Hardware multitasking within a softcore CPU

Ulrich Hoffmann (FH Wedel University of Applied Sciences), Andrew Read

June 2015

uh@fh-wedel.de, andrew81244@outlook.com

Abstract

We have developed and implemented hardware multitasking support for a softcore CPU. The N.I.G.E. Machine's softcore CPU is an FPGA-based 32 bit stack machine optimized for running the FORTH programming language. The virtualization model that we have developed provides at least 32 independent CPU virtual machines within a single hardware instance. A full task switch takes place in only two clock cycles, the same duration as a branch or jump instruction. We have use the facility to provide a multitasking platform within the N.I.G.E. Machine's FORTH environment. Both cooperative multitasking, by means of the PAUSE instruction, and pre-emptive multitasking are supported.

1 Introduction

The N.I.G.E. Machine is a complete microcomputer system implemented on an FPGA development board [1]. It comprises a 32 bit softcore processor optimized for the FORTH programming language, a set of peripheral hardware modules, and FORTH system software (figure 1). The N.I.G.E. Machine was presented at EuroFORTH in 2012, 2013 and 2014 [2, 3, 4]. The N.I.G.E. Machine follows in the footsteps of a number of significant FORTH processors [6, 7, 8, 9, 10, 11], most especially the J-1 [6]. The N.I.G.E. Machine design files are are freely available with an open source license [5].

Most embedded systems, including those that control scientific instruments (such as the Open Network Forth system that controls the Munich particle accelerator [25]), require some level of multitasking. In this paper we explain how we have implemented multitasking in a novel manner on the N.I.G.E. Machine at the hardware level.

The development of the N.I.G.E. Machine has followed a path of utilizing FPGA hardware to enhance the performance and features of a softcore CPU. The first version of the N.I.G.E. Machine, presented at EuroFORTH 2012 [2], demonstrated the integration of a softcore CPU with a full set of peripheral modules (VGA adapter, DMA controller, I/O ports) within the same FPGA to create a standalone microcomputer system intended for the rapid prototyping of experimental scientific apparatus. The second version, presented at EuroFORTH 2013 [3], added a custom memory controller to facilitate faster and more flexible access to FPGA system memory. The third version, presented at EuroFORTH 2014 [4], introduced a sophisticated hardware return stack to allow the FORTH exception handling constructs, CATCH and THROW, to be implemented within the CPU as atomic machine language instructions. With the same philosophy in mind, the fourth version of the N.I.G.E. Machine described in this paper, adds the facility of hardware multitasking with resulting benefits for performance and reliability.

After a short overview the paper begins with a review of prior work that looks at the history of hardware support for concurrency, the history of multitasking FORTH systems and considers some notable examples of hardware designs used to assist multitasking. The following section sets the terms of reference for implementing a hardware multitasking scheme by noting the requirements for multitasking on a FORTH system in general, the requirements for the multitasking of a stack machine, and specific additional requirements that are applicable to the N.I.G.E. Machine. After that, the implementation of hardware multitasking on the N.I.G.E. Machine is described in detail at both the hardware and software levels, along with a description of how the N.I.G.E. Machine's multitasking functionality can be accessed by user applications. Finally there is a brief discussion of the advantages and limitations of our design and implementation.

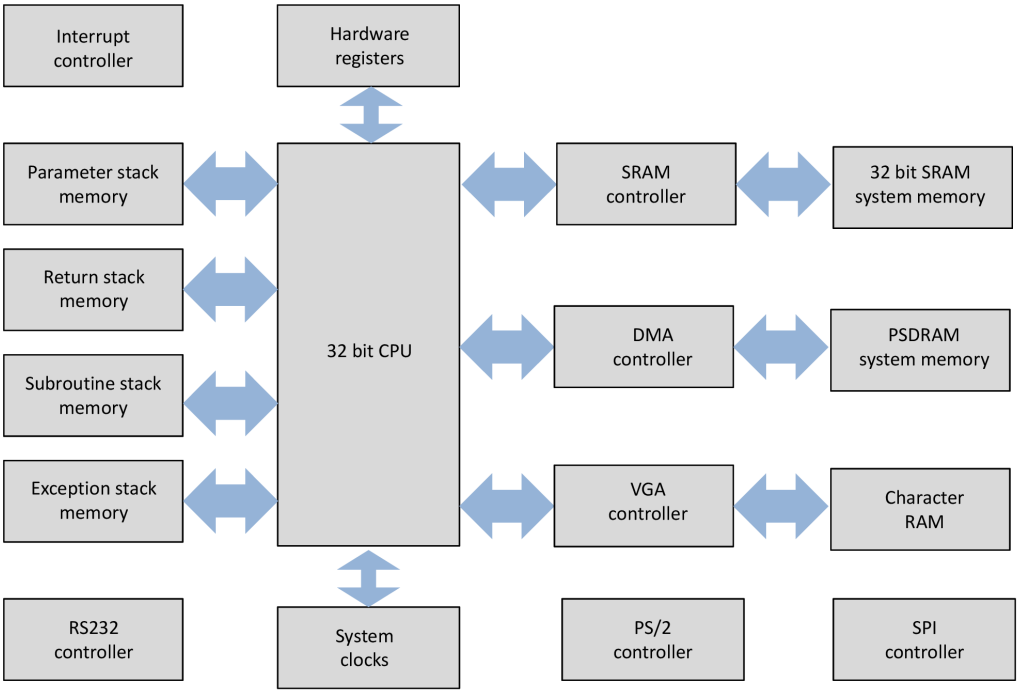

Figure 1: System diagram of the N.I.G.E. Machine

2 Outline of our model for hardware multitasking on the N.I.G.E. Machine

Our design provides 32 separate tasks hosted within a single softcore CPU instance. At any given point in time one task will be executing live on the CPU. The complete state of the other 31 tasks

that are not executing, including their program counters, stack pointers, and various registers are stored within a new sub-unit of the CPU, the virtualization unit. Because the multitasking unit is fabricated from FPGA logic within the CPU, a task switch (i.e. transfer of execution from one task to another) can be actioned with a single machine language instruction that executes very quickly, in two clock cycles in fact, the same duration as a branch or jump instruction. A second consequence of the fact that task switches are conducted entirely in hardware is that they are entirely atomic from the perspective of the flow of program code.

