been designed so that all instructions execute in a fixed number of cycles, including conditional branches and static RAM (SRAM) memory access. The CPU's execution pipeline has been designed so that there are no conflict states that could result in missed cycles.

Fast branch performance Fast, deterministic, branch performance is important to optimize response times in an embedded application[9]. On the N.I.G.E. Machine conditional and unconditional branches (BEQ and BRA) are designed such that they always execute in only 3 clock cycles (2 cycles for decode, 1 cycle for memory).

Maximum code density The fastest memory resources available to a softcore CPU are FPGA SRAM blocks. These also have the advantage over external memory of deterministic access (i.e. guaranteed single clock cycle read/write). However FPGA SRAM resources are typically limited to a several tens or hundreds of kilobytes. To maximize the use of SRAM as program memory, code density needs to be as high as possible[9]. On the N.I.G.E Machine almost all instructions are encoded in a single byte. This is achieved by using microcode as opposed to a hardwired decoder in the CPU.

2.1.3 CPU performance objectives

High instruction throughput High instruction throughput translates directly into higher processing performance. Without super-scalar features or parallel cores, the best achievable goal is throughput of one cycle per instruction. The N.I.G.E. Machine's CPU design features a three-stage execution pipeline that delivers single cycle throughput for most instructions.

Flexible memory access The N.I.G.E. Machine is a 32-bit (longword) CPU and all system memory is byte addressable. To optimize the speed and flexibility of memory access: (1) separate CPU instructions have been created to read and write memory in byte, word, and longword format, (2) the CPU is designed with three separate memory buses (one for SRAM access and two for byte and word access to the external pseudo-static dynamic RAM (PSDRAM) that is part of the Nexys 2 development board), and (3) even address alignment is not required when accessing word or longword data in SRAM system memory.

Fast subroutine performance As an optimization for the execution of FORTH, the instruction set includes a compound RTS (return from subroutine) instruction that can be overlaid on top of most single byte instructions, saving one clock cycle on each subroutine return. This follows the design of the J1 processor[3].

2.2 Limitations

Hardware and design tradeoffs also resulted in some limitations of the N.I.G.E. Machine as set out below. None of the limitations are fundamental to the design and hopefully they will be addressed in future developments.

Program memory space The softcore CPU is currently only able to execute instruction code located within FPGA SRAM memory. The external PSDRAM on the Nexys 2 development board memory cannot be used as a program memory store unless its contents are first copied to SRAM for execution.

Narrow instruction fetch Instructions are fetched from SRAM one byte at a time. This means that instructions requiring several bytes (mainly the load literal instructions) necessarily take several cycles to execute.

Lack of floating point The current implementation of FORTH does not include floating point software routines nor does the digital logic design include any floating point functionality in FPGA hardware.

Blocking interrupts The interrupt scheduler provides interrupt prioritization but once an interrupt is in progress it will block all other interrupts, even those of higher priority.

Lack of double precision (64-bit) arithmetic Some FORTH words in the ANSI core set require double precision division. However the current implementation of FORTH does not include double precision arithmetic software routines. Additionally the hardware dividers used in the CPU are limited to 32-bit operands.

2.3 Implementation of the CPU and other digital logic

2.3.1 CPU instruction set

The softcore CPU has 63 instructions as follows:

Stack manipulation 15 instructions: NOP (no operation), the FORTH words DROP, DUP, ?DUP, SWAP, OVER, NIP, ROT, >R, R@, R>, plus four words for loading or saving the parameter and return stack pointers

Math operations 12 instructions: +, -, NEGATE, 1+, 1-, arithmetic shift left and right, signed and unsigned multiply, add and subtract with carry, and signed and unsigned divide

Comparison operations 11 instructions: the bitwise equality tests = and <>, signed comparisons <, >, unsigned comparisons U<, U>, comparisons with zero: 0=, 0<>, 0<, 0>, and FALSE, which returns zero

Bitwise operations 7 instructions: the Boolean operations AND, OR, INVERT, XOR, logical shift left and right, and byte and word sign extension to 32 bits

Memory operations 6 instructions: FETCH and STORE of byte, word, or longword values

Load literal operations 3 instructions: LOAD longword, word or byte values from within the program code

Flow control 6 instructions: JMP (jump to the address on the parameter stack), BSR and JSR (branch/jump to subroutine), RTS (return from subroutine), and BEQ, BRA (conditional and unconditional flow control)

Exception handling 3 instructions: TRAP (software trap vector) RTS_TRAP (a single-step), and RTI (return from interrupt)

2.3.2 Memory data format

Data is stored in memory in big-endian format. That is, for multi-byte data the highest value byte is stored at the lowest numbered memory address. By way of context, Motorola 68k processors also use big-endian format while Intel processors use little-endian format. Either format could have been implemented in the softcore CPU but the big-endian format was found to be more suitable for the design of the shift registers that fetch data from memory, as well as being the author's preference because of its Motorola 68k heritage.

