The N.I.G.E. Machine

Andrew Read

Table of Contents

[1. Introduction 3](#_Toc321927256)

[2. System Architecture 4](#_Toc321927257)

[3. 32-bit softcore CPU 5](#_Toc321927258)

[3.1. Design objectives and general features 5](#_Toc321927259)

[3.2. CPU specifications 6](#_Toc321927260)

[3.3. CPU operation 9](#_Toc321927261)

[3.3.1. Instruction encoding 9](#_Toc321927262)

[3.3.2. CPU design: the datapath and control unit 10](#_Toc321927263)

[3.4. Datapath of the parameter stack 11](#_Toc321927264)

[3.5. Datapath of the Return Stack 14](#_Toc321927265)

[3.6. The Control Unit 15](#_Toc321927266)

[3.6.1. The finite state machine 15](#_Toc321927267)

[3.6.2. Microcode 15](#_Toc321927268)

[3.6.3. Program counter 16](#_Toc321927269)

[3.6.3. Memory channels 17](#_Toc321927270)

[4. Other hardware components 19](#_Toc321927271)

[4.1. Interrupt controller 19](#_Toc321927272)

[4.2. Reset Controller 20](#_Toc321927273)

[4.3. Direct Memory Access (DMA) Controller 21](#_Toc321927274)

[4.4. Video controller 22](#_Toc321927275)

[4.5. RS232 controllers 24](#_Toc321927276)

[4.5. PS/2 controller 25](#_Toc321927277)

[4.7. System hardware registers 26](#_Toc321927278)

[4.8. Digilent I/O port 26](#_Toc321927279)

[4.9. PSDRAM 26](#_Toc321927280)

[4.10. SRAM 27](#_Toc321927281)

[5. CPU instruction set reference 28](#_Toc321927282)

[6. System software 37](#_Toc321927283)

[6.1 FORTH implementation 37](#_Toc321927284)

[6.2 Cross assembler 37](#_Toc321927285)

[6.1 FORTH word set reference 37](#_Toc321927286)

[6.3.1 CORE words 37](#_Toc321927287)

[6.3.2 CORE EXTENSION words 41](#_Toc321927288)

[6.3.3 FACILITY words 42](#_Toc321927289)

[6.3.4 FILE ACCESS words 42](#_Toc321927290)

[6.3.5 PROGRAMMING TOOLS words 42](#_Toc321927291)

[6.3.6 STRING words 42](#_Toc321927292)

[6.3.7 N.I.G.E. Machine specific FORTH words 43](#_Toc321927293)

[8. System memory map 46](#_Toc321927294)

[8.1 Overall memory structure 46](#_Toc321927295)

[8.2 System memory in SRAM 46](#_Toc321927296)

[8.3 hardware registers 46](#_Toc321927297)

[8.4 System memory in PSDRAM 47](#_Toc321927298)

[9. FPGA statistics and measurements 48](#_Toc321927299)

[9.1 Overall utilization 48](#_Toc321927300)

[9.2 Module level utilization 48](#_Toc321927301)

[9.3 Timing summary 48](#_Toc321927302)

[10. Design decisions 49](#_Toc321927303)

[10. References and acknowledgements 53](#_Toc321927304)

# 1. Introduction

The N.I.G.E. Machine is a complete computer system running on an FPGA development board. The system comprises a stack-based, 32-bit softcore CPU, video controller, I/O ports, RAM, and system software in the form of a FORTH environment. The system hardware is coded in VHDL and the FORTH environment is coded in the assembly language of the softcore CPU. All of the design for the CPU and most of the other components is original.

The N.I.G.E. Machine has been implemented and tested on a Digilent Nexys 2 development board with a Xilinx XC3S1200E FPGA with a clock frequency of 50 MHz. With a PS/2 keyboard and VGA monitor connected to the board the system is a fully operational native FORTH environment from power on.

The N.I.G.E. Machine’s intended applications are in four areas:

* Prototyping scientific hardware applications. The N.I.G.E. Machine is an open platform for bringing together hardware and software. It provides the flexibility of using VHDL to develop very specific custom hardware components and electronic interfaces, that are directly available within a local computer environment with all the interactive benefits of a FORTH interpreter and a compiler. This is intended provide interesting possibilities for developing one-off experimental apparatus.
* Micro-computer style programming on modern hardware. The 8/16-bit micro-computers of the 1980’s attracted enthusiasts because of their simplicity and direct access to hardware. At that time the bottleneck to developing interesting projects was the limited computing power. With modern computers, the bottleneck is the confusion that comes from multiple layers of software abstraction and legacy code. The N.I.G.E. Machine is a “pure-bred” descendant of the micro-computer concept in its usability and focus, but which also takes full advantage of modern advances in silicon computing power.
* Experimentation in a native FORTH environment. These days FORTH is usually only available either as a hosted application on a modern PC or within an embedded micro-controller that lacks the full scope of a micro-computer environment. The N.I.G.E. Machine is a stand-alone, native environment that provides alternative possibilities for FORTH development.
* Investigating tradeoff decisions and approaches in digital design. Begin implemented in VHDL, the N.I.G.E. machine is a living design that continues to iterate as problems are better understood and new ideas come to mind. In the future the N.I.G.E. machine will develop and improve as it is ported to new development boards and more sophisticated FPGA’s.

The purpose of this manual is two-fold. Firstly, to provide a sufficient description of the N.I.G.E. Machine to allow interested users to confidently program the system as a stand-alone micro-computer in FORTH. Secondly, to describe the N.I.G.E Machine’s main components with enough detail and clarity to guide those wanting to develop or extend the hardware in VHDL.

# 2. System Architecture

The N.I.G.E. Machine has been designed around an FGPA development board that can be thought of as providing facilities in three areas:

* The FPGA itself, in this case a Xilinx XC3S1200E, provides the following (a) soft logic elements into which all of the hardware design is encoded and placed, (b) static RAM (SRAM), 50Kbytes in 2K blocks that is used for SRAM system memory, stack memory, various buffers, and CPU microcode, and (c) facilities such as dedicated multipliers and carry logic that are used within the CPU datapath
* Pseudo Static Dynamic RAM (PSDRAM), 16Mbytes, in a separate I.C. on the development board. The PSDRAM provides additional data memory for and text and pixel graphics buffers for the video controller and other application uses
* Various connectors and associated circuitry, for example a VGA connector with a simple digital-to-analog converter, an RS232 D-Sub connector with appropriate voltage conversion, a PS/2 keyboard connector, etc.

The hardware components of the N.I.G.E. Machine are implemented within the FPGA as follows:

|  |  |
| --- | --- |
| **Component** | **Description** |
| 32-bit softcore CPU | Central processing unit of the N.I.G.E Machine |
| Interrupt controller | Prioritizes and schedules interrupt requests from other hardware units and interfaces them with the CPU |
| Reset controller | Responsible for timing the power-on and flash reset of all hardware units |
| Direct Memory Access (DMA) Controller | Interfaces the external PSDRAM and provides separate channels to all hardware units that require access, e.g. the CPU and video controller |
| Video controller | Drives a standard VGA display with both character/text graphics and pixel graphics |
| RS232 controllers, x 2 | One for general serial port access and one intended for interfacing with an external SD card reader/writer. Further RS232 ports can be added |
| PS/2 controller | Interface for a standard PS/2 keyboard |
| System hardware registers | Registers which govern the various hardware units are memory mapped into the CPU address space enabling their control via software |
| Digilent I/O port | Debugging facility for the flashing system software during development |

# 3. 32-bit softcore CPU

## 3.1. Design objectives and general features

The softcore CPU has been designed with two goals in mind: (a) usability and performance in a micro-computer FORTH environment, and (b) suitability for directly controlling embedded hardware. Key design features that resulted are as follows:

* Stack based CPU storage: two stacks (parameter and return) are directly integrated with the CPU and writable in a single cycle. There are no other registers
* Code density: CPU microcode allows almost all instructions are encoded in a single byte to maximizes code density
* Throughput: a 3-stage pipeline provides throughput of one cycle per instruction for most instructions
* Flexible memory access: separate instructions are available to read and write memory in byte, word, and longword format. Even address alignment is not required when accessing word or longword data in SRAM system memory (PSDRAM must be evenly addressed)
* Subroutine performance: a compound RTS instruction can be overlaid on top of most single byte instructions, saving one clock cycle on each subroutine return
* Deterministic execution: all execution is deterministic, including conditional branches (PSDRAM access through the DMA controller being excepted).
* Interrupt response time: typically 4 cycles only, including branching to the appropriate interrupt vector code (with no registers, save/restore of a register file is not required)

The design goals and hardware tradeoffs also resulted in some limitations. It is planned to address these with improvements in future versions of the N.I.G.E. machine. Current limitations include:

* Program memory space: the CPU is only able to execute instruction code residing within SRAM and cannot execute instruction code residing in PSDRAM. This limitation arose from the desire to maintain single cycle throughput and deterministic execution at all times
* Lack of floating point: there is no hardware floating point facility
* 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
* 32-bit operands for multiplication and division: the hardware multipliers in the fabric of the current FPGA are limited to 32-bit operands, although MULT produces a 64-bit result