In addition, our design includes a lightweight task monitor that leverages the CPU hardware multitasking facility to provide multitasking capability at the FORTH software level.

3 Review of prior work

The IBM VM/370 as described by Love Seawright and Richard MacKinnon [12] offered hardware concurrency support in the form of virtualization. Rather than provide multitasking to software applications via additional functionality within the operating system, Seawright and MacKinnon's insight was to provide each application (or each operating system) with a virtual machine that was an exact copy of the underlying hardware. The software layer that provided these hardware replicas became known as the Virtual Machine Monitor (VMM).

At a lighter level than full virtualization are processor architectures that provide hardware support for context switching, but without the full resource isolation of virtualization. They are generally referred to as multithreading architectures of various types.

Barrel processors are processors that switch between n threads of execution on every cycle, thus guaranteeing that each processor will execute one instruction every n cycles. This has advantages for real-time threads operating with precise timing. The CDC 6000 range of supercomputers were pioneers of this architecture [29]. Barrel processors are an example of interleaved multithreading architectures.

Other processors such as the ARM [20] have multiple register banks to allow quick context switching for interrupt processing. Multiple register bank designs are examples of block multithreading architectures.

Another example of a block multithreading architecture is the Microcode Level Timeslicing architecture [28]. In this architecture CPU context information is held in hardware for a fixed number of tasks. Context switching overhead is eliminated since a task switch requires only the appropriate manipulation of select lines. A "stream control unit" performs the select line manipulation and coordinates context switching for the prefetch and execution units of the CPU according to the availability or otherwise of valid instructions in the prefetch registers.

Hardware multitasking support focused specifically on the efficient handling of exceptions has been tackled by two notable systems.

Klaus Schleisiek's microcore includes a hardware mechanism to support multitasking and is specifically focused on the problem of dealing with busy resources [7]. The microcore EXCEPTION mechanism allows routines to access external resources without having to query status bits to ascertain their availability/ readiness. This greatly simplifies the software needed for serial channels communicating with external devices or processes. It works as follows: when the processor intends to access a resource, the resource may not be ready yet. In such an event, the resource can assert the EXCEPTION signal before the end of the current instruction execution cycle. This disables storing the next processor state into any register except for the instruction register, which loads a special "exc" instruction instead of the next instruction from program memory. In the next processor cycle, exc will be executed calling the Exception Service Routine (ESR) at its fixed address. The ESR address will typically hold a branch to code that performs an operating system dependent task switch.

The INMOS Transputer [17], employed a rendezvous communication mechanism on external I/O ports that was used to perform a task switch entirely in hardware.

An alternative to virtualization or hardware multithreading is a multi-core processor architecture. In field of embedded design we note that most ARM Cortex [20] processors are now dual core or quad core. In addition the Parallax Propeller [21] is a low cost micro-controller with eight 32 bit cores that has a simple tool chain making it attractive for prototyping applications. Finally, the GreenArrays GA144 is a more specialized system with 144 polyFORTH execution units on a single chip [22]. Both the Transputer and the GA144 feature high speed connections between cores.

4 Hardware multitasking requirements

4.1 Requirements for a multitasking system in FORTH

The ultimate purpose of implementing hardware multitasking on the N.I.G.E. Machine is to provide a multitasking FORTH system. Brad Rodriguez's 1992 article, "FORTH Multitasking in a Nutshell" [15], provides a comprehensive review of the requirements. These are, in terms of memory: private stacks, private user areas and private buffers, and in terms of software: re-entrant FORTH system code, suitable mechanisms for the mutual exclusion of resources that cannot be shared, and suitably designed task switching functionality.

Many FORTH systems offer cooperative multitasking (where a call to the word PAUSE is required to yield the CPU to the next task) in preference to pre-emptive multitasking (where the CPU is automatically time sliced between tasks). In an embedded environment where all tasks are part of a single integrated system pre-emptive multitasking may not be necessary. In these cases cooperative multitasking may have some advantages for simplicity of design and testing, provided that all tasks truly cooperate.

4.2 Requirements for the multitasking of a stack machine

The general requirements for multitasking of a CPU are the ability to (a) switch the execution of the CPU from one task to another, (b) store the state of the task that is being "frozen" (i.e. preserve the "state vectors") and (c) restore the state of the task that is being "thawed". This requirement can be applied to a stack machine where the CPU state vector will in general comprise three elements (1) the program counter, (2) the stack pointers and (3) the stack memory space. (If stack memory space is global to all virtual machines then it is sufficient to switch only the stack pointers.)

In addition, a stack machine may utilize a number of registers. For example, top of stack values may be held in registers to enhance processing throughput, there may be flags such as arithmetic carry/overflow, or an interrupt processing indicator, and the internal state of the CPU is likely to be a finite state machine (FSM) with its own state register. For each of these registers a decision needs to be made as to whether (a) it will be included in the saved CPU state vector, (b) it will be discarded on each virtual machine switch, or (c) whether virtual machine switching will be arranged so that it is not necessary to save the value (e.g. switching can only occur when the FSM is in a single, predetermined state).

4.3 Specific requirements relating to the design of the N.I.G.E. Machine

The parameter and return stacks of the N.I.G.E. Machine are connected directly to the datapath through dedicated memory ports rather than being accessed via the general CPU system memory bus. This design leads to performance advantages because the datapath is always in a position to

read or write from stacks with no latency. However it also means that a "software only" multitasking implementation is not feasible on the N.I.G.E. Machine. Modifications must be made to the datapath itself in order to facilitate the switching of stacks for each task.

In addition to the parameter and return stacks, the N.I.G.E. Machine datapath utilizes internal subroutine and exception stacks that provide hardware support for subroutine calls, local variables and the FORTH exception handling words CATCH and THROW [4]. Although these stacks are not directly accessible to user applications, it is necessary for each task to have a private copies.

Lastly, the N.I.G.E. Machine has been designed specifically for embedded control and scientific prototyping. As EuroFORTH 2012 paper explained [2], short interrupt response times and deterministic execution are critical in these applications. Any hardware multitasking scheme employed needs to respect both of these objectives.