2.3.3 Instruction set encoding

The default instruction size is a single byte, encoded as follows (figure 1): bit 7 identifies whether the instruction is a branch or an ordinary instruction. If the instruction is a branch then bit 6 specifies if the branch is conditional or unconditional. If the instruction is ordinary (not a branch) then bit 6 specifies whether a return from subroutine is to be taken along with the execution of the instruction (this is the compound RTS instruction). For ordinary instructions, bits $5 - 0$ (figure 2) are read as an integer that identifies the instruction in question (i.e. the "opcode"). For branch instructions bits $5 - 0$ of the instruction are read as the high part (bits $13 - 8$) of the branch address, with a following byte holding the low part (bits $7 - 0$) of the branch address. Where literal data is required as part of an instruction it follows in the succeeding bytes (figure 3).

Bit 7	Bit 6	Interpretation
1	1	Unconditional branch (BRA)
1	0	Conditional branch (BEQ)
0	1	Ordinary instruction plus return from subroutine (RTS)
0	0	Ordinary instruction

Figure 1: Bits 7 and 6 of an instruction specifies its type.

Bit 7	Bit 6	Bits 5 - 0
1	x	High part of branch address
0	x	Opcode

Figure 2: Bits 5-0 of an instruction either specify the high part of the branch address or the opcode.

	Byte 1 (bits 5-0)	Byte 2	Byte 3	Byte 4	Byte 5
Branch	14 bit branch address		-	-	-
Load.L	Opcode	32-bit literal			
Load.W	Opcode	16-bit literal		-	-
Load.B	Opcode	8-bit literal	-	-	-

Figure 3: Multi-byte instructions specify literal data (big-endian format).

2.3.4 Overall CPU architecture

The CPU comprises a datapath and a control unit[10].

The datapath holds the registers and computation components associated with the data held in the parameter and return stacks. The datapath is a passive entity in the sense that it does not contain any control logic or state information of its own. Rather it includes a network of multiplexers and other switches that route data between registers and through computation components in various configurations. The behavior of the datapath at any moment is entirely governed by a set of external control signals feeding to it from the control unit.

The control unit is built around a sophisticated finite state machine (FSM) that is responsible for reading program instructions from system memory, decoding those instructions, and then setting the control signals to the datapath as appropriate for the execution of each. The control unit is also responsible for adjusting the program counter (PC) so that program instructions are read from memory in the appropriate order taking into account program jumps and branches, dealing with interrupts and other exceptions, and supporting data transfers between system memory and the data path.

2.3.5 Execution pipeline

The architecture of the CPU is built around a three stage execution pipeline. The pipeline stages are as follows:

1. "FETCH OPCODE". The next instruction is read from SRAM at the current address of the program counter. The control unit identifies the instruction type and extracts the opcode.

2. "DECODE AND EXECUTE" The current opcode is decoded via microcode and the appropriate control signals are sent to the datapath. The datapath configures according to the control signals and the result of the computation becomes available in combinatorial logic.

3. "SAVE" The parameter and return stack registers and memory (i.e. the synchronous logic) are updated with the result of the computation performed by DECODE AND EXECUTE in the previous stage.

The operation of the pipeline is illustrated with a worked example in figure 4. In this example, as at CPU clock cycle #0 the program counter is pointing to memory address zero. The execution pipeline proceeds thus:

Component / clock cycle	Cycle #0	Cycle #1	Cycle #2	Cycle #3
Program counter	0			
Instruction byte		38		
Opcode		38		
Microcode			1210	
TOS_n (combinatorial logic)			0	
TOS (synchronous logic register)				0

Figure 4: Illustration of the execution pipeline for the CPU instruction FALSE, which places zero on the parameter stack

1. "FETCH OPCODE". On the rising edge of clock cycle #1 SRAM system memory reads the data byte at the memory address pointed to by the program counter and makes it available to the control unit where it is known as the instruction byte. In this example the instruction byte has a value of 38 (corresponding to the instruction

"FALSE" which places zero on the top of the parameter stack). During the same clock cycle combinatorial logic within the control unit identifies (based on bit 7 of the instruction byte) that this is an ordinary instruction and extracts the opcode from the instruction byte. In this case the opcode also has the value 38.