## 3.2. CPU specifications

|  |  |
| --- | --- |
| **Item** | **Specification** |
| Stacks / registers | 2 dedicated data stacks: parameter and return. There are no other registers |
| Parameter stack width / depth | 32 bits wide, 512 cells deep |
| Return stack width /depth | 32 bits wide, 512 cells deep |
| Data format for memory access | Big-endian |
| Instruction size | Most instructions are encoded in a single byte. Otherwise up to 5 bytes including in-line literal data. |
| Instruction throughput | 1 cycle per instruction for most instructions |
| Instruction latency | 3 cycles for most instructions based on a 3 stage pipeline   * Read instruction from system SRAM * Decode microcode using control unit SRAM block * Compute and rewrite to the stack |
| Stack manipulation: 15 instructions | * **NOP** (no operation) * FORTH words: **DROP**, **DUP**, **?DUP** \*, **SWAP**, **OVER**, **NIP**, **ROT**, **>R**, **R@**, **R>**, * **LOADPSP** (load parameter stack pointer onto stack), * **LOADRSP**(load return stack pointer onto stack), * **SAVEPSP** (save parameter stack pointer from stack), * **SAVERSP** (save return stack pointer from stack)   \* instruction executes in 2 cycles |
| Math operations: 12 instructions | * **+**, **-**, **NEGATE**, **1+**, **1-**, * **ASL** (arithmetic shift left), **ASR** (arithmetic shift right) * **MULTS**\*, **MULTU**\*, Signed and unsigned hardware multiplication take 32 bit operands and produce a 64 bit result * **ADDX**, **SUBX** (add /subtract with carry), * **DIVS** \*\*, **DIVU** \*\*   Signed and unsigned hardware division take 32 bit operands and produce a 32 bit quotient and a 32 bit remainder  \* instructions execute in 5 cycles  \*\* instructions execute in around 40 cycles |
| Comparison operations: 11 instructions | * Bitwise equality tests: **=**, **<>**, * Signed comparisons: **<**, **>**, * Unsigned comparisons: **U<**, **U>**, * Comparisons with zero: **0=**, **0<>**, **0<**, **0>** * **FALSE** (returns zero) |
| Bitwise operations: 7 instructions | Boolean operations: **AND**, **OR**, **INVERT**, **XOR**  Shift operations: **LSL** (logical shift left), **LSR** (logical shift right)  Sign extension to 32 bits: **XBYTE**, **XWORD** |
| Memory operations: 6 instructions | * **FETCH.L**, **STORE.L** * **FETCH.W**, **STORE.W** * **FETCH.B**, **STORE.B**   The CPU is capable of addressing RAM in byte, word, or longword units. The CPU transparently differentiates between addressing on-chip SRAM and external SDRAM with separately optimized logic. SRAM access takes 2 cycles for bytes, 3 cycles for words and 5 cycles for long words. SDRAM access is via the DMA controller and will depend on the controller to respond with a data ready signal. The CPU utilizes a 16 bit wide DMA datapath for word and longword access and an 8 bit wide datapath for byte access.  No address alignment for word and longword access is required for on-chip SRAM access, but even alignment is required for external PSDRAM access due to the access characteristics of the PSDRAM module on the Nexys2 board. Violations do not result in an exception or failure but incorrect memory access results will occur. |
| Load literal operations: 3 instructions | * **LOAD.B** (alternative assembler mnemonic **#.B**) * **LOAD.W** (or **#.W**) * **LOAD.L** (or **#.L**)   The literal instructions load the sack with a big-endian inline literal value, zero extended in the case of #.B and #.W. The #.L instruction is 5 bytes in length and executes in 5 cycles, likewise the instruction length and duration for #.W and #.B is 3 and 2 respectively. The XWORD and XBYTE instructions can be used to sign extend word and byte literals as required. |
| Flow control: 6 instructions | * **JMP** (jump to the address on the parameter stack) * **BSR**, **JSR** (either branch or jump to subroutine) * **RTS** (return from subroutine) * **BEQ**, **BRA** (conditional and unconditional branches)   The JMP, BSR, and JSR instructions are single byte length and pop the top of the parameter stack as the next instruction address. In the case of JMP and JSR, this is an absolute address, in the case of BSR it is a relative offset to the current instruction address.  JSR and BSR also push the address of the next following instruction onto the return stack before making the jump or branch. RTS pops the return address from the return stack and returns execution to that point.  BEQ and BRA are 2 byte instructions. The branch offset is encoded as a signed (two’s complement) 14 bit number, giving a branch range of 8 Kbytes either forwards or backwards calculated from the second byte of the instruction. BEQ is a conditional branch which pops the top item from the parameter stack. The branch is taken when equal to zero.  Standalone flow control instructions generally take exactly two cycles to execute. For efficiency, an RTS instruction can be overlaid on the encoding of a single byte arithmetic instruction so that the total execution time for both operations will be two cycles only (e.g. DROP,RTS) |
| Exception handling: 3 instructions | * **TRAP** * **RTS\_TRAP** * **RTI**   The TRAP instruction is used to insert a breakpoint into a program that will cause the CPU to execute a subroutine jump to the TRAP vector. The TRAP vector code should be terminated with an RTS. Alternatively, if the trap vector code is terminated with an RTS\_TRAP then the CPU will return to the next instruction of the main program, execute it, and then execute a subroutine branch to the TRAP vector again. This feature is used for single stepping through a user application.  The RTI instruction is use to terminate the vector program code for external interrupt handlers. In addition to returning from a subroutine in the same way as an RTS instruction, the RTI signals to the interrupt scheduler that the current interrupt has been completed |
| Reserved: 2 instructions | Reserved instruction encodings could be used in future as headers for two byte instructions, allowing the CPU to include additional functionality such as 3D array calculations, floating point stack operations, or others. |
| Interrupts | The CPU supports hardware interrupts (in addition to the software TRAP instruction described above). When an interrupt trigger occurs the CPU completes the current instruction and then executes a subroutine branch into the interrupt vector table. Each interrupt trigger also supplies an interrupt vector number (1 – n) which determines the associated interrupt vector address. Entries in the interrupt vector table consist of 2 bytes BRA instructions that further vector to the interrupt handling code.  There is a BRA instruction at address 0x00 which is the power-on and reset vector, and a BRA instruction at address 0x02 which is the TRAP vector.  Assuming that the current instruction when an interrupt occurs is a single cycle instruction, then the interrupt response time will be 4 cycles only, including further branching from the interrupt vector table to the interrupt handling code. |

## 3.3. CPU operation

### 3.3.1. Instruction encoding

The default instruction size of the CPU is single byte in order to maximize code density. Where literal data is required as part of an instruction it follows in the succeeding bytes.

Encoding is as follows. Firstly, 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.

|  |  |  |
| --- | --- | --- |
| **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 |

For ordinary instructions, bits 5 – 0 are read as an integer in the range 0 – 63 that identifies the instruction in question. This is called the “opcode” in N.I.G.E. Machine’s documentation. 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). Thus the interpretation of bits 5 – 0 of the instruction depends on the setting of bit 7 as follows:

|  |  |  |
| --- | --- | --- |
| **Bit 7** | **Bit 6** | **Bits 5 - 0** |
| 1 | x | High part of branch address |
| 0 | x | Opcode 0 - 63 |

Branches and load literal instructions are the only multi-byte encoded instructions. They are organized as follows:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Bits 5 - 0 of  1st byte** | **2nd byte** | **3rd byte** | **4th byte** | **5th byte** |
| **Branch** | 14 bit branch address  (big endian) | | - | - | - |
| **LOAD.L** | opcode | Longword literal (big endian) | | | |
| **LOAD.W** | opcode | Word literal (big endian) | | - | - |
| **LOAD.B** | opcode | Byte literal | - | - | - |

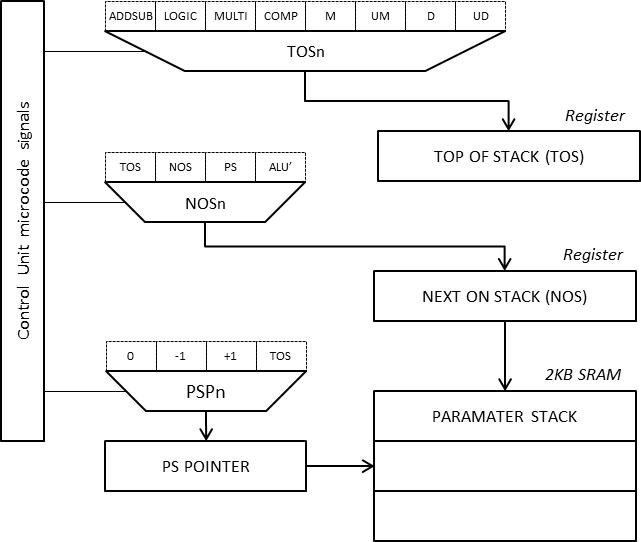
### 3.3.2. CPU design: the datapath and control unit

The CPU comprises a datapath and a control unit. 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. Essentially 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 one. The control unit is also responsible for adjusting the program counter (PC) so that program instructions are read from memory in the appropriate order including program jumps and branches, dealing with interrupts and other exceptions, and supporting data transfers between system memory and the data path.

3.4. Datapath of the parameter stack

The figure below illustrates the parameter stack datapath.



The top-of-stack (TOS) and next-on-stack (NOS) storage locations are 32 bit hardware registers and 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 system memory where it can be accessed by the CPU. 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 to determine the data flow. The main multiplexers controlling the parameter stack are as follows:

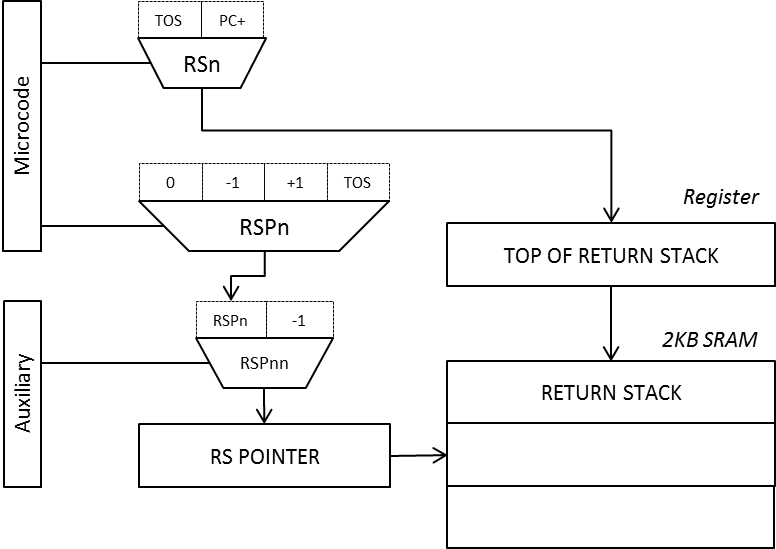
* The multiplexer TOSn selects the value for the update of the TOS register from one of eight computation units: addition/subtraction, logic operations, multipurpose, comparison, multiply, unsigned multiply, divide and unsigned divide.
* The multiplexer NOSn selects the value for update of the NOS register from: TOS, NOS (i.e. itself, no-update), the item below NOS in the parameter stack RAM, and 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 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 further directed by signals from the control unit microcode as necessary according to the functionality required by each function. Some of the computation functionality is provided by Xilinx CORE modules that often leverage special purpose circuitry available within the FPGA such as hardware multipliers and carry logic structures. The computation units are summarized below.