5 Hardware multitasking design on the N.I.G.E. Machine

5.1 General features

As explained in the introduction, the purpose of incorporating hardware multitasking into the N.I.G.E. Machine is to provide multitasking for the FORTH system software. (Since each task at the FORTH system level will run on its own virtual machine instance.)

In the default configuration of the N.I.G.E. Machine, 32 tasks are available, each with a parameter stack depth of 256 cells and a return stack depth of 128 cells. Cooperative multitasking is achieved with a PAUSE machine language instruction which executes a full task switch in 2 clock cycles (the same duration as the execution of a branch or subroutine call). A pre-emptive multitasking mode is also available that implements a task switch after a user definable count of executed instructions. The default task scheduling system is round-robin among active tasks.

There is a lightweight task monitor included in the FORTH system software that comprises words for starting, stopping, and otherwise managing tasks. The task monitor is not involved with task switching since this is handled entirely by hardware. Each task has a 2 KiB private user memory area that offers space for 245 longword user variables and holds private buffers and certain task specific system variables. The N.I.G.E. Machine's remaining 128 KiB of FPGA systems memory (including the FORTH dictionary) and all of the 16 MiB of PSDRAM is shared memory available to all tasks.

Simple semaphore based locks have been implemented in the FORTH system software to mediate access to shared I/O resources. The locks are arranged so that no FORTH system routine will ever attempt to lock more than one resource at any given time. In this way it is not possible for user applications to enter a deadlock situation if they only call system functions.

The N.I.G.E. Machine also provides a feature that we describe as "virtual interrupts". With a virtual interrupt, a task may cause another task to jump to a subroutine (typically a FORTH execution token) before returning to its prior point of execution. A virtual interrupt may be scheduled in advance at any time and will be actioned when a switch to the task in question next occurs.

5.2 Multitasking unit

The multitasking unit is a new component within the CPU alongside the control unit and the datapath (figure 2). The multitasking unit is responsible for two functions: firstly for storing the states of all of the tasks that are not currently executing, and secondly for managing the transition between executing tasks.

To achieve the first objective the multitasking unit relies on an internal RAM module termed the "freezer RAM". The freezer RAM module is 116 bits wide and 32 addresses deep. (The RAM modules within the multitasking unit are implemented with FPGA logic elements (registers) rather than BLOCK RAM - see section 6 for further explanation).

The state of each task is arranged as a 116 bit state vector as illustrated in table 1.

Task state vector component	**Number of bits**
Program counter	20
Top-of-stack register	32
Next-on-stack register	32
Parameter stack pointer	8
Return stack pointer	8
Subroutine stack pointer	8
Exception stack pointer	8
Total	116

Table 1: Storage of the task state vectors within the "freezer" RAM module

The freezer RAM module is dual ported with a single write port and a single read port. When a task switch is signaled, the state vector of the current (and now retiring) task is presented at the write port with a valid write enable signal, while the state vector of the next-to-execute task is taken from the read port. These state vectors are routed to and from the control unit (for the program counter) and the datapath (for all other elements). Figure 2 illustrates the place of the multitasking unit in relation to the control unit and datapath within the CPU.

The address input of the freezer RAM module's write port comes from a 5 bit wide register within the multitasking unit that holds the number of the currently executing task. The address input for the read port (i.e. the number of the next-to-execute task) is drawn from a second RAM module within the multitasking unit called the task control RAM. The signal to execute a task switch originates from the control unit (described more fully below).

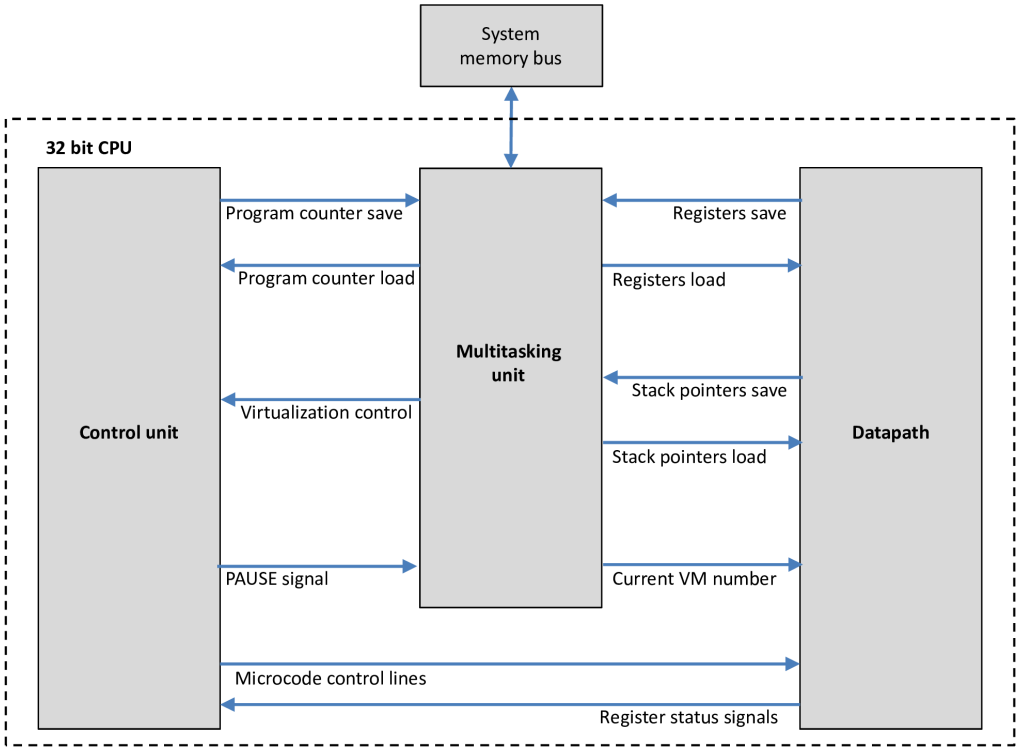

Figure 2: Relationship between the multitasking unit, control unit and datapath with the CPU

The task control RAM module is 16 bits wide and 32 addresses deep. It can be considered as a set of 32 x 16 bit registers, one belonging to each task. The lower 5 bits of each resister hold the number of the next-to-execute task following that task. For example if register 1 holds the value "00010", then task 2 is the next-to-execute task following task 1. The task control RAM module is dual ported. The first port is read-only and is addressed with the register holding the value of the currently executing task. The output from this port is therefore the number of the next-to-execute task (in the lowest 5 bits). It is used to address the read port of the freezer RAM as described above. The second port is a read/write port that is memory mapped to the system memory address space. The task monitor within the FORTH system software uses these memory mapped addresses to configure the order of task execution by appropriately setting the individual task control registers. The upper 11 bits of each task control register are not read by the multitasking hardware but are used by the task monitor as general storage for further task control purposes as described below.