2. "DECODE AND COMPUTE". On the rising edge of clock cycle #2 SRAM microcode memory within the control unit takes the opcode as a lookup address and returns the corresponding microcode value. During the same clock cycle the combinatorial logic in the datapath is configured according to the microcode value through its control signals and the value of the computation becomes available at the output of the multiplexer TOS_n (figure 5). In this case the microcode value is 1210 (corresponding to a particular configuration of control lines that will cause the datapath to push a zero onto the top of the parameter stack).

3. "SAVE". On the rising edge of clock cycle #3, the value presented by the multiplexer TOS_n (i.e. the result of the computation in the previous pipeline stage) is written into the synchronous logic register TOS (figure 5). At the same time the current value of TOS is written into NOS, and the current value of NOS is pushed into the SRAM block that holds the remainder of the parameter stack.

The CPU has a throughput of one instruction per clock cycle for most instructions since each pipeline stage executes in a single cycle, thus on every clock cycle another instruction is completing execution. The CPU has a latency of three cycles since it takes three pipeline stages to execute each instruction in full. As with any pipeline design there is a tradeoff between the number of pipeline stages and the maximum feasible CPU clock frequency. Longer pipelines have less logic to execute at each stage, thus requiring less time and permitting a higher clock frequency, but at the expense of higher latency and the introduction of issues such as conflicts between instructions at the beginning and end of the pipeline. The N.I.G.E. Machine's pipeline was designed to ensure deterministic execute at all times (i.e. no conflict states, failed branch predictions, etc.) at the same time as a maximizing clock frequency subject to that constraint. In particular the design mixes SRAM access (which tend to be very fast) with combinatorial logic functions (which tend to be slower) in stages 1 and 2 to balance the overall load throughout the pipeline.

2.3.6 The CPU datapath: parameter stack

Figure 5 illustrates the parameter stack datapath.

The top-of-stack (TOS) and next-on-stack (NOS) storage locations are 32 bit hardware registers while the remainder of the parameter stack is implemented with a dedicated 2KB SRAM block. This SRAM block is dual ported and the second port is mapped to the CPU address space. (This is useful for implementing FORTH instructions such as PICK.) The datapath is directed from the control unit via a 14 bit wide signal generated from control unit microcode that drive a set of multiplexers in the datapath and determine the data flow. The main multiplexers controlling the parameter stack are as follows:

- The multiplexer TOSn selects the value for the update of the TOS register from one of eight computation units: addition/subtraction, logic operations, multipurpose, comparison, multiply, unsigned multiply, divide and unsigned divide.

- The multiplexer NOSn selects the value for update of the NOS register from: TOS, NOS (i.e. itself, no-update), the item below NOS in the parameter stack RAM, or an arithmetic value from one of the computation units.

- The multiplexer PSPn is responsible for updating the parameter stack (PS) pointer, which is a 9 bit address signal spanning 512 * 32 bit cells in 2KB SRAM. The PS pointer can be incremented (the stack grows by one item), decremented (the stack

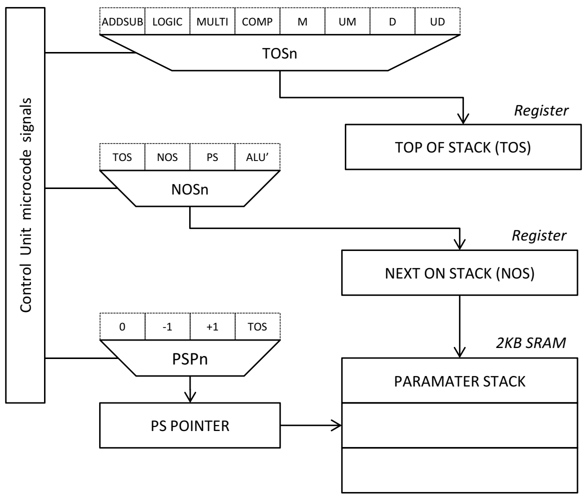

Figure 5: The parameter stack datapath illustrates the use of an SRAM block in combination with hardware registers.

shrinks by one item), held constant or updated with the current TOS value. When the PS pointer is incremented, the current NOS item is written from the 32 bit register to SRAM. The opposite dataflow occurs when the PS pointer is decremented.