|  |  |
| --- | --- |
| **Computation unit** | **Available outputs** |
| ADDSUB | * **TOS** (no change) * **–TOS** (negate) * **NOS + TOS** (add) * **NOS – TOS** (subtract) * **NOS + TOS** (add with carry) * **NOS – TOS** ( subtract with carry) * **TOS + 1** ( increment) * **TOS – 1** (decrement)   The adder/subtractor is implemented using a XILIX 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. This allows the N.I.G.E. machine 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. |
| LOGIC | * **TOS AND NOS** (binary AND) * **TOS OR NOS** (binary OR) * **INVERT TOS** (binary NOT) * **TOS XOR NOS** (binary XOR) * **TOS LSL** (logical shift left 1 bit) * **TOS LSR** ( logical shift right 1 bit) * **TOS ASL** (arithmetic shift left 1 bit) * **TOS ASR** (arithmetic shift right 1 bit)   The logic computation is implemented in VHDL |
| MULTI | * **NOS** * **Parameter stack memory** (i.e. current third-on-stack) * **Top of Return Stack** * **Parameter stack pointer** (i.e. parameter stack count) * **Return stack pointer** (i.e. return stack count) * **XBYTE** (sign extension of TOS from 8 to 32 bits) * **XWORD** (sign extension of TOS from 16 to 32 bits) * **Memory data register** (either for load literal or fetch operations)   Multi is a general purpose multiplexer implemented in VHDL. |
| COMP | * **NOS = TOS** (Boolean equals) * **NOS <> TOS** (Boolean not equals) * **NOS < TOS** (Boolean signed less than) * **NOS > TOS** (Boolean signed greater than) * **NOS U< TOS** (Boolean unsigned less than) * **NOS U> TOS** (Boolean unsigned greater than) * **TOS = 0** (Boolean equals zero) * **TOS <>0** (Boolean not equals zero) * **TOS <0** (Boolean less than zero) * **TOS >0** (Boolean greater than zero) * **0** (zero, i.e. false)   The comparison unit is implemented using a XILINX CORE template with supporting logic in VHDL.The following inactive functionality is also available within the comparison unit but not accessible from the instruction set due to instruction set design tradeoffs.   * **NOS <= TOS** (Boolean less than or equals) * **NOS >= TOS** (Boolean greater than or equals) * **TOS <= 0** (Boolean less than or equals zero) * **TOS >= 0** (Boolean greater than or equals zero) * **0xFFFFFFF** (i.e. true) |
| M | * **NOS \* TOS** (signed multiply)   Signed multiply is implemented using an on-chip XILINX pipelined multiplier with 32 bit operands and a 64 bit result. It completes operation in 5 clock cycles. |
| UM | * **NOS \* TOS** (unsigned multiply)   Unsigned multiply is implemented using an on-chip XILINX pipelined multiplier with 32 bit operands and a 64 bit result. It completes operation in 5 clock cycles. |
| D | * **NOS / TOS** (signed divide with quotient and remainder)   Signed divide is implemented in logic fabric using a XILINX CORE template using 32 bit operands, a 32 bit quotient and a 32 bit remainder. It completes in 42 clock cycles. |
| UD | * **NOS / TOS** (unsigned divide with quotient and remainder)   Unsigned divide is implemented in logic fabric using a XILINX CORE template using 32 bit operands, a 32 bit quotient and a 32 bit remainder. It completes in 41 clock cycles. |

3.5. Datapath of the Return Stack

The figure below 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.

3.6. The 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 which decodes instruction opcodes into control signals that can be routed directly to the datapath
* A program counter and associated logic which step 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.

### 3.6.1. 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

The manner in which the FSM impacts the operation of the microcode, the program counter, and memory access logic is documented within the following sections.

### 3.6.2. 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 some or all of these control lines to 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. The enables higher code density with single-byte instruction encoding to be achieved.

There is a latency of one clock cycle for reading the microcode from the SRAM. This corresponding to the second stage of the 3 stage pipeline of the CPU (that is instruction read, microcode read, and register update).

The FSM controls use of microcode as follows:

* Ordinarily the least significant 5 bits of the current instruction (opcode) are passed to the microcode as an address value to access the 2KB SRAM block.
* Upon an interrupt, an override occurs and the microcode is fed with the special opcode for an interrupt since the current instruction is being deferred in favor of the interrupt.
* Upon a BRA or BEQ instruction, a NOP override to microcode occurs (i.e. the microcode is fed with the same opcode as per the NOP instruction) to prevent the lower 5 bits, which are actually now part of the branch offset in the instruction encoding, being misinterpreted as an opcode.
* Multi-cycle instructions such as load literal, memory read, and division require that the CPU control lines be as a NOP instruction until the last cycle when the new data becomes available. At that time the microcode is fed with the required special opcodes to complete register update.

### 3.6.3. Program counter

The control unit operates a simple pipeline whereby the code of the next instruction is being read from SRAM at the same time as the current instruction is being executed. This allows for the throughput of 1 cycle per instruction, although the latency in the control unit remains two cycles (A further stage for data register update in the datapath increasing the total CPU latency to three cycles).

Update of the program counter is controlled by the FSM. At each cycle the possibilities for update of the program counter are 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 pipeline at the appropriate time so that the next-but-one instruction is read from memory just as the state machine prepares to execute the next instruction.
* 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 simple 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.

### 3.6.3. Memory channels

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

* An 8-bit data channel to SRAM
* An 8-bit data channel to PSDRAM via the DMA controller
* An 16-bit data channel to PSDRAM via the DMA controller

Depending on the memory address and the instruction being executed, the FSM will determine which of these channels should be active, set the appropriate control signals, and route the address and data connection.

The following describes how the memory channel components are managed by the FSM in various situations:

* Address out bus – either the current top of stack (TOS) value (for fetch and store instructions) a or the current program counter value (for all other instructions, including load literal and instructions that do no access system memory).
* Data out bus – since all store instructions have the address value on the top of stack (TOS) and the data value as the next on stack (NOS), data out in any state will either be the high or low word of NOS, or one of the four bytes of NOS.
* Data in bus – data in is routed to a 32 bit accumulator register that is able to read either 16 or 8 bits at a time at the low-end, and concurrently shift from the low end to the higher end of the register the data read on the previous cycle. When filled the accumulator is read into the TOS register. In this way the use of datapath multiplexers is reduced and the state machine becomes more economical than would be the case if TOS were populated directly byte-by-byte or word-by-word from memory.
* Control signals – SRAM access requires only a write request (WRQ) control signal. Since both read and write operations to SRAM are guaranteed to complete in a single clock cycle the state logic can proceed from state to state without waiting for any acknowledgement signal. Access to PSDRAM through the DMA controller also requires a read request (REQ) signal and the FSM must wait and monitor the ready (RDY) signal from the DMA controller before advancing to the next state in the memory access process.

# 4. Other hardware components

## 4.1. Interrupt controller

The interrupt controller is a scheduler for managing CPU interrupts. Interrupt requests (IRQ’s) are made via edge-triggered signal lines from the interrupt generating hardware units. An interrupt mask applied by the interrupt controller determines which signal lines are considered active at any one time. This interrupt mask is readable and writable by the CPU as a memory mapped hardware register. Post masking, each interrupt signal line is assigned a priority number (1 – n), with the lower numbers having the higher priority.

Interrupts on the N.I.G.E. Machine are blocking. Once an interrupt is in progress no further interrupts may occur until the original interrupt completes, regardless of their relative priority. In the default configuration of the N.I.G.E. Machine there are six hardware interrupt signal lines connected to the interrupt controller.

|  |  |  |
| --- | --- | --- |
| **Interrupt number** | **CPU vector address** | **Hardware connection** |
| 1 | 0x04 | Read Data Available (RDA) of the RS232 port 1 |
| 2 | 0x06 | Read Data Available (RDA) of the RS232 port 0 |
| 3 | 0x08 | Transfer Bus Enable (TBE) of the RS232 port 1 |
| 4 | 0x0A | Transfer Bus Enable (TBE) of the RS232 port 0 |
| 5 | 0x0C | PS/2 data available (input) |
| 6 | 0x0E | System generated millisecond interrupt |

RDA signals from the two RS232 ports are given highest priority since these signals indicate that new data is available and must be read before being overwritten. RS232 port 1 is given top priority since it is intended to connect to an SD-card reader/writer and will generally operate at a higher baud rate than port 0, which is intended as a general purpose terminal. TBE signals from the RS232 ports on interrupts 3 and 4 indicate that current byte transmission is now finished and that the transfer bus should be reloaded. The PS/2 interrupt number 5 indicates a key code has been received from the keyboard. A system generated timer driven interrupt operates at 1 kHz on interrupt 6, but this is masked off at power-on as the FORTH environment does not use it.

The memory mapped register used to operate the video controller are as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Register** | **Address** | **Width (bytes)** | **Description** |
| IRQ\_MASK | 0xF81C | 2 | Mask applied to all interrupt signal lines by the interrupt controller. ‘1’ indicates interrupt active. Bit 0 is unused. |

The interrupt controller can easily be extended to accept interrupts from custom hardware units. As currently configured there are 15 available interrupt lines of which 6 are already assigned as described above. The VHDL file for the interrupt decoder includes comments indicating how this would be done, but essentially all that is required is for a new interrupt signal like to be routed to the interrupt controller and for some small extensions to be made to the finite state machine to accommodate it.