In order to initialize a new task the task monitor needs to be able to configure the program counter of a task before it begins execution for the first time. A third dual ported memory block, the "PC override" RAM block, is used to provide this facility. The PC override RAM module is 20 bits wide and 32 addresses deep. It can also be thought of as 32 individual registers. Like the task control RAM module, each register is mapped to the system memory address space and can be written to by the task monitor. When a task switch occurs, if the PC override register of the next-to-execute task contains a non-zero value, then that value is sent to the control unit as the new program counter address in place of whatever value may have been held in the task's state vector in freezer RAM. The PC override register for that task is then automatically reset to zero

so that the following task switch will proceed without a second override.

The fourth RAM block (the "virtual interrupt" RAM block) provides functionality for virtual interrupts in a similar manner to the PC override RAM. However in the case of a task switch with a virtual interrupt, the control unit pushes the saved value of the PC from the state vector onto the subroutine and return stacks before setting the program counter register to the value from the virtual interrupt RAM.

The multitasking unit also contains a number of individual memory mapped registers that allow the task monitor to control the conduct of task switching. These are scheduled in table 2. Table 3 schedules the overall memory map of the multitasking unit as seen from the system memory bus.

Figure 3 illustrates the key operational features of the of the multitasking unit.

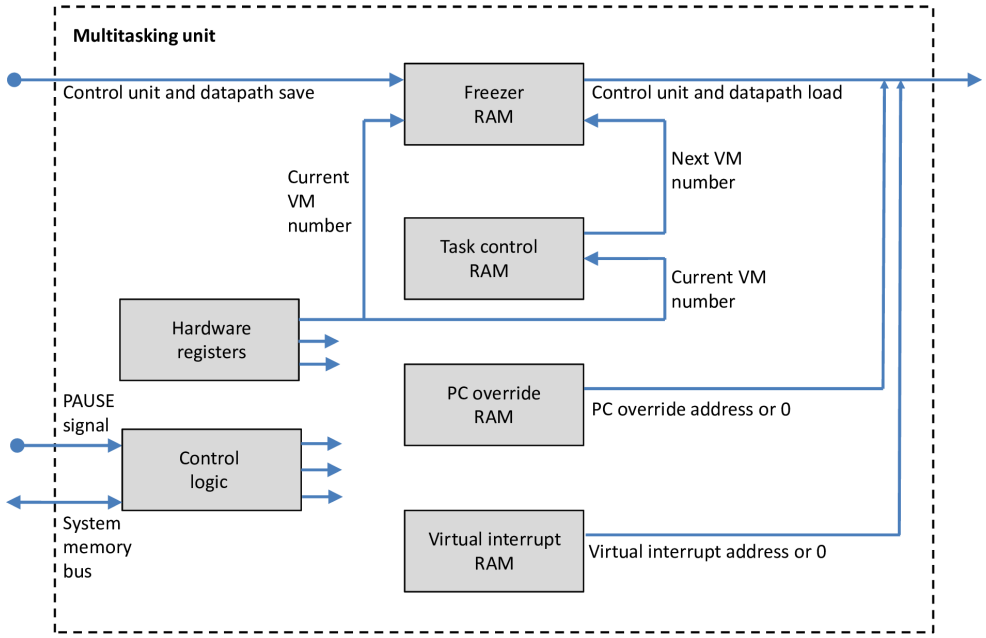

Figure 3: Illustration of the key operational features of the multitasking unit

Register name	Function	R/W	Width (bits)	System address
SINGLEMULTI	Enable ('1') or disable ('0') multitasking. If a PAUSE machine language instruction is encountered with multitasking disabled then it will be treated as a NOP	R/W	1	3F000
CURRENTVM	The number of the currently executing task. Tasks are numbered 0 through 31. At power-on task 0 will be executing the FORTH system software	R	5	3F004
INTERVAL	The interval for pre-emptive multitasking task switches, in count of instructions. If INTERVAL = 0 then pre-emptive multitasking is disabled.	R/W	16	3F008

Table 2: Multitasking control registers

Register name	Function	# registers	System address
Multitasking control registers	As table 2	3	3F000
Task control registers	Each virtual machine has an associated task control. Bits 4 down to 0 of this register specify the next-to-execute task. Bits 15 down to 5 are for task monitor usage	32	3F200
PC override registers	Writing a non-zero address to a PC override register will cause the task in question to continue from that address on the next occasion that it executes	32	3F400
Virtual interrupt register	Writing a non-zero address to a virtual interrupt register will cause the task to branch to a subroutine at that address on the next occasion that it executes	32	3F600

Table 3: Memory map of the multitasking unit as viewed from the system memory bus

5.3 Softcore CPU - datapath

Two updates were necessary to the CPU datapath in order to support hardware multitasking. Firstly, the parameter, return, subroutine and exception stacks were extended so that each task would have its own private stack. This was achieved by increasing the addressable width of each stack and allocating to each stack additional FPGA BLOCK RAM. Following this modification, each stack is addressed within the datapath by concatenating the number of the current virtual machine (higher 5 bits) with the current stack pointer (lower 8 bits).

Secondly, the values of the top-of-stack and next-on-stack registers and the values of the stack pointers were interfaced with the freezer RAM within the multitasking control unit. This was done simply by including additional multiplexers and extending the width of microcode instructions communicated from the control unit to 23 bits. Further details on the operation of the datapath and how the control unit uses microcode to choreograph the datapath multiplexers is given in the

EuroFORTH 2012 paper [2]. By way of illustration, table 4 shows how the exception stack pointer is updated each clock cycle according to the microcode signaled from the control unit.