The eight multiplexed computation units attached to TOSn essentially form the arithmetic logic unit (ALU) of the CPU. Each computation unit is directed by signals from the control unit microcode according to the functionality required by each operation. Some of the computation functionality is provided by Xilinx CORE modules that may leverage special purpose circuitry available within the FPGA such as hardware multipliers and carry logic structures. The computation units are summarized as follows:

- ADDSUB, an adder/subtractor implemented using a XILINX CORE template that leverages special purpose carry structures on the FPGA. There is a carry flag within the ADDSUB unit that is not directly accessible to the CPU but which gives the N.I.G.E. machine the capability to perform double precision addition and subtraction. The carry flag is only changed by one of the 7 addition or subtraction operations above and remains unchanged during the execution of all other instructions. (Interrupts should be temporarily suspended via the interrupt mask hardware register before performing double precision addition and subtraction to ensure that the hidden carry flag is not changed inadvertently).

- LOGIC, bitwise logic computations implemented in VHDL.

- MULTI, a general purpose multiplexer implemented in VHDL.

- COMP, a comparison unit implemented using a XILINX CORE template with supporting logic in VHDL.

- M and UM, signed and unsigned multiply implemented using on-chip FPGA pipelined multipliers with 32 bit operands and a 64 bit result. The operations complete in 5 clock cycles.

- D and UD, signed and unsigned divide implemented in logic fabric using a XILINX CORE template with 32 bit operands, a 32 bit quotient and a 32 bit remainder. The operations complete in approximately 40 clock cycles.

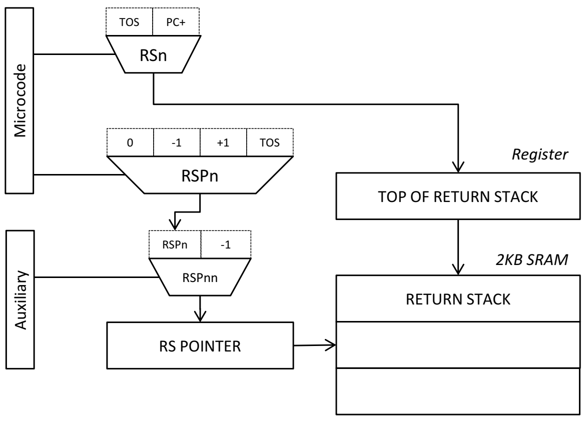

Figure 6: The return stack datapath illustrates the use of an SRAM block in combination with hardware registers.

2.3.7 The CPU datapath: return stack

Figure 6 illustrates the return stack datapath.

The Top of Return Stack (TORS) value is implemented as a 32 bit hardware register and the remainder of the return stack in a dedicated, dual ported 2KB SRAM block the second port of which is mapped to the CPU address space.

- The multiplexer RSn updates TORS with either the value from the top of the parameter stack (TOS), or the program counter of the next instruction following the instruction that is currently being executed. (The latter represents the operation of a JSR or BSR instruction.)
- The multiplexer RSPn updates the return stack pointer with either no change (0), decrement (-1, return stack size decreases), increment (+1, return stack size increases), or load from the parameter stack (TOS). The multiplexer is driven by a signal from control unit microcode.
- There is a secondary multiplexer, RSPnn driven by an auxiliary signal from the control unit that is able to decrement the return stack pointer regardless of the state of control unit microcode and the RSPn multiplexer. This is required because logic for the RTS instructions is hardwired rather than controlled by microcode.

2.3.8 The CPU Control Unit

The main components of the control unit are:

- A finite state machine (FSM) which determines next state logic and control signal outputs.
- Microcode held in a 2KB SRAM block that decodes instruction opcodes into control signals that will be routed directly to the datapath.
- A program counter and associated logic which steps program execution through memory in the appropriate order.
- Memory access logic which (a) routes memory write connections between the relevant bytes of the parameter stack registers and the appropriate system memory channels, and (b) accumulates byte or word length data from successive memory read cycles into a longword register which is connected to the datapath.

2.3.9 The finite state machine

The FSM is responsible for setting the values of control signals according to the current state and the current program instruction. Since the majority of CPU instructions execute in a single cycle, in most cases there is no change of state from instruction to instruction. The state in which all of the single-cycle instructions are executed is documented in the VHDL source code with the name "common". The state machine changes state from common to one of a number of other states for the following events:

- Instructions that take more than a single cycle to execute (?dup, multiply, divide, load literal, memory fetch, and memory store).
- Jumps, branches, and returns.
- Interrupts and traps.