The following is a more detailed description of the interrupt controller process:

1. Signal lines from interrupt generating hardware units are masked with the interrupt mask. (There are no non-makeable interrupts.)
2. An interrupt request (IRQ) is generated when a masked signal line transitions from low to high.
3. When the CPU is not currently executing an interrupt subroutine, the lowest numbered interrupt requested at a particular cycle is actioned. Unsuccessful IRQ’s occurring during that same cycle are carried forward indefinitely until they can be satisfied.
4. When the CPU is currently executing an interrupt, no further interrupts are actioned regardless of their priority. All IRQ’s received at this time are carried forward indefinitely until they can be satisfied.
5. Execution of an RTI instruction by the CPU signals to the interrupt controller that the CPU has completed the current interrupt subroutine and is now free to accept another interrupt.
6. Unsatisfied IRQ’s occurring multiple times with same interrupt number are not counted. Only a single IRQ of that interrupt number is carried forward.

## 4.2. Reset Controller

The reset controller provides a reset signal to the CPU, DMA controller and video adapter. A short reset signal is given to all three sub-systems at power on. In addition, a two-second reset signal is triggered when the Digilent IO interface writes to memory address 0x00 as part of a firmware update. This allows time for the firmware update to complete before the CPU resumes execution.

## 4.3. Direct Memory Access (DMA) Controller

The DMA controller provides shared access for the CPU, the graphics adapter, and other components to the 16MB of PSDRAM (Pseudo Static DRAM) provided on the Nexys2 board.

The available DMA channels and their characteristics are scheduled below:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Channel** | **Address width** | **Data width** | **Direction** | **Pri-or-ity** | **Notes** |
| PSDRAM | 23 | 16 | Bi | n/a | External connection from the FPGA to the 16MB PSDRAM. The PSDRAM is byte or word addressable using a 23 bit address bus (i.e. address bits 24 downto 1) with separate lines for enabling hi and lo byte access |
| CPU-16 | 23 | 16 | Bi | 3 | 16 bit wide read/write used by the CPU operations that read or write word and longword data |
| CPU-8 | 24 | 8 | Bi | 4 | 8 bit wide read/write used by the CPU operations that read or write byte data |
| GFX | 9 | 8 | Read only | 2 | Feeds pixel graphics data to the video controller from a 512 byte SRAM buffer |
| TXT | 7 | 16 | Read only | 1 | Feeds character/text graphics data to the video controller from a 256 word SRAM buffer |
| Others | 8/16 | 8/16 | Bi | \* | Other channels are available for expansion. For example for use by logic units such as a blitter or hardware line drawer |

The available PSDRAM operating in random access mode does not have sufficient throughput speed to support the requirements of the video controller with a 25MHz pixel clock. Instead the DMA controller includes two fast on-board SRAM (static RAM) buffers for feeding character and pixel data to the graphics controller in synchrony with the VGA pixel clocks. The buffer maintenance logic of the DMA controller leverages the burst mode capabilities of the PSDRAM to fill the buffers with sequential data from PSDRAM at a much higher rate than would be possible with random access.

Highest DMA priority is given to the character and pixel graphics buffer maintenance logic so that the buffers can be filled before the pixel clock begins each relevant scan line.

The base address of both the pixel and character graphics frame buffers in PSDRAM are set by memory mapped registers that can be read and written by the CPU (see the section on System Hardware Registers below), thus enabling video features such as flexible double buffering.

## 4.4. Video controller

The video controller drives a standard VGA interface socket on the Nexys2 board and provides both character/text and pixel graphics with the following specifications:

|  |  |  |  |
| --- | --- | --- | --- |
| **Display** | **Resolution** | **Color depth** | **Aspect ratio** |
| Character/text graphics | 80\*60 characters, each 8\*8 pixels | 8 bits | 1:1 |
| Pixel graphics | 640\*480 pixels | 8 bits | 1:1 |

Character graphics are stored in the PSRAM frame buffer as 80\*60 consecutive 16 bit words. Each word represents a single on-screen character position. The high byte (lower memory address) is the color data and the low byte (higher memory address) is the character selection.

There are two modes of color data interpretation. In mode 0, the highest 4 bits of the color byte (i.e. bits 23 down to 16 of the word = bits 7 down to 4 of the high byte) are interpreted as the background color for that character position, and the lowest 4 bits as the foreground color. Mode 0 therefore provides 16 foreground and 16 background colors for each character position. In mode 1, all 8 color bits are interpreted as the foreground color, and a common background color for the whole screen is set via a separate CPU addressable hardware register. Mode 1 therefore provides 256 colors for each character position and a single screen background color. Mode selection is made by another hardware register.

Character definitions are held in a dedicated 2KB SRAM block. Each character is 8\*8 pixels with a 1:1 aspect ratio. There are 256 characters which include the ASCII set, the Greek alphabet, graphics characters, some Japanese kana, and some scientific symbols. The character definitions SRAM block is memory mapped and can be read and written to by the CPU, thus allowing user-defined character-definitions.

The frame buffer allocated to the character/text display is 19,200 bytes in size which corresponds to 120 lines of characters/text (i.e. twice the number of lines as there are on screen). The FORTH environment uses a technique for text display where scrolling is achieved by moving the base address of the character/text frame buffer forward by 160 bytes (80 characters, one line) until the end of the buffer is reached. This is a much faster system that also considerably reduces the amount of memory coping needed when screen scrolling compared with a static frame buffer the same size as the display area.

Pixel graphic data is stored in PSRAM as 640\*480 consecutive bytes (i.e. “chunky” format). Each byte is interpreted as the color of a single on-screen pixel, with 256 colors available.

The character/text and pixel graphics displays may be independently switched on and off via a hardware register. When both are active, output pixels are combined with XOR. When either one is inactive, DMA to the PSRAM is switched off for that channel thus saving memory bandwidth for the CPU.

The power-on configuration of the N.I.G.E. Machine is character/text graphics on in mode 1 and pixel graphics off.

Memory mapped registers used to operate the video controller are as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Register** | **Address** | **Width (bytes)** | **Description** |
| CHAR\_ZERO | 0xF800 | 4 | Pointer to the character/text graphics frame buffer in PSDRAM. The default location is 0x10700, with 19,200 bytes reserved. The character graphics frame buffer must lie within PSRAM, i.e. at or above address 0x10000 |
| GFX\_ZERO | 0xF804 | 4 | Pointer to the pixel graphics frame buffer in PSDRAM. The default location is 0x015200, with 307,200 bytes reserved. The pixel graphics frame buffer must lie within PSRAM, i.e. at or above address 0x10000 and on a 256 byte boundary. |
| BACKGROUND | 0xF808 | 1 | Background color displayed in character/text mode 1 |
| VIDEO\_MODE | 0xF809 | 1 | Bit 0 – Character/text graphics off or on (0/1)  Bit 1 – Character/text graphics mode 0 or 1 (0/1)  Bit 2 – Pixel graphics off or on (0/1)  Bit 3-7 - Unused |

## 4.5. RS232 controllers

There are two RS232 controllers in the default configuration of the N.I.G.E. Machine. Port 0 drives a standard DB9 RS232 interface socket on the Nexys 2 board. Port 1 is routed to a PMOD expansion port where it can be connected to an external SD-card reader/writer.

The configuration of the RS232 interface fixed as below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Data bits** | **Stop bits** | **Parity bits** | **Baud rate** | **Handshaking** |
| 8 | 1 | 0 | Software selectable | none |

The controllers are operated by the use of memory mapped registers and interrupts as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Register** | **Address (HEX)** | **Width (bytes)** | **Description** |
| RS232\_0\_DATAIN | 0xF80A | 1 | This register is updated with latest byte received from port 0 each time RDA (see below) goes high |
| RS232\_0\_DATAOUT | 0xF80B | 1 | Writing to this register will trigger output of the byte to port 0, provided that TBE (see below) is high |
| RS232\_0\_UBRR | 0xF80C | 2 | Baud rate selector, set according to the following UBRR = CPU / (baud+1) / 16  The default value is 325, corresponding to 9600 baud with a 50MHz clock frequency. Port 0 register. |
| RS232\_1\_DATAIN | 0xF80E | 1 | Port 1 read data. |
| RS232\_1\_DATAOUT | 0xF80F | 1 | Port 1 write data |
| RS232\_1\_UBRR | 0xF810 | 2 | Port 1 UBRR |
| RS232\_SIGNAL | 0xF812 | 1 | Bit 0 – RDA (Read Data Available) on port 1, goes high for one cycle when a byte is received.  Bit 1 – RDA (Read Data Available) on port 0  Bit 2 – TBE (Transfer Bus Enable) on port 1, is held high when the output bus is available and held low when the bus is busy  Bit 3 – TBE (Transfer Bus Enable) on port 0  Bits 4 – 7 – Unused |

|  |  |  |  |
| --- | --- | --- | --- |
| **Interrupt** | **Priority** | **Vector (HEX)** | **System interrupt handler** |
| RDA\_1\_EVENT | 1 | 0x04 | Triggered by the RS232 RDA signal, the interrupt handler copies the received byte to a 256 byte RS232 input circular buffer. The RS232 input buffer is tested and read with the FORTH words SKEY? and SKEY. The buffer is overwrite protected; once the buffer is full newly received bytes are discarded until sufficient calls to SKEY free space in the buffer. Port 1 |
| RDA\_0\_EVENT | 2 | 0x06 | RDA on port 0 |
| TBE\_1\_EVENT | 3 | 0x08 | Triggered by the RS232 TBE signal, the interrupt handler will consecutively output the bytes held in the RS232 output buffer. The location and size of the RS232 output buffer are set for each write by the FORTH word STYPE, which thus provides asynchronous RS232 output. Port 1 |
| TBE\_0\_EVENT | 4 | 0x0A | TBE on port 1 |

## 4.5. PS/2 controller

The PS/2 controller connects a PS/2 keyboard interface socket on the Nexys2 board. Memory mapped registers and interrupts used to operate the controller are as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Register** | **Address (HEX)** | **Width (bytes)** | **Description** |
| PS2\_DATAIN | 0xF813 | 1 | This register is updated with latest byte received from the port. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Interrupt** | **Priority** | **Vector (HEX)** | **System interrupt handler** |
| PS2\_EVENT | 5 | 0x0C | Triggered by an interrupt from the PS/2 controller |

The FORTH environment includes an interrupt handler and a decoder for translating raw PS/2 keystrokes into ASCII values.

## 4.7. System hardware registers

The system hardware registers are a set of registers implemented with FPGA logic elements used for I/O interface and communication with the various peripheral controllers. The registers have been memory mapped to the CPU address space adjacent to the top of the SRAM address space. The system memory map later in this manual contains full details.