Function	**Microcode bits 22 down to 20**
No change	000
Decrement	001
Increment	010
Set to zero	011
Load value from virtualization unit	100

Table 4: Control table for the exception stack pointer illustrating the extension for hardware multitasking. The exception stack pointer is a datapath register that is operated from the control unit by microcode bits 22 down to 20

5.4 Softcore CPU - control unit

The first modification made to the control unit was to specify a new PAUSE instruction to effect a cooperative multitasking task switch. No major "rewiring" of the control unit was required to accomplish this: the PAUSE instruction acts on the same level and in the same way as all machine language instructions within the control unit's finite state machine via microcode lookup. In fact the PAUSE instruction was implemented as a modified jump (JMP) instruction, but with the appropriate microcode to control the datapath registers, and with the next-instruction address read from the multitasking unit rather than from the parameter stack. It is because a task switch can be executed as a standard machine language instruction that the latency for a task switch is the same as for any other jump or branch (2 clock cycles). A task switch involving a virtual interrupt involves an additional stage to push the value of the program counter onto the subroutine stack and therefore executes in 3 clock cycles.

Secondly, a 16 bit counter was introduced to count the number of instructions executed since the last task switch. This counter is compared with the INTERVAL register of the multitasking unit and a pre-emptive task switch is triggered when the INTERVAL count has been reached or exceeded, if preemptive multitasking is enabled.

The control unit uses a common mechanism to handle a PAUSE instruction and a preemptive task switch. One important difference however is the value of the program counter that is saved to the multitasking unit. In the case of encountering a PAUSE instruction, the task should resume execution at the instruction following this PAUSE. In the case of a preemptive task switch, then the current instruction (whatever it is) will not be executed since the preemptive task switch has priority. In this case the task should resume execution at this instruction. This differentiation is similar to how the control unit selects the appropriate value of the program counter to push onto the subroutine stack in the cases of jump to subroutine (JSR) instructions and interrupts, and identical hardware logic was used.

5.5 Interrupts

With any multitasking design a decision is needed as to how interrupts will interact with task switching. There are two broad alternatives: either interrupts are synchronized with task switches so that interrupt handlers always run within their own tasks, or interrupts are handled by whichever task happens to be running at the time when the interrupt request occurs.

We did not examine the trade-off between these two approaches in great detail. The N.I.G.E. Machine takes the latter approach. For us the simplicity of design thus afforded and the avoidance

of any possible latency in interrupt responses were sufficiently compelling advantages in the absence of any obvious considerations to the contrary.

Given this design decision, the N.I.G.E. Machine system interrupts for RS232 and PS/2 I/O execute in whichever task is running when the interrupt request occurs. The RS232 and PS/2 interrupt service routines operate by transferring characters between the relevant hardware interface and memory buffers, updating the buffer counters accordingly. Tasks that need to wait for RS232 or PS/2 communications do so by polling for updates to the relevant buffer counters in a loop that includes a PAUSE statement. The FORTH word KEY? is implemented in this manner.

In order to avoid any performance or reliability impact in interrupt handling, it is necessary to ensure that interrupts cannot themselves be interrupted by task switches. Pre-emptive multitasking is automatically disabled by the control unit during interrupt processing. If the preemption instruction counter reaches INTERVAL while an interrupt is in progress then the task switch is postponed until immediately after the interrupt service routine has concluded. For cooperative multitasking, it is a N.I.G.E. Machine software design requirement that PAUSE machine language instructions should not be included within interrupt service routines.

5.6 Task monitor software

The task monitor software is a set of words within the FORTH system software available to initiate and control tasks. A description of some of the words is presented in section 7.

The task monitor's method to control the default sequencing and allocation of tasks is as follows: the lowest 5 bits of each task control register (there is one task control register for each task) indicate the number of the next-to-execute task. These 5 bits are utilized directly within the multitasking unit to sequence a task switch as described above. The remaining 11 bits do not directly control hardware but are utilized by the task monitor. Bit 15 is used to indicate whether a task has been allocated to a running task ('1') or is unassigned and therefore available for a new task upon request ('0'). Bits 9 down to 5 are used by the task monitor to indicate the number of the task that points execution to this task. In this way the lower 10 bits form the nodes of a doubly linked list that specifies the task execution order. Bits 14 down to 10 are reserved for expansion.

When a new task is requested (see RUN in section 7) the task monitor first searches the set of task control register to identify an unassigned task (indicated if bit 15 is clear). The new task is then inserted into the doubly linked list of executing tasks after the currently executing task by updating the nodes of the double linked list maintained within the lower 10 bits of the task control registers. For all newly initialized tasks, the program counter for the new task is directed to a common initialization routine. This brief (~60 byte) initialization routine is responsible for resetting the stack pointers of that task to zero, copying the set of initialization parameters specified by the RUN command to the stack via. intermediate storage in shared memory, initializing the user variable area, initializing the exception stack variables and then jumping to the specified execution token.

When a task is put to sleep (see SLEEP in section 7), it is removed from the doubly linked list of executing tasks but bit 15 of the task control register remains set to indicate that the task is still allocated. When a task is woken (see WAKE in section 7) it is re-inserted into the doubly linked list of executing tasks. When a task is stopped, then in addition to removing it from the list of currently executing tasks, bit 15 is cleared to indicate that it is now free for reallocation.

5.7 Controls over task switching

The multitasking unit operates two controls to limit task switching and thus avoid glitches:

Firstly, because the state-vector of the next-to-execute task is automatically transferred from the RAM blocks within the multitasking unit into the control unit and datapath when a task switch

occurs, it is necessary for these RAM blocks to have time to update properly between task switches. The minimum interval between task switches is 3 clock cycles due to this update requirement. A control device within the multitasking unit imposes a 5 clock cycle minimum interval between task switches. If a PAUSE instruction or pre-emptive task switch occurs within this limit then it will simply be ignored. This is a critical control since cooperative PAUSE instructions continue to remain effective even after pre-emptive multitasking has been enabled.

Secondly, multitasking is automatically disabled by the multitasking unit when there is only one active task (i.e. where the lower 5 bits of the currently active task's task control register references itself). This is necessary on account of the same update constraint.

5.8 User memory area

Each task has a private 2 KiB user memory area that is mapped to the system address space. The user memory areas are hosted within 64 KiB of FPGA BLOCK RAM. The upper 5 address bits are linked directly to the register that holds the number of the currently executing task. In this way each of the 32 user memory areas can be mapped to the same address range in the system address space. The currently executing task will always have guaranteed private assess to its own user memory area but no access to the user memory areas of other tasks. Table 5 is an outline memory map of the user memory area. Further details are given in the N.I.G.E. Machine documentation.