2.3.10 Microcode

The CPU datapath requires 14 control lines to direct the various multiplexers and computation units appropriately for each instruction (plus one auxiliary control line for the RTS instruction). A simple "hardwired" decoder in the CPU control unit might require that these control lines be represented directly in the bits of the CPU instruction set. However by using microcode, the 14 control lines can be obtained from only 6 bits in the CPU instruction by configuring a 2K SRAM block with 6 address lines and 14 data lines. This enables higher code density through single-byte instruction encoding . There is a latency of one clock cycle for reading the microcode from the SRAM. This corresponds to part of the second stage of the pipeline ("DECODE AND EXECUTE").

2.3.11 Program counter

The control unit operates such that the code of the next instruction is being read from SRAM at the same time as the current instruction is being executed. This is part of the first stage of the pipeline ("FETCH OPCODE"). Update of the program counter is controlled by the FSM. At each cycle the possibilities for update of the program counter and return stack are:

- For single cycle instructions and load literal instructions, add one to the PC. Load literal instructions proceed byte by byte through the literal data using the PC.
- For other multi cycle instructions, add zero to the PC until the last cycle of the instruction and then add one. This is required to prime the first stage of the pipeline ("FETCH OPCODE") so that the next-but-one instruction is read from memory at the appropriate time
- For an external interrupt, redirect the program counter according to the vector number provided by the interrupt controller. In this case the current value of the program counter needs to be placed on the return stack since the current instruction will not be executed.
- For a TRAP or RTI_TRAP instruction, redirect the program counter to the trap vector. (The RTI_TRAP instruction is a two-phase instruction used for single stepping; first of all an RTI from the current trap routine is made, then one instruction at the current PC is executed, and then control is passed immediately back to the TRAP vector). For a TRAP instruction the PC of instruction following the current instruction is stored on the return stack.

- For a jump (JSR, JMP), redirect the program counter to the value currently on the top of the parameter stack (TOS). In the case of a JSR, also save the address of the next instruction on the return stack.
- For a branch instruction (BSR, BRA, BEQ), if the branch is taken redirect the program counter to the value of the PC plus the value on the top of stack. BRA and BEQ are two byte instructions and the PC will be on the second byte when the branch calculation is made. This needs to be taken into account by the assembler when calculating branch offsets.

2.3.12 Memory Channels

The CPU has three separate memory channels. Each of these channels has a read data bus, a write data bus plus memory control signals as required. A single address bus is common to all channels. The three data channels are as follows:

- An 8-bit data channel to SRAM.
- An 8-bit data channel to PSDRAM via the direct memory access (DMA) controller.
- An 16-bit data channel to PSDRAM via the DMA controller. This memory channel has twice the bandwidth of the 8-bit channel and is used for the read /write of word and longword data.

2.3.13 Other hardware components

The other principal hardware components implemented in VHDL are as follow:

Interrupt controller Responsible for prioritizing and scheduling interrupt signals from I/O devices to the CPU. The interrupt controller can be configured to accept additional interrupts from user-designed components in the digital logic layer that have their own interrupt vector routines. An interrupt mask register is available for enabling and suspending interrupts.

Video controller Responsible for providing a VGA signal for connection to a monitor. The video controller provides 256 colours and text/character graphics resolutions of 100*75, 100*60, 80*60, or 80*48 characters per screen, plus pixel graphics resolutions of 800*600 or 640*480 pixel per screen, double buffered.

Direct Memory Access (DMA) controller Responsible for multiplexing access from the CPU, the video controller, and other components to the 16MB PSDRAM on board the Nexys 2 development board.

RS232 controllers There are two by default, which provide an RS232 port for general purpose I/O and a dedicated RS232 port for connection to a third party SD-card reader/writer that serves as an external storage medium for the N.I.G.E. Machine.

PS/2 keyboard controller For direct connection to a PS/2 keyboard.

Memory-mapped hardware registers The interface between system control registers to the CPU address space. The hardware registers are expandable to accommodate user-designed components in the digital logic layer.

2.4 Implementation of system software

The N.I.G.E. Machine's FORTH environment is coded in assembly language and occupies just less than 8K of system memory. Published versions of FORTH were examined for guidance with the implementation[12][13]. For the most primitive FORTH words, there is a one-to-one correspondence with the CPU instruction set. Other FORTH words are implemented as machine language subroutines. There is no inner interpreter. The operating model on the N.I.G.E. Machine's FORTH environment could be classified as subroutine threaded or native.