## 4.8. Digilent I/O port

The Nexys 2 board incorporates a USB port for connection to a PC running the Digilent programming software package, Adept. Adept allows configuration of the FPGA and also facilitates general data transfer from the PC to the Nexys 2 board. As an aid to debugging, a Digilent I/O port has been incorporated into the N.I.G.E. machine to allow re-flashing of the system software in SRAM without needing to resynthesize the entire design.

## 4.9. PSDRAM

The Nexys 2 board incorporates 16 MB of Pseudo Static Dynamic RAM (PSDRAM) which is accessible via the DMA controller. The N.I.G.E. Machine CPU cannot execute code residing in PSDRAM, but may freely access PSDRAM for data storage. The frame buffers for character/text and pixel graphics are held in PSDRAM along with some FORTH language buffers.

## 4.10. SRAM

The Xilinx XC3S1200E FPGA includes 56KB of 2KB each dual ported SRAM blocks. These are utilized in the N.I.G.E. Machine as follows:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Component** | **RAM** | **CPU address** | **Port A** | **Port B** | **Comments** |
| System RAM | 44KB | 0x0000 | CPU address space | Digilent I/O port | System RAM holds the executable code and is pre-loaded with the system software (FORTH environment) at power on |
| Parameter stack | 2KB | 0xE000 | CPU address space | CPU datapath | To facilitate a throughput of one cycle per instruction for most instructions dedicated SRAM blocks are used for both the parameter and return stacks. These are accessed directly by the CPU datapath as if they were a register file, but are also available in the CPU address space |
| Return stack | 2KB | 0xE800 | CPU address space | CPU datapath |
| Character RAM | 2KB | 0xF000 | CPU address space | Video controller | Character RAM is pre-configured with the N.I.G.E. machine 256 character 8x8 pixel character set. It can be accessed by the CPU for reprogramming as desired |
| Microcode | 2KB | n/a | CPU control unit | n/a | The CPU control unit microcode is organized as 14 bits of data available over 64 addresses (6 bits) for decoding CPU instructions |
| DMA buffers | 4KB | n/a | DMA controller | Video controller | Two 2KB buffers are filled by the DMA controller with data from the text and graphics buffers in PSDRAM via burst mode. The video controller reads display data from these buffers at the pixel clock rate of 25MHz. |

# 5. CPU instruction set reference

|  |  |  |  |
| --- | --- | --- | --- |
| **Assembler mnemonic** | **Instruction length (bytes)** | **Encoding** | **Duration (cycles)** |
| **Description** | | **Parameter stack effect** ( before -- after)  ( 3rd 2nd 1st on stack --) | **Return stack effect** |

|  |  |  |  |
| --- | --- | --- | --- |
| **NOP** | 1 byte | 0x00 | 1 cycle |
| No operation | | ( --) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **DROP** | 1 byte | 0x01 | 1 cycle |
| Remove top item from parameter stack | | ( x --) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **DUP** | 1 byte | 0x02 | 1 cycle |
| Duplicate the top stack item | | ( x -- x x) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **?DUP** | 1 byte | 0x03 | 2 cycles |
| Duplicate the top stack item only if non zero | | (x -- x x | x) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **SWAP** | 1 byte | 0x04 | 1 cycle |
| Exchange the two top stack items | | ( x y -- y x) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **OVER** | 1 byte | 0x05 | 1 cycle |
| Make a copy of the second item on the stack | | ( x y -- x y x) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **NIP** | 1 byte | 0x06 | 1 cycle |
| Dispose of the second item on the stack | | ( x y -- y) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **ROT** | 1 byte | 0x07 | 1 cycle |
| Rotate the top three stack times so that the second item becomes top | | (x y z -- z x y) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **>R** | 1 byte | 0x08 | 1 cycle |
| Remove the top item from the parameter stack and place it on the return stack | | ( x --) | ( -- x) |

|  |  |  |  |
| --- | --- | --- | --- |
| **R@** | 1 byte | 0x09 | 1 cycle |
| Copy the top item from the return stack to the parameter stack | | ( -- x) | ( x -- x) |

|  |  |  |  |
| --- | --- | --- | --- |
| **R>** | 1 byte | 0x0A | 1 cycle |
| Remove the top item from the return stack and place it on the parameter stack | | ( -- x) | ( x --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **LOADPSP** | 1 byte | 0x0B | 1 cycle |
| Load the parameter stack with the current value of the parameter stack pointer. The stack pointer is the count of items currently on the stack and also directs the CPU datapath to the first stack item held in SRAM | | ( -- PSP) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **LOADRSP** | 1 byte | 0x0C | 1 cycle |
| Load the parameter stack with the current value of the return stack pointer. The stack pointer is the count of items currently on the stack and also directs the CPU datapath to the first stack item held in SRAM | | ( -- RSP) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **SAVEPSP** | 1 byte | 0x0D | 1 cycle |
| Save the top item from the stack as the current parameter stack pointer | | ( PSP --) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **SAVERSP** | 1 byte | 0x0E | 1 cycle |
| Save the top item from the stack as the current return stack pointer | | ( --) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **+** | 1 byte | 0x0F | 1 cycle |
| Add two 32 bit integer numbers. x3 = x1 + x2 | | ( x1 x2 -- x3) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **-** | 1 byte | 0x10 | 1 cycle |
| Subtract two 32 bit integer numbers.   x3 = x1 - x2 | | ( x1 x2 -- x3) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **NEGATE** | 1 byte | 0x11 | 1 cycle |
| Negate a 32 bit integer in two’s complement format | | (x1 -- x2) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **1+** | 1 byte | 0x12 | 1 cycle |
| Add 1 | | ( x1 -- x2) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **1-** | 1 byte | 0x13 | 1 cycle |
| Subtract 1 | | ( x1 -- x2) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **2\*** | 1 byte | 0x14 | 1 cycle |
| Arithmetic shift left | | ( x1 -- x2) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **2/** | 1 byte | 0x15 | 1 cycle |
| Arithmetic shift right | | ( x1 -- x2) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **MULTS** | 1 byte | 0x16 | 5 cycles |
| Multiply two signed 32 bit integers to produce a 64-bit integer that is held in the top two stack positions, highest part top of stack | | ( x1 x2 -- d3) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **MULTU** | 1 byte | 0x17 | 1 cycle |
| Multiply two unsigned 32 bit integers to produce a 64-bit integer that is held in the top two stack positions, highest part top of stack | | ( u1 u2 -- ud3) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **ADDX** | 1 byte | 0x18 | 1 cycle |
| Add two integers with extend flag as carry. The extend flag resides within the datapath and is not otherwise accessible to software. The flag is only affected by arithmetic instructions. | | ( x1 x2 -- x3) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **SUBX** | 1 byte | 0x19 | 1 cycle |
| Subtraction with extend flag as borrow. The extend flag resides within the datapath and is not otherwise accessible to software. The flag is only affected by arithmetic instructions. | | ( x1 x2 -- x3) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **DIVS** | 1 byte | 0x1A | 42 cycles |
| Divide two 32-bit signed numbers to produce a 32-bit quotient (top of stack) and a 32-bit remainder (next on stack) | | (x1 x2 -- u-rem u-quot) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **DIVU** | 1 byte | 0x1B | 41 cycles |
| Divide two 32-bit unsigned numbers to produce a 32-bit quotient (top of stack) and a 32-bit remainder (next on stack) | | (x1 x2 -- rem quot) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **=** | 1 byte | 0x1C | 1 cycle |
| Returns -1 (true) if x1 = x2 | | ( x1 x2 -- flag) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **<>** | 1 byte | 0x1D | 1 cycle |
| Returns -1 (true) if x1 <> x2 | | ( x1 x2 -- flag) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **<** | 1 byte | 0x1E | 1 cycle |
| Returns -1 (true) if x1 < x2 | | ( x1 x2 -- flag) | ( --) |
| **>** | 1 byte | 0x1F | 1 cycle |
| Returns -1 (true) if x1 > x2 | | ( x1 x2 -- flag) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **U<** | 1 byte | 0x20 | 1 cycle |
| Returns -1 (true) if u1 < u2, where u is unsigned | | ( u1 u2 -- flag) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **U>** | 1 byte | 0x21 | 1 cycle |
| Returns -1 (true) if u1 > u2, where u is unsigned | | ( u1 u2 -- flag) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **0=** | 1 byte | 0x22 | 1 cycle |
| Returns -1 (true) if x1 = 0. Equivalent to Boolean NOT | | ( x1 -- flag) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **0<>** | 1 byte | 0x23 | 1 cycle |
| Returns -1 (true) if x1 <> 0 | | ( x1 -- flag) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **0<** | 1 byte | 0x24 | 1 cycle |
| Returns -1 (true) if x1 < 0 | | ( x1 -- flag) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **0>** | 1 byte | 0x25 | 1 cycle |
| Returns -1 (true) if x1 > 0 | | ( x1 -- flag) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **FALSE** | 1 byte | 0x26 | 1 cycle |
| Place zero (false) on the stack. Equivalent to ZERO | | ( -- 0) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **AND** | 1 byte | 0x27 | 1 cycle |
| Bitwise AND | | ( x1 x2 -- x3) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **OR** | 1 byte | 0x28 | 1 cycle |
| Bitwise OR | | ( x1 x2 -- x3) | ( --) |
| **INVERT** | 1 byte | 0x29 | 1 cycle |
| Bitwise NOT | | ( x1 -- x2) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **XOR** | 1 byte | 0x2A | 1 cycle |
| Bitwise XOR | | ( x1 x2 -- x3) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **LSL** | 1 byte | 0x2B | 1 cycle |
| Logical shift left | | ( x1 -- x2) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **LSR** | 1 byte | 0x2C | 1 cycle |
| Logical shift right | | ( x1 -- x2) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **XBYTE** | 1 byte | 0x2D | 1 cycle |
| Sign extend a byte to 32 bits | | ( x1 -- x2) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **XWORD** | 1 byte | 0x2E | 1 cycle |
| Sign extend a word to 32 bits | | ( x1 -- x2) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **FETCH.L** | 1 byte | 0x2F | 5 cycles in SRAM |
| Fetch a longword from memory, big endian | | ( addr -- n) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **STORE.L** | 1 byte | 0x30 | 5 cycles in SRAM |
| Store a longword in memory, big endian | | ( n addr --) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **FETCH.W** | 1 byte | 0x31 | 3 cycles in SRAM |
| Fetch a word from memory, big endian | | ( addr -- n) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **STORE.W** | 1 byte | 0x32 | 3 cycles in SRAM |
| Store a word in memory, big endian | | ( n addr --) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **FETCH.B** | 1 byte | 0x33 | 2 cycles in SRAM |
| Fetch a byte from memory | | ( addr -- n) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **STORE.B** | 1 byte | 0x34 | 2 cycles in SRAM |
| Store a byte in memory | | ( n addr --) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **LOAD.B** or **#.B** | 2 bytes | 0x35, x1 | 2 cycles |
| Fetch inline byte to stack and zero extend | | ( -- x) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **LOAD.W** or **#.W** | 3 bytes | 0x36, x2, x1 | 3 cycles |
| Fetch inline word to stack and zero extend. High byte first | | ( -- x) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **LOAD.L** or **#.L** | 5 bytes | 0x37, x4, x3, x2, x1 | 5 cycles |
| Fetch inline longword to stack. Highest byte first | | ( -- x) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **JMP** | 1 byte | 0x38 | 2 cycles |
| Redirect program execution to the address on the parameter stack | | ( addr --) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **BSR** | 1 byte | 0x39 | 2 cycles |
| Redirect program execution by the address offset on the parameter stack and place the original next following instruction address on the return stack | | ( offset --) | ( -- addr) |