Usage	**# bytes**	**System address**
FORTH system task specific variables	44	3D000
Available for USER variables	980	3D02C
The FORTH PAD buffer	512	3D400
The FORTH ACCEPT buffer	256	3D600
Reserved for expansion	256	3D700

Table 5: Outline memory map of the 2 KiB user memory area

5.9 Memory management

We took the decision not to implement any form of memory management over the 128 KiB of system memory that holds the FORTH system dictionary and user applications, or over the 16 MiB of off-chip PSDRAM that holds the screen buffer and is available for application data storage. As a result, all of the system memory and PSDRAM is available to any task without restriction. The discussion in section 8 considers the merits and limitations of this approach.

5.10 Motivation for the design decisions

The principal motivations for our design decisions are discussed in section 8 by way of comparison to alternative concurrency strategies.

A hardware multithreading approach offers some advantages, as described in this paper, but necessarily also entails some fixed allocation of resources at design time. Our resource allocations were based on "educated guesses" rather than specific research, but this is a softcore design and flexibility is retained since the allocations can quite easily be adapted for individual builds, often simply by changing VHDL GENERIC declarations.

6 FPGA implementation

The N.I.G.E. Machine is implemented on a Digilent Nexys4 development board [18] which features a Xilinx Artix-7 FPGA (Xilinx part number XC7A100T-1CSG324C [19]). The design has been developed using the VHDL hardware description language and the Xilinx ISE development studio, version 14.6. Table 6 shows the FPGA utilization for the fully synthesized design. Table 7 analyzes the usage of BLOCK RAM.

FPGA resource	**Utilization**
Slice registers	4%
Slice look up tables (LUT's)	9%
FPGA BLOCK RAM	97%

Table 6: FPGA utilization

Design component	**BLOCK RAM count**
System memory	32.0
Task private user memory	16.0
Parameter stacks	7.5
Return stacks	4.0
Subroutine stacks (including space for local variable storage)[4]	68.0
Exception stacks [4]	5.0
VGA display interface	1.0
CPU microcode	0.5

Table 7: FPGA BLOCK RAM usage by design component. Each BLOCK RAM resource represents 4Kbytes.

The Xilinx XC7A100T is a latest generation FPGA in the Xilinx "value" range. The N.I.G.E. Machine utilizes less than 10% of the fabric logic on this device. On the other hand the BLOCK RAM is significantly utilized at 97%. The simple reason for the high utilization of BLOCK RAM is that the private stacks and user memory areas for all of the 32 tasks are pre-dedicated at synthesis time regardless of how many active tasks any given application will actually create. However it is also possible to synthesis the design with 16, 8, 4, or 2 virtual machines instead of 32 by adjusting the top-level VHDL GENERIC declarations and the ipCORE declarations of the relevant RAM blocks.

Not all of the RAM modules on the N.I.G.E. Machine are instantiated with BLOCK RAM. All of the RAM modules within the multitasking unit are instantiated using distributed FPGA logic elements for resource efficiency reasons. Xilinx Artix-7 BLOCK RAM units can be configured in a variety of formats between 32K x 1 and 512 x 36 (address depth x bit width), but the freezer RAM has a relatively wide but shallow format of 32 x 116 which would lead to poor utilization in BLOCK RAM.

We had concerns that the over utilization of BLOCK RAM would significantly impede place and route performance, since the design effectively requires that signals be routed to and from BLOCK RAM instances right across the FPGA. During development we found that ISE's simulated annealing placement algorithm was quite sensitive to design changes (meaning that small changes in the logic design could have significant impact on the place and route performance, presumably due to their implications for routing). ISE's SmartXplorer, which is a tool for automatically optimizing

placement using for example, different cost tables within the simulated annealing algorithm, was able to meet timing with a clock frequency of 100 MHz but significant search effort was required (8 out of 100 strategies succeeded). At a clock frequency of 95 MHz, SmartXplorer was able to meet timing with the vast majority (95 out of 100) of strategies. The N.I.G.E. Machine's 100 MHz clock frequency has been retained, but it would likely be easier to develop future projects with a clock frequency of 95 MHz and then use SmartXplorer re-optimize place and route for a clock frequency of 100 MHz as the final step.

7 N.I.G.E. Machine multitasking functionality

This section describes a selection of the FORTH words that are available to user applications to control multitasking on the N.I.G.E. Machine. A full list is give in the N.I.G.E. Machine system documentation [5]. The majority of words are directly analogous to those of the PolyFORTH multitasking system [16].

7.1 Multitasking configuration

`SINGLE ( --)`

Disable multitasking. PAUSE instructions will be treated as a NOP. Multitasking is enabled at power-on by default on the N.I.G.E. Machine.

`MULTI ( --)`

Enable multitasking. Note that if there is only a single active task then PAUSE will be treated as NOP.

7.2 Task initiation

`RUN ( p1 ... pn n XT -- TN true | false)`

Initialize a new task to take n stack parameters (p_1 ... p_n) and execute the code pointed to be execution token XT. Return the number of the task allocated to this task (TN) and true if successful, or false if all tasks are currently otherwise allocated. The newly created task will be positioned in the round-robin sequence immediately after the current task. Tasks are numbered 0 through 31. Note that the XT must either code an infinite loop or contain termination instructions to self-abort.

7.3 Task switching

`PAUSE ( --)`

Task switch. Yield CPU execution of the current task and switch CPU execution to the next-to-execute task.

`SLEEP ( n --)`

Put task n to sleep by removing it from the list of executing tasks. The task remains allocated and can be woken at a later time.

`WAKE ( n --)`

Wake task n by inserting it into the list of executing tasks immediately following the current task.

`STOP ( n --)`

Deallocate task n and remove it from the list of executing tasks. This task may now be recycled by RUN.

7.4 Pre-emptive multitasking

`PREEMPTIVE ( n --)`

Enable preemptive multitasking with period of n instructions between task switches. If n = 0 then preemptive multitasking is disabled. Preemptive multitasking is disabled at power on by default on the N.I.G.E. Machine.