Because the CPU instruction set is in general a subset of primitive FORTH words, the FORTH environment serves as the "local assembler" for the N.I.G.E. Machine. The N.I.G.E. Machine's FORTH implementation was developed in assembly language on a PC with a specially developed two-pass cross assembler and the cross assembler itself is written in standard ANSI FORTH.

The ANSI FORTH CORE[14] wordset has been implemented with very few exceptions, along with a selection of the most applicable words from the CORE EXTENSION, FACILITY, FILE ACCESS, PROGRAMMING TOOLS and STRING wordsets . Where minor departures from the ANSI standard have occurred they are due to reasons of implementation efficiency on an embedded system. In addition, a set of system specific words have been developed to enable convenient control of the N.I.G.E. Machine's facilities.

3 Results

3.1 Synthesis results obtained from the Xilinx ISE development software.

Version 13.2 of the Xilinx ISE tools was used to develop and synthesize the logic design for the Xilinx XC3S1200E Spartan-3E FPGA that is used in the N.I.G.E Machine. To put this FPGA in context, broadly speaking Xilinx has for several years offered two main families of device, Virtex and Spartan. Virtex are the high performance devices and Spartan are the economy or high-volume devices. Device families are also differentiated by generation numbers that indicate improving technology, typically driven by advances in the manufacturing process. Currently the latest devices in the Virtex family are at generation 7 and the latest devices in the Spartan family are at generation 6. Within the Spartan family, the prior generation to 6 was 3 (generation numbers 4 and 5 were skipped). Table 1 offers a comparison of Xilinx FPGAs according to the performance of the proprietary Xilinx softcore CPU, the MicroBlaze.

The Spartan-3E FPGA used in the N.I.G.E. Machine is therefore a one-generation-old device in the economy family of Xilinx FPGAs, and so relatively modest in comparison to the latest available technology. Nevertheless, it is in itself a highly capable device.

Device family	Typical MicroBlaze clock speed (3 stage pipeline format)
Virtex-6/7	240 MHz
Spartan-6	150 MHz
Spartan-3	50 MHz

Table 1: Xilinx device families compared according to MicroBlaze performance[15].

Resource	Used	Available	Utilization
4-input LUT's	3,884	17,344	22%
Slice flip flops	2,920	17,344	16%
2K block RAM	28	28	100%
Multipliers	8	28	28%

Table 2: N.I.G.E. Machine FPGA utilization on a Xilinx XC3S1200E, Spartan-3E FPGA.

Resource	LUT's
CPU	2,529
of which datapath	1,920
of which control unit	609
DMA controller	364
Hardware registers	280
Video controller	114
Diligent IO port	95
RS232 controller	71
Reset controller	52
PS/2 controller	39
System RAM	33
Interrupt controller	31

Table 3: N.I.G.E. Machine FPGA utilization at module level.

Parameter	
Maximum frequency	50.140 MHz
Minimum period	19.944 ns
Minimum input required time before clock	11.507 ns
Minimum output required time after clock	13.097 ns

Table 4: N.I.G.E. Machine timing summary on a Xilinx XC3S1200E, Spartan-3E FPGA.

3.2 Instruction frequency

Instruction	Frequency
LOAD.W	17.88%
JSR	9.17%
RTS and ,RTS*	9.06%
LOAD.B	6.47%
BEQ	4.60%
DUP	4.17%
OVER	3.67%
FETCH.L	3.56%
DROP	3.42%
STORE.L	3.02%
SWAP	2.91%
R>	2.37%
BRA	2.37%
FALSE	2.23%
FETCH.B	2.19%
1+	2.05%

* Of which RTS 6.36% and ,RTS 2.70%

Table 5: The 80% most used CPU instructions in the FORTH system software (as counted by the cross-assembler and ignoring execution frequency differences due to loops and conditional code, etc.)

3.3 Implications for design objectives

The FPGA logic utilization of 22% for a full micro-computer system on a relatively modest device is a very positive result. There is a significant amount of room remaining on the FPGA for the digital logic layer. If anything, further improvements in the design could be focused on addressing some of the current design limitations even at the expense of consuming a reasonable amount of additional logic area.

The maximum frequency of 50MHz was roughly as expected on this particular FPGA given the benchmark to the MicroBlaze (Table 1). Detailed analysis of the post place-and-route timing report did not reveal obvious bottlenecks in any one area of the design. The delays are roughly balanced between logic and routing. There was some evidence that the carry structure in the datapath adder unit may be a slightly slower path, and likewise the hardware registers.