|  |  |  |  |
| --- | --- | --- | --- |
| **JSR** | 1 byte | 0x3A | 2 cycles |
| Redirect program execution to the address on the parameter stack and place the original next following instruction address on the return stack | | ( addr --) | ( -- addr) |

|  |  |  |  |
| --- | --- | --- | --- |
| **TRAP** | 1 byte | 0x3B | 2 cycles |
| Jump to the trap vector (address 0x02) and place the original next following instruction address on the return stack. Used for breakpoint debugging. | | ( --) | ( -- addr) |

|  |  |  |  |
| --- | --- | --- | --- |
| **RTS\_TRAP** | 1 byte | 0x3C | 2 cycles |
| Return from subroutine, execute one program instruction and trap again. Used for single step debugging | | ( --) | ( addr --)  ( -- addr) |

|  |  |  |  |
| --- | --- | --- | --- |
| **RTI** | 1 byte | 0x3D | 2 cycles |
| Return from an interrupt routine. Similar to RTS but also changes the interrupt controller state to unblocks further interrupts | | ( --) | (addr --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **RTS** | 1 byte | 0x40 | 2 cycles |
| Return from a subroutine that was entered via a JSR or BSR instruction | | ( --) | ( addr --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **,RTS** | 1 byte | (0x40 OR opcode) | 1 cycle |
| As RTS but is a compound for any single-cycle instruction that does not itself reference or impact the return stack. The compound instruction saves one cycle and one byte on each subroutine return (e.g. DROP,RTS). | | ( --) | ( addr --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **BEQ** | 2 bytes | (0x80 OR hi), lo | 3 cycles |
| Branch if the top of stack item is zero. The top 6 bits of the branch offset are in the first instruction byte, the bottom 8 bits of the branch address follow in a second instruction byte. The branch offset is calculated from the address of the second byte | | ( flag --) | ( --) |

|  |  |  |  |
| --- | --- | --- | --- |
| **BRA** | 2 bytes | (0xC0 OR hi), lo | 3 cycles |
| Branch. The top 6 bits of the branch offset are in the first instruction byte, the bottom 8 bits of the branch address follow in a second instruction byte. The branch offset is calculated from the address of the second byte | | ( --) | ( --) |

# 6. System software

The N.I.G.E. Machine includes a FORTH environment that is available at power on. FORTH has been described as “a language for direct communication between human beings and machines”. It combines an interactive shell with a compiler. Commands, known as FORTH words, can be issued directly at the keyboard for immediate execution or compiled into the definition of new words. FORTH is also a stack based language. Functions are implemented as words that accept numeric parameters from the top of the stack and leave return values on the stack.

## 6.1 FORTH implementation

The N.I.G.E. Machine’s FORTH environment is coded in assembly language and occupies just less than 8K of system memory. There is a one-to-one correspondence between the most primitive FORTH words and the CPU instruction set. Other FORTH words are implemented as machine language subroutines or as inline code (there is no inner interpreter). The operating model on the N.I.G.E. Machine is therefore native and subroutine threaded.

## 6.2 Cross assembler

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. However the N.I.G.E. Machine’s FORTH implementation was developed on a PC with a specially developed two-pass cross assembler and the cross assembler itself is written in standard ANSI FORTH.

## 6.1 FORTH word set reference

The following reference is organized according to the categories of the ANSI FORTH documentation. Status indicates whether the word has been implemented on the N.I.G.E. Machine. Y indicates yes, N no, and that Y\* that the word is implemented by with some limitation or difference as compared with ANSI FORTH.

### 6.3.1 CORE words

|  |  |  |
| --- | --- | --- |
| **Word** | **Status** | **Notes** |
| ! | Y | See also W! |
| # | N | Use U# instead. Division with a 64-bit dividend is not implemented on the N.I.G.E. Machine. |
| #> | N | Use U#> instead. See #. |
| #S | N | Use U#S instead. See #. |
| ‘ | Y |  |
| ( | Y |  |
| \* | Y |  |
| \*/ | Y\* | The intermediate value is single (32-bit) precision only |
| \*/MOD | Y\* | The intermediate value is single (32-bit) precision only |
| + | Y |  |
| +! | Y |  |
| +LOOP | Y |  |
| , | Y | See also W, M, and $, |
| - | Y |  |
| . | Y |  |
| .” | Y |  |
| / | Y |  |
| /MOD | Y |  |
| 0< | Y |  |
| 0= | Y |  |
| 1+ | Y |  |
| 1- | Y |  |
| 2! | N | Less suitable for a big-endian processor |
| 2\* | Y |  |
| 2/ | Y |  |
| 2@ | N | Less suitable for a big-endian processor |
| 2DROP | Y |  |
| 2DUP | Y |  |
| 2OVER | Y |  |
| 2SWAP | Y |  |
| : | Y |  |
| ; | Y |  |
| < | Y |  |
| <# | Y |  |
| = | Y |  |
| > | Y |  |
| >BODY | N | Would be a no operation in the N.I.G.E. Machine FORTH environment. Will not be implemented for space and efficiency reasons |
| >IN | Y |  |
| >NUMBER | Y |  |
| >R | Y |  |
| ?DUP | Y |  |
| @ | Y |  |
| ABORT | Y |  |
| ABORT” | N | Will not be implemented for space and efficiency reasons |
| ABS | Y |  |
| ACCEPT | Y |  |
| ALIGN | N | Would be a no operation in the N.I.G.E. Machine FORTH environment. Will not be implemented for space and efficiency reasons |
| ALIGNED | N | Would be a no operation in the N.I.G.E. Machine FORTH environment. Will not be implemented for space and efficiency reasons |
| ALLOT | Y |  |
| AND | Y |  |
| BASE | Y |  |
| BEGIN | Y |  |
| BL | Y |  |
| C! | Y | See also W! |
| C, | Y | See also W, |
| C@ | Y | See also W@ |
| CELL+ | Y |  |
| CELLS | Y |  |
| CHARS | N | Would be a no operation in the N.I.G.E. Machine FORTH environment. Will not be implemented for space and efficiency reasons |
| CONSTANT | Y |  |
| COUNT | Y |  |
| CR | Y |  |
| CREATE | Y |  |
| DECIMAL | Y |  |
| DEPTH | Y |  |
| DO | Y |  |
| DOES> | Y |  |
| DROP | Y |  |
| DUP | Y |  |
| ELSE | Y |  |
| EMIT | Y |  |
| ENVIRONMENT? | N | Will not be implemented for space and efficiency reasons |
| EVALUATE | Y |  |
| EXECUTE | Y |  |
| EXIT | Y |  |
| FILL | Y | See FILL.W |
| FIND | Y |  |
| FM/M | N | Will not be implemented for space and efficiency reasons |
| HERE | Y |  |
| HOLD | Y |  |
| I | Y |  |
| IF | Y |  |
| IMMEDIATE | Y |  |
| INVERT | Y |  |
| J | Y |  |
| KEY | Y |  |
| LEAVE | Y |  |
| LOOP | Y |  |
| LSHIFT | Y |  |
| M\* | Y |  |
| MAX | Y |  |
| MIN | Y |  |
| MOD | Y |  |
| MOVE | Y |  |
| NEGATE | Y |  |
| OR | Y |  |
| OVER | Y |  |
| POSTPONE | Y |  |
| QUIT | Y |  |
| R> | Y |  |
| R@ | Y |  |
| RECURSE | Y |  |
| REPEAT | Y |  |
| ROT | Y |  |
| RSHIFT | Y |  |
| S” | Y | See also C” and ,” |
| S>D | N | Equivalent to FALSE on the N.I.G.E. Machine. Will not be implemented for space and efficiency reasons. |
| SIGN | Y |  |
| SM/REM | N | Division with a 64-bit dividend is not supported |
| SOURCE | Y |  |
| SPACE | Y |  |
| SPACES | Y |  |
| STATE | Y |  |
| SWAP | Y |  |
| THEN | Y |  |
| U. | Y |  |
| U< | Y |  |
| UM\* | Y |  |
| UM/MOD | N | Division with a 64-bit dividend is not supported |
| UNLOOP | Y |  |
| UNTIL | Y |  |
| VARIABLE | Y |  |
| WHILE | Y |  |
| WORD | Y |  |
| XOR | Y |  |
| [ | Y |  |
| [‘] | Y |  |
| [CHAR] | Y |  |
| ] | Y |  |