7.5 Virtual interrupts

`VIRQ ( XT n --)`

Virtual interrupt. Cause task n to branch to the subroutine at XT and then return to its prior point of execution. The virtual interrupt will be actioned when task n is next scheduled to execute.

7.6 Mutual exclusion

`ACQUIRE ( sem --)`

Acquire the binary semaphore (sem) or yield until it becomes free. A semaphore can be any FORTH variable with global scope. Semaphores are minimum single byte in length (word or longword length variables may also be used). A semaphore contains the number of the latest successfully acquiring virtual machine XOR 255, or 0 if not acquired.

`RELEASE ( sem --)`

Release the binary semaphore (sem).

7.7 Inter-task communication

We have not attempted to provide hardware based support for inter-task communication. As noted above, each task's 2 KiB private memory is not accessible by other tasks but the rest of the 128 KiB system memory and all of the 16 MiB of PSDRAM on the N.I.G.E. Machine is accessible by all tasks. Application specific inter-task communication designs can utilize shared memory for data passing and may take advantage of the ACQUIRE and RELEASE words for mutual exclusion.

8 Discussion

8.1 Comparison with other hardware multithreading strategies

The N.I.G.E. Machine's approach to hardware multitasking has some similarities with the multiple register file / block multithreading architectures referenced in section 3 but as a whole is different from those strategies.

Multiple register file architectures essentially use control lines to select which instances of the CPU registers are to be updated each cycle. The N.I.G.E. Machine takes a similar approach in selecting the parameter and return stack for each task by extending the address width of each stack and concatenating the number of the currently executing thread at the high end of the address bus with the relevant stack pointer at the low end.

However the N.I.G.E. Machine does not apply this approach to the registers within the CPU and datapath (e.g. the program counter and top-of-stack register), rather they are saved and reloaded from an external store (the multitasking unit) each time a task switch occurs. The reason for doing

this is timing efficiency. The N.I.G.E. Machine operates at 100 MHz meaning that each clock cycle must complete within 10 ns if the design is to "meet timing". Each additional layer of logic in a signal path adds delay due to both the response time of the logic itself and the necessary signal routing. In terms of logic layout, a register file of 32 registers is essentially a 32 way multiplexer that must sit between say the ALU and the register to be updated. Even though modern FPGA multiplexers are highly optimized, a 32 way multiplexer would need to be implemented in 2 or 3 additional layers of FPGA logic[30]. By storing register information for each task in a separate unit, the N.I.G.E. Machine avoids the need for any extra FPGA logic on the signal path that updates each cycle. Rather the "update burden" is shifted to the cycles that occur between clock cycles, which in the case of the N.I.G.E. Machine is only 2 clock cycles in any case.

A final difference is that a task switch on the N.I.G.E. Machine switches more than the CPU context: the USER memory areas that are private to each task are switched concurrently. Hence we have termed the N.I.G.E Machine's architecture as hardware multitasking rather than hardware multithreading.

8.2 Comparison with multi-core processor strategies

In recent years the trend in processor development has been firmly towards multi-core CPU's, even in embedded applications [20]. However this trend is not without a number of difficulties imposed by the complications of multi-core software development [23]. Our focus on developing a virtualization model for the N.I.G.E. Machine has been to attempt to balance the pursuit of absolute performance with simplicity for the application programmer.

The key difference between programming a single core multitasking architecture such as the N.I.G.E. Machine as compared with a multi-core CPU is the elimination of possible asynchronous effects. The N.I.G.E. Machine used in cooperative multitasking mode will have absolutely deterministic behaviour (and timing) since there is a single execution path through all contexts. In a multi-core CPU multiple asynchronous execution paths must be modelled, programmed and debugged.

The N.I.G.E. Machine is intended for rapid prototyping applications where fast and easy software development should be a particular advantage. The certain absence of asynchronous effects is the programmer simplification motivating our preference for a single core rather than a multi-core approach.

8.3 Chosen approach to memory management

As described in section 5, aside from providing a 2 KiB private memory area, we decided not to implement any memory management mechanisms for the 128 KiB of main system memory and the 16 MiB of PSDRAM. We recognize that on a modern server or desktop based multitasking system it would be considered a fatal weakness not to provide memory protection mechanisms that prevent tasks from corrupting the memory used by other tasks. High-reliability computing is also in demand in the embedded space.

However the intended focus of the N.I.G.E. Machine is in relatively small scale deeply embedded applications. For that reason we envisage that in most situations, all tasks will be sub-modules of a single overall application and so inter-task protections may be a less critical factor in total application reliability.

Another reason for our decision is that since the overall system memory is only 128 KiB (albeit this is probably still a reasonable system memory size for a deeply embedded device [24]), it would not be feasible to subdivide this memory between tasks and still retain a sensible amount for each.

Finally, the FORTH language is dictionary based and it is typical for multitasking FORTH systems to have access to a common system dictionary. The FORTH system software on the N.I.G.E.

Machine implements the ANSI Search-Order word list which allows individual applications or tasks to extend or restrict the dictionary with private word lists if desired.

8.4 Alternative task scheduling models

The task scheduling model implemented by the task monitor is a simple round-robin scheme. That is, all active tasks take their turn to be executed once per round. This is the fastest task switching model that can be implemented using the N.I.G.E. Machine's hardware multitasking framework since the next-to-execute task is determined in advance and task switches take place atomically in only two clock cycles. Given the high performance of this scheduling model within the N.I.G.E. Machine, and it's ubiquity on multitasking FORTH systems [15], we believe that it would likely be the most effective choice for most embedded applications.

However many other priority based task scheduling models exist [13]. A priority based task scheduling model can be accommodated within the N.I.G.E. Machine's hardware multitasking framework by changing the way that task control registers are used by the task monitor. An outline of how this could be achieved is as follows: the task control registers of all tasks are set such that the next-to-execute task is always a common scheduling task (say task 31). The scheduling task would be responsible for maintaining a list of task priorities and determining the next-to-execute on a real time basis. It would conclude its operation by setting the value of its own task control register before executing PAUSE.