The instruction frequency results were illuminating when considered in relation to the design objectives. The table well illustrates the load-store architecture of the CPU (LOAD.W is the most used instruction), and the subroutine threaded nature of FORTH (JSR and RTS are the second and third most used instructions). Given that high instruction throughput was specified as a design objective, it is interesting that the most used instruction (LOAD.W) is one of the minority of instructions that do not execute in a single cycle. (LOAD.W executes in three cycles as a direct result of the narrow instruction fetch that was discussed under design limitations.) Another optimization specified at the design stage, the compound RTS instruction, is only used in 30% of all return-from-subroutine instances, perhaps because of the limitation that it is only compatible with single cycle instructions that do not otherwise involve the return stack.

Clear priorities for future versions of the CPU softcore will be to widen the instruction fetch and to broaden the applicability of the compound RTS instruction, as well as identification of the hardware modules that can be further developed to increase the maximum potential clock frequency.

4 Application discussion

An experimental setup in applied physics served as an illustration of the use of the N.I.G.E. Machine. A short video is available to demonstrate [16]. The objective of the experiment was to measure the response of a certain light sensor to changes in the brightness of an LED. The light sensor in question was a light-to-frequency converter manufactured by the firm TAOS which comprises a photodiode and a current-to-frequency converter in a single package. The package has three connecting pins: 5V supply, ground, and output. The output signal is a square wave that varies in frequency from less than 1Hz to around 500kHz in response to the illumination of the photodiode. A tri-colour LED provided the illumination for the experiment. The LED has a common anode that connects to the positive supply voltage and three separate cathodes on the red, green, and blue elements that connect to ground via resistors of appropriate value. Three general purpose PNP transistors were connected between the cathodes and the resistors to provide switching for each colour element.

The circuit was constructed on breadboard. The output pin of the TAOS sensor and bases of the three PNP transistors were connected via hookup wires to an expansion port on the Nexys 2 board that routes directly to free pins on the FPGA.

A digital logic layer was designed to control and take measurements from the circuit in real time. As with the N.I.G.E. Machine overall, the Xilinx webpack tools were use for this development work. (The webpack tools are a free download from the Xilinx website[15].)

A frequency counter was required for measuring the output of the TAOS unit. A straightforward frequency counter module was developed in VHDL that comprised (a) a debounce process to eliminate any switching noise on the signal line, (b) a finite state machine to follow the square wave of the TAOS signal and (c) a counter and register to record the number of cycles of the square wave in one second. For controlling the three LED elements three variable duty cycle square wave outputs were required (with a variable duty cycle square wave, the average current to an LED element can be adjusted whilst keeping the supply voltage constant. If the square wave has a sufficiently high frequency (say 100Hz or more) there will not be any flicker observed by the light sensor.) A variable duty cycle square wave generator is very straightforward to design in VHDL and a suitable module was written in less than about 10 lines of logic description. Both the frequency counter and the variable duty cycle module were interfaced to the micro-computer layer of the N.I.G.E. Machine via hardware registers memory mapped to the address space of the CPU. This was accomplished by extending the existing hardware register module of the N.I.G.E. Machine. Three single-byte memory addresses were mapped to the duty cycles for the red, green and blue LED elements. Writing values of 0-100 to these registers adjusts the brightness of each of the red, green, and blue LED elements from full off to full on in real time. The output from the frequency counter was mapped to a longword memory address that could be read directly as the current frequency reading in Hz. The frequency reading in the register is automatically updated each one second in this design.

The digital logic layer was initially developed independently of the N.I.G.E. Machine in a stand-alone application. After a brief logic design was drawn up, the VHDL simulator was used to verify that the modules were operating as expected. A few small enhancements and simplifications were made at the simulation stage. After simulation was complete the Nexys2 board was programmed with the design of the two modules and connected to the breadboard and the electronic circuit. The modules were verified working as expected. At this stage a branch was made in the Subversion version control repository where all of the source code for the N.I.G.E. Machine is held. Using a branch structure in the version control system allowed a special version of the N.I.G.E. Machine source code to be created without affecting the main development path. The frequency counter and variable duty cycle modules were incorporated into the source code of the N.I.G.E. Machine and the Nexys2 board was programmed with the revised version. The functionality was again verified.