### 6.3.2 CORE EXTENSION words

|  |  |  |
| --- | --- | --- |
| .( | Y |  |
| .R | Y |  |
| 0<> | Y |  |
| 0> | Y |  |
| <> | Y |  |
| ?DO | Y |  |
| AGAIN | Y |  |
| BUFFER: | Y | Allocates space in PSDRAM. Suitable for larger data blocks |
| C” | Y | In the interpretation state C” will copy the text until the following “ to the PAD as a counted string and return its address |
| CASE | Y |  |
| COMPILE, | Y |  |
| DEFER | Y |  |
| ENDCASE | Y |  |
| ENDOF | Y |  |
| FALSE | Y | Returns 0 |
| HEX | Y |  |
| INTERPRET | Y | Interpret a line from the input buffer |
| IS | Y |  |
| MARKER | Y |  |
| NIP | Y |  |
| OF | Y |  |
| PAD | Y |  |
| PARSE | Y |  |
| PICK | Y |  |
| RESTORE-INPUT | Y | SAVE-INPUT and RESTORE-INPUT use internal variables to store the input source specification. RESTORE-INPUT does not return a flag |
| SAVE-INPUT | Y | See RESTORE-INPUT |
| TRUE | Y |  |
| U.R | Y |  |
| U> | Y |  |
| UNUSED | Y |  |
| WITHIN | Y |  |
| \ | Y |  |

### 6.3.3 FACILITY words

|  |  |  |
| --- | --- | --- |
| KEY? | Y | See also KKEY?, S0KEY? And S1KEY? |

### 6.3.4 FILE ACCESS words

|  |  |  |
| --- | --- | --- |
| INCLUDED | Y |  |

### 6.3.5 PROGRAMMING TOOLS words

|  |  |  |
| --- | --- | --- |
| .S | Y |  |
| ? | Y |  |
| DUMP | Y |  |
| WORDS | Y |  |
| STATE | Y |  |

### 6.3.6 STRING words

|  |  |  |
| --- | --- | --- |
| COMPARE | Y |  |
| SLITERAL | Y | See also CLITERAL |

### 6.3.7 N.I.G.E. Machine specific FORTH words

#### General screen control

|  |  |  |
| --- | --- | --- |
| TAB | ( -- addr) | VARIABLE pointing to the current size of tab stops. The default is 3 |
| ACOL | ( -- addr) | Byte length VARIABLE pointing to the color code for user input keystrokes |
| BCOL | ( -- addr) | Byte length VARIABLE pointing to the color code for output from the FORTH environment |
| TCOL | ( -- addr) | Byte length VARIABLE pointing to the color code for current output by VEMIT or CSR-PLOT |
| CLS | ( --) | Clear the screen |
| COLOR-TABLE | ( --) | Display a table of the 256 available character/text colors |

#### Programming the text/character display

|  |  |  |
| --- | --- | --- |
| VEMIT | ( n --) | Emit a character to the VDU |
| VTYPE | ( c-addr n --) | Type a string to the VDU |
| CSR-X | ( -- addr) | VARIABLE pointing to the current column position of the cursor |
| CSR-Y | ( -- addr) | VARIABLE pointing to the current row position of the cursor |
| CSR-ADDR | ( -- addr) | Return the memory address of the current cursor position within the character/text buffer in PSDRAM |
| CSR-PLOT | ( x --) | Plot the ASCII character x at the current cursor position |
| CSR-ON | ( --) | Plot the cursor symbol at the current cursor position. The character at that position is saved in an internal variable |
| CSR-OFF | ( --) | Unplot the cursor symbol from the current cursor position and restore the character which was previously there |
| CSR-FWD | ( --) | Advance the cursor by one character |
| CSR-BACK | ( --) | Move back the cursor by one character |
| CSR-TAB | ( --) | Advance the cursor to the next tab stop |
| SCROLL | ( n -- flag) | Scroll the screen fwd or back n lines within the 120 line frame buffer. Returns true if out of range, or false otherwise |
| NEWLINE | ( --) | Scroll the screen downwards by one line of text and return the cursor to the first column of the blank line below |

#### External I/O

|  |  |  |
| --- | --- | --- |
| **Word** | **Stack effect** | **Description** |
| S0EMIT | ( x --) | Emit a character to RS232 port 0 |
| S0KEY? | ( -- flag) | Check if a character is waiting to be read from the 256 byte circular buffer maintained for RS232 port 0 |
| S0KEY | ( -- n) | Wait for and read the next character available at RS232 port 0 |
| S0TYPE | ( c-addr n --) | Type a string to RS232 port 0, asynchronous. |
| S1EMIT | ( x --) | Emit a character to RS232 port 1 |
| S1KEY? | ( -- flag) | Check if a character is waiting to be read from the 256 byte circular buffer maintained for RS232 port 1 |
| S1KEY | ( -- n) | Wait for and read the next character available at RS232 port 1 |
| S1TYPE | ( c-addr n --) | Type a string to RS232 port 1, asynchronous. |
| S1ZERO | ( --) | Reset the 256 byte circular buffer maintained for RS232 port 1 and abandon characters currently waiting to be read from there |
| KKEY? | ( -- flag) | Check if a character is waiting to be read from the 256 byte circular buffer maintained for the PS/2 keyboard |
| KKEY | ( -- n) | Wait for and read the next character available from the PS/2 keyboard |
| PS2DECODE | ( n -- n) | Decode a PS/2 scan code into ASCII. Returns 0 if there is no valid ASCII match. PS2DECODE is called directly by the PS/2 interrupt controller. |

#### I/O redirection

|  |  |  |
| --- | --- | --- |
| >REMOTE | ( --) | Redirect FORTH environment output to RS232 port 0 |
| >LOCAL | ( --) | Redirect FORTH environment output to the screen |
| <REMOTE | ( --) | Receive FORTH environment input from RS232 port 0 |
| <LOCAL | ( --) | Receive FORTH environment input from the keyboard |

#### Compiler extensions

|  |  |  |
| --- | --- | --- |
| HERE1 | ( -- addr) | VARIABLE pointing to the dictionary pointer for the PSDRAM dictionary space. Only used by BUFFER: |
| INLINESIZE | ( -- addr) | VARIABLE pointing to the maximum code-length in bytes that the compiler will compile inline rather than as a subroutine call. The default value is 10 and the minimum allowable is 9 since certain code, such as LOOP code, much be compiled inline |
| W, | (w -- ) | Allocate 2 bytes in the dictionary and store a word from the stack |
| M, | (addr u --) | Allocate and store u bytes from addr into the dictionary. u is not saved in the dictionary. Compiles a string or other block of data from memory |
| $, | ( addr u --) | Allocate and store u bytes from addr into the dictionary. u is is compiled as the first byte. Compiles a counted string. |
| LITERAL | ( n --) | Compile a literal to the dictionary |
| CLITERAL | ( addr u --) | Compile to the dictionary a string literal as an executable that will be re-presented at run time as a counted string c-addr |

#### Other common FORTH words

|  |  |  |
| --- | --- | --- |
| BINARY | ( --) | Set BASE = 2 |
| WAIT | ( u --) | Wait for u milliseconds |
| RESET | ( --) | Reset the N.I.G.E. Machine to power on configuration but otherwise preserve memory contents |
| NOT | ( n – n) | Equivalent to 0= |
| XBYTE | ( n – n) | Sign extend a byte on the stack to 32 bits |
| XWORD | ( n –n) | Sign extend a word on the stack to 32 bits |
| W@ | ( addr – n) | Fetch a word from memory |
| W! | ( n addr --) | Store a word in memory |
| FILL.W | ( addr n w --) | Fill a region of memory with n words w. FILL.W utilizes the STORE.W machine language instruction and will be faster than FILL in accessing PSDRAM |
| UPPER | (x -- X) | Convert one ASCII character to uppercase |
| DIGIT | ( char base -- n true | char false ) | Convert a single ASCII character to a number in the given base |
| NUMBER? | ( c-addr u - false | n true ,) | Convert an ASCII string to a number and return with a success or failure flag |
| COMP | ( n1 n2 – n) | Return -1 if n1<n2, +1 if n1>2, 0 if n1=n2 |
| $= | ( c-addr1 u1 c-addr2 u2 -- flag) | Test two strings for equality. Case insensitive. |
| ERROR | ( n --) | Print “ERROR n” and ABORT |

#### SD card reader/writer

|  |  |  |
| --- | --- | --- |
| LOAD | ( c-addr n -- len true | false) | Load a file with the name given by c-addr n into the file buffer from an SD card reader/writer attached to RS232 port 1 |
| SAVE | (c-addr n) | Save the file currently in the file buffer to an SD card reader/writer attached to RS232 port 1 with the name given by c-addr n |
| DOS | ( --) | Enter terminal mode for direct communication with an SD card reader/writer attached to RS232 port 1. Exit terminal mode by pressing ESC |

# 8. System memory map

The memory configuration of the N.I.G.E. Machine is as follows

## 8.1 Overall memory structure

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Address (Hex)** | **Address (Dec)** | **Bytes** | **Type** | **Description** |
| **0000** | **0** | **49152** | FPGA SRAM | System memory |
| **F800** | **63488** | **32** | FPGA logic elements | Hardware registers |
| **010000** | **65536** | **~16MB** | External PSDRAM | System memory |

## 8.2 System memory in SRAM

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Address (Hex)** | **Address (Dec)** | **Bytes** | **R/W** | **Description** |
| **0000** | **0** | **45056** | R/W | FORTH dictionary area (i.e. program and data memory) |
| **B000** | **45056** | **12288** | *-* | *not used* |
| **E000** | **57344** | **2048** | R/W | Parameter stack addressable by CPU |
| **E800** | **59392** | **2048** | R/W | Return stack addressable by CPU |