Paul Bennett has pointed out [26] that an alternative model for cooperative multitasking is the Time-Triggered Systems (TTS) approach [27]. TTS is typically based on just one interrupt (the system tick timer). It schedules cooperative tasks to run at intervals according to their order in the "tick list". There is no main loop of program execution aside from the tick list itself. All I/O is polled. TTS could quite likely be implemented on the N.I.G.E. Machine using this multitasking hardware with light adjustments to the task monitor software.

Although we have not further investigated in any detail, it was mentioned to us that this architecture might also be leveraged to support the high level language features of co-routines and continuations.

8.5 Limitation of the hardware multitasking approach

An obvious limitation of the N.I.G.E. Machine's approach to multitasking is that the number of tasks is limited to the 32 that are pre-instantiated in hardware. We did not conduct a feasibility study of the number of tasks typically required by an embedded system but would expect based on general experience that parallel programming complexities might become a constraint in an embedded application before the limit of 32 tasks had been reached.

Another limitation is that hardware resources (mainly FPGA RAM blocks) are pre-allocated to tasks that may never be used, in which case they are effectively wasted. With 32 tasks allocated there is 128Kbytes of system memory for FORTH applications and parameter and return stack depths of 256 and 128 cells respectively in each virtual machine. So sufficient resources are available for a meaningful FORTH system. As explained in section 6, it is possible to synthesize the N.I.G.E. Machine with 32, 16, 8, 4 or 2 tasks by adjusting the generics declarations in the VHDL code.

8.6 Advantages of the hardware multitasking approach

We suggest that there are a number of advantages to using hardware multitasking to provide a multitasking FORTH system as compared with traditional software-based multitasking.

Firstly although task switching on FORTH systems is typically very fast [15], the ability to complete a full task switch in the same duration as a jump or branch means that effectively multitasking has no performance overhead on the N.I.G.E. Machine.

One way this performance advantage could be put to use for enhancing reliability is by including PAUSE instructions within FORTH loop structure words (LOOP, +LOOP, UNTIL, AGAIN, REPEAT) [26]. Although that has not been done with the current version of the N.I.G.E. Machine's FORTH system software, the advantage of doing so would be to decrease the likelihood of tasks failing to cooperate due to their being insufficient PAUSE instructions within their routines.

Secondly, because pre-emptive multitasking is implemented directly by the CPU and not via interrupts, it is never necessary to disable interrupts during tasking switching, or even during the initiation of new tasks. This means that the N.I.G.E. Machine avoids any interrupt latency due to multitasking.

Lastly, since task switching is handled by a single machine language instruction, each task switch is atomic, i.e. the thread of execution is always with one task or another, never in-between tasks. This may have some reliability benefits since there are no task switching software routines that have the potential to become corrupted during program execution.

9 Conclusion

FORTH systems have offered multitasking since very early in the history of FORTH language and on very lightweight system [15, 16]. Now that the N.I.G.E. Machine includes multitasking capability with fast and efficient hardware support, we believe that the platform is sufficiently developed to be applied to its intended field in the rapid prototyping of experimental scientific apparatus. It is hoped that the next stage of development will focus on opportunities in this area. In addition there is the possibility to port the design to other FPGA development boards (e.g. the Diligent Nexys4-DDR) or enhance the range of input/output ports.

The authors wish to thank the anonymous academic reviewers for their comments, especially those relating to the terminology of the architecture. Their comments have significantly improved the clarity of the paper.

References

[1] Andrew Read, YouTube video demonstrations
https://www.youtube.com/channel/UCz_LqPfKT0r2rEID7Av-Chw

[2] Andrew Read, "The N.I.G.E. Machine: an FPGA based micro-computer system for prototyping experimental scientific hardware", in *EuroFORTH*, 2012

[3] Andrew Read, "Optimizing memory access design for a 32 bit FORTH processor", in *EuroFORTH*, 2013

[4] Andrew Read, "Concept and implementation of an extended return stack to enhance subroutine and exception handling in FORTH", in *EuroFORTH*, 2014

[5] Andrew Read, Github open source repository
https://github.com/Anding/N.I.G.E.-Machine

[6] James Bowman , "J1: a small Forth CPU Core for FPGAs" in *EuroFORTH*, 2010

[7] K. Schleisiek, "MicroCore," in *EuroFORTH*, 2001.

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

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

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

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

[12] L. H. Seawright, R. A. MacKinnon: “VM/370-a study of multiplicity and usefulness”, IBM Systems Journal, 1979

[13] Andrew S. Tanenbaum, Albert S. Woodhull, “Operating Systems Design and Implementation”, Prentice Hall, 2nd ed., 1997

[14] Dawson R. Engler, “The Design and Implementation of a Prototype Exokernel Operating System”, MIT, 1995

[15] Brad Rodriguez, “Forth Multitasking in a Nutshell”, The Computer Journal #58, 1992

[16] GreenArrays Inc., PolyFORTH Reference Manual, 1986-2012

[17] INMOS Limited, The Transputer Reference Manual, 1988

[18] Digilent Inc, Nexys4 FPGA Board Reference Manual, 2013-2015

[19] Xilinx, Artix-7 FPGAs datasheet, 2014

[20] ARM, ARM Cortex Portfolio, 2014

[21] Parallax Semiconductor, Propeller P8X32A datasheet, 2011

[22] GreenArrays Inc, GA144A12 chip reference, 2011

[23] David Patterson and John Hennessey, “Computer Organization and Design, Fifth Edition: The Hardware/Software Interface”, 2013

[24] Atmel AVR 8-bit and 32-bit microcontrollers datasheet, 2015

[25] Open Network Forth Control System for the Munich Accelerator Facility, Ludwig Roher, Heinz Schnitter, Egmot Woitzel, at the FORML conference, 1998

[26] Paul E. Bennett IEng MIET, private correspondence, 2015

[27] Michael J. Pont, “The Engineering of Reliable Embedded Systems”, ISBN 978-0-9930355-0-0

[28] Daniel Curtis McCrankin, “The Microcode Level Timeslicing Processor Architecture”, McMaster University, 1988

[29] Control Data Corportation “CDC Cyber 170 Computer Systems; Models 720, 730, 750, and 760; Model 176 (Level B); CPU Instruction Set; PPU Instruction Set”, 1979 - 1981

[30] Xilinx, “Multiplexer Design Techniques for Datapath Performance with Minimized Routing Resources”, 2014