Finally, the FORTH environment was used to begin investigating the properties of the circuit. Initially small FORTH definitions were constructed to set the duty cycles of the three LED colour elements and read the frequency count from the TAOS sensor. These words were used to informally investigate such things as the dynamic range of the sensor/LED combination, the level of illumination background in a lit and unlit room, and the illumination levels of the three different LED colours. This general "tinkering" allowed the experimenter to gain a general feeling for the apparatus before running an actual experiment. Next, simple FORTH words were constructed to test the frequency output at various levels of duty cycle input and repeat these measurements over a series of input values. The results were read from the N.I.G.E. Machine and analyzed on a PC using Microsoft Excel. (The N.I.G.E. Machine includes an interface to a third party SD-card reader/writer than could also be used for logging the experimental results and transferring them to a PC for analysis.)

There are various ways in which the sophistication of the experimental set-up could be extended. For example the frequency counter module in the digital logic layer currently counts the signal from the light sensor over a period of one second. This is a simple and direct way to obtain a frequency count but a better dynamic range and/or response speed for measurements could be achieved by making the count period user selectable, for example 1/16 second, 1/4 second, 1 second, 4 seconds. This could be easily achieved with another writable hardware register that directs the frequency counter module accordingly. Or, the measurement system could be changed so that rather than counting the number of square waves over a fixed time period, the time taken to receive a certain number of waves would be measured instead, and thus the dynamic-range would be self-adjusting. An interrupt could also be used to signal when each new reading is ready.

The example described here illustrates the general approach of using a custom built digital logic interface for a new application and developing with a rapid prototyping environment. The initial digital logic interface functionality may be kept quite straightforward so as to minimize the time needed prepare the first design. As the user finds that there are experimental boundaries that need to be pushed back, it is straightforward to go back and iterate the design of the digital logic layer in a focused way to meet those goals. FORTH is used as a "language for direct communication between human beings and machines". The interactivity allows the experimenter to tinker with the apparatus at the initial stage, perhaps checking the overall characteristics or problem-solving any issues. When ready, the experiment proper can be built bottom up using the small routines that are already tested and understood.

5 Next steps and acknowledgments

The experiment described here was chosen just as an example to illustrate the capabilities of the N.I.G.E. Machine and the typical steps in developing an application. It is hoped that the N.I.G.E. Machine will find use in real scientific projects going forward. Future developments are anticipated that will enhance its capabilities including porting the design to more advanced FPGA development boards that offer additional speed and functionality (for example the Digilent Atlas board using a Spartan-6 FPGA).

I would like to acknowledge and thank Gunnar von Boehn and Jens Künzer of the Natami project for their thoughts on VHDL development and the N.I.G.E. Machine system manual, Jonathan Keelan of the Open University for his advice on the presentation of this paper, and the anonymous reviewers who's suggestions considerably improved its clarity of communication. The remaining deficiencies of the paper are solely the responsibility of the author.

References

[1] Volnei A. Pedroni, “Circuit Design and Simulation with VHDL”, 2nd edition, MIT Press, 2011

[2] The author, http://www.youtube.com/watch?v=0v-HuVLRoUc

[3] James Bowman , “J1: a small Forth CPU Core for FPGAs” in *EuroForth*, 2010

[4] K. Schleisiek, “MicroCore,” in *EuroForth*, 2001.

[5] B. Paysan, “b16-small – Less is More,” in *EuroForth*, 2004.

[6] E. Hjrtland and L. Chen, “EP32 - a 32-bit Forth Microprocessor,” in Canadian Conference on Electrical and Computer Engineering, pp. 518–521, 2007.

[7] E. Jennings, “The Novix NC4000 Project,” Computer Language, vol. 2, no. 10, pp. 37–46, 1985.

[8] Rible, John, "QS2: RISCing it all," Proceedings of the 1991 FORML Conference, Forth Interest Group, Oakland, CA (1991), pp. 156-159.

[9] Stephen Pelc, “Programming FORTH”, MPE, 2011

[10] Enoch O. Hwang, “Digital Logic and Microprocessor Design with VHDL”, C.L. Engineering, 2005

[11] P. J. Koopman, Jr., “Stack computers: the new wave”, Halsted Press, 1989

[12] Brad Rodriguez, “Moving Forth”, The Computer Journal, 1993

[13] Phil Burk, “pFORTH”, public domain, 1994-2012

[14] Elizabeth D. Rather and Edward K. Conklin, “Forth Programmer’s Handbook”, 3rd edition, Booksurge Publishing, 2008

[15] Xilinx website, http://www.xilinx.com

[16] The author, http://www.youtube.com/watch?v=0Kj5EMdnkMk