## 8.3 hardware registers

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Address (Hex)** | **Address (Dec)** | **Bytes** | **R/W** | **Description** |
| **F800** | **63488** | **4** | R/W | Pointer to the character/text graphics buffer in PSDRAM |
| **F804** | **63492** | **4** | R/W | Pointer to the pixel graphics buffer in PSDRAM |
| **F808** | **63496** | **1** | R/W | Background color displayed in mode 1 |
| **F809** | **63497** | **1** | R/W | Video mode |
| **F80A** | **63498** | **1** | R | RS232 port 0 data in |
| **F80B** | **63499** | **1** | W | RS232 port 0 data out |
| **F80C** | **63500** | **2** | R/W | RS232 port 0 UBRR |
| **F80E** | **63502** | **1** | R | RS232 port 1 data in |
| **F80F** | **63503** | **1** | W | RS232 port 1 data out |
| **F810** | **63504** | **2** | R/W | RS232 port 1 UBRR |
| **F812** | **63506** | **1** | R | RS232 status signals |
| **F813** | **63507** | **1** | R | PS2 port data in |
| **F814** | **63508** | **4** | R | 32-bit free running clock counter at 50MHz |
| **F818** | **63512** | **4** | R | 32-bit free running millisecond counter |
| **F81C** | **63516** | **2** | R/W | IRQ mask for interrupt controller |
| **F81E** | **63518** | **2** | W | Nexys II board seven segment display |
| **F821** | **63520** | **1** | R | Nexys II board switches |
| **F823** | **63521** | **2016** | *-* | *not used* |

## 8.4 System memory in PSDRAM

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Address (Hex)** | **Address (Dec)** | **Bytes** | **R/W** | **Description** |
| **010000** | **65536** | **256** | R/W | RS232 port 0 buffer used by interrupt handler |
| **010100** | **65792** | **256** | R/W | RS232 port 1 buffer used by interrupt handler |
| **010200** | **66048** | **256** | R/W | PS2 buffer used by interrupt handler |
| **010300** | **66304** | **256** | R/W | FORTH input message buffer |
| **010400** | **66560** | **256** | R/W | FORTH parse buffer |
| **010500** | **66816** | **256** | R/W | FORTH PAD (low area) |
| **010600** | **67072** | **256** | R/W | FORTH PAD (high area) |
| **010700** | **67328** | **19200** | R/W | Text/character graphics frame buffer |
| **015200** | **86528** | **307200** | R/W | Pixel graphics frame buffer |
| **060200** | **393728** | **65536** | R/W | FORTH SD-card file buffer |
| **070200** | **459264** | **16317951** | R/W | Additional data memory allocated by FORTH BUFFER: |

# 9. Statistics and measurements

## 9.1 FPGA overall utilization

|  |  |  |  |
| --- | --- | --- | --- |
| **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% |

## 9.2 FPGA module level utilization

|  |  |
| --- | --- |
| **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 |

## 9.3 FPGA timing summary

|  |  |
| --- | --- |
| 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 |

## 9.4 Instruction frequency

The 80% most used CPU instructions in the FORTH system software (as counted by the cross-assembler and ignoring loops and conditional code, etc.) are as listed below.

|  |  |
| --- | --- |
| **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%

The results illustrate the load-store architecture (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).

# 10. Design decisions

This section attempts to communicate the rational for some of the major design decisions in the N.I.G.E. Machine by sharing some of the original design notes as they were made at the time. To provide a clearer view of the design process, the notes have not been adjusted in retrospect.

**Memory access: Pipeline program memory in SRAM only. Data in SRAM and S(D)RAM**

|  |  |
| --- | --- |
| **Equivalent access to SRAM and (S)DRAM through DMA controller** | **Pipeline program memory in SRAM only. Data in SRAM and S(D)RAM** |
| CPU access all RAM through a standard DMA control interface with handshaking | Pipeline scheme takes advantage of guaranteed 1 cycle access in SRAM. Flexible data access |
| Advantages:   * All memory can be used for program memory. Largest program memory space * Reliable design does not need to switch modes for different memory types * No need to differentiate RAM type on data access | Advantages:   * Fastest possible one cycle per instruction execution * Faster 16 or 32 bit (S)DRAM access for data * Further optimizations are likely with independent program and data memory access in SRAM * Reliable design does not need to switch modes for different memory types * Flexible data memory access likely best suited to FORTH implementation |
| Implementation problems:   * CPU needs to implement request/ready memory hand shaking throughout | Implementation problems:   * Need separate FETCH and STORE execution for SRAM and (S)DRAM   + Three sets: SRAM (8 bit), SDRAM(8bit) and SDRAM(16bit), later DRAM(32 bit)   + SDRAM states utilize handshake   + Use common instructions for all cases and split logic flow based on memory location   + Cannot “inc” on first cycle for DRAM, but include in microcode and override this |
|  | |
| **Pipeline access SRAM and DMA access (S)DRAM** | **Pipeline program memory in SRAM only. Data in S(D)RAM** |
| CPU access automatically switches between pipeline access in SRAM and DMA access in (S)DRAM | Full Harvard architecture. |
| Advantages:   * Best of all worlds | Advantages:   * Fastest possible one cycle per instruction execution * Faster 16 or 32 bit (S)DRAM access for data * Further optimizations are likely with independent program and data memory throughout * Reliable design does not need to switch modes for different memory types * No need to implement different FETCH and STORE instructions |
| Implementation problems:   * May not ultimately be feasible since the state machine of the control unit has different requirements in pipeline and handshake modes (PC +1 vs. PC at any given instruction) * Ensuring reliability when crossing memory boundaries likely to be tricky | Implementation problems:   * Pure Harvard architecture cannot support a FORTH interpreter/complier since these need to interrogate as well as execute the FORTH dictionary |

**Interrupt processing: Sequential**

|  |  |
| --- | --- |
| **Sequential** | **Nested hierarchical** |
| Advantages:   * Straightforward logic and programming * May be better to divert LE resources from a complex interrupt unit to bespoke system peripherals on a FPGA core | Advantages:   * Most sophisticated and flexible * Fastest interrupt response times |
| Implementation problems:   * Simple priority encoder * Two state FSM and register to store pending interrupt requests | Implementation problems:   * Supervisor stack required to hold nested interrupt information |

**Cell width: 32 bit**

|  |  |
| --- | --- |
| **32 bit** | **16 bit** |
| Advantages:   * Compatible with main memory address bus width * Faster 32 bit computations * Reduced usage of 2DUP, etc. so can implement these in software | Advantages:   * Compatible with main memory data bus width * Smaller overall FORTH size * Lower LE cost |
| Implementation problems:   * 32 bit fetch and store will require microcode steps | Implementation problems:   * Memory paging will be required |

**Stack implementation:** **small register set plus dedicated dual ported RAM block**

|  |  |
| --- | --- |
| **Large register set** | **Small resister set plus dedicated dual ported RAM block** |
|  | TOS, NOS held in registers, remainder of stack in a dedicated 2K RAM block. Pointer to 3OS (always available) incremented from element zero. Similar for return stack decrementing from top element |
| Advantages:   * Likely most simple implementation of stack manipulation operations * Slightly lower RAM usage | Advantages:   * Much lower LE count * Elegant solution with block RAM |
| Implementation problems:   * On demand memory push/pull system required | Implementation problems:   * Scheme need to prevent stack over/under flow |

**CPU: microcode**

|  |  |
| --- | --- |
| **Hardwired** | **Microcode** |
| Advantages:   * Fastest possible CPU clock speed * Instructions execute in one clock cycle | Advantages:   * Most flexible for implementing as many instructions in hardware as possible * Maybe only feasible way to deal with hardware divide * Smaller instructions and variable length encoding reduce slow memory traffic |
| Implementation problems:   * FSM may be needed anyway for memory access | Implementation problems:   * Will need to include a decoder in the CPU |

**Instruction format: variable 8 – 32 bit**

|  |  |
| --- | --- |
| **Fixed 16 bit** | **Variable 8 – 32 bit** |
| Advantages:   * ‘RISC’ like design smaller and faster at CPU level | Advantages:   * Optimum for code size * Faster at memory access level for SDRAM * Likely implement more FORTH instructions in hardware |
| Implementation problems:   * Direct loads likely need to be byte-at-a-time * Branches may be limited to 14 or 15 bits * JSR (32 bit) will have to be implemented via a previous direct load to the TOS | Implementation problems:   * Variable advancement of program counter |

**Memory scheme: contiguous SRAM and SDRAM**

|  |  |
| --- | --- |
| **Contiguous SRAM and SDRAM** | **SRAM as cache for SDRAM** |
| Arrange address lines so that SRAM appears in the low page of total 16MB memory space. Expect approx 54KB available. Use Port A for CPU and Port B for char RAM, IO DMA, sprite access. (MemSet and other custom chips will be confined to SDRAM using a 2K buffer for burst reads and writes) |  |
| Advantages:   * Fastest possible execution of core code * Can use .COE file for initialization of soft ROM * Minimal LE requirements | Advantages   * Vanilla interface to all RAM from CPU * May have better overall execution speed |
| Implementation problems:   * CPU address logic will need to separate SRAM and SDRAM access in different FSM’s | Implementation problems:   * Will require substantial LE for cache controller * Cache controller likely complex |

# 10. References and acknowledgements

1. **J1: a small Forth CPU Core** **for FPGAs** by James Bowman

The N.I.G.E. Machine instruction set encoding; especially the format for branches and compound RTS instructions is directly inspired by the J1.

1. **Circuit Design and Simulation with VHDL** by Pedroni, 2nd Edition

Concise and very practical guide for learning VHDL

1. **Digital Logic and Microprocessor Design with VHDL** by Enoch O. Hwang

Thorough guide to digital logic and the principals of microprocessors

1. **Stack Computers: the new wave** by Philip J. Koopman, <http://www.ece.cmu.edu/~koopman/>

Explanation of the theory and practice of stack computers

1. **Starting FORTH** by Leo Brodie, <http://www.forth.com/starting-forth>

Standard reference for learning FORTH

1. **Forth Programmer's Handbook** by Elizabeth D. Rather and Edward K. Conklin

Standard textbook on ANSI FORTH

1. **pForth** by Phil Burk, <http://www.softsynth.com/pforth/>

Public domain FORTH implementation that greatly assisted the development of N.I.G.E. Machine FORTH

1. **Programming FORTH** by Stephen Pelc  
   An introduction to modern FORTH systems
2. **Practical and theoretical aspects of FORTH software development** by Peter J. Knaggs

PhD thesis looking in particular at FORTH optimizations

1. **Natmai team**, Gunnar von Boehn and Jens Künzer

Serious experts in softcore design and VHDL who helped me lean how to develop in VHDL