Optimizing memory access design for a 32 bit FORTH processor

Andrew Read

Abstract

This paper compares and contrasts two alternative approaches to designing system memory access for a 32 bit FORTH processor. One approach maximizes clock speed, whilst the other maximizes instruction throughput. This tradeoff parallels the difference between the RISC and CISC approaches to CPU design. The project's conclusion is that a hybrid memory access design that addresses the differing needs of the CPU control unit and datapath is likely to be the optimum performance strategy for a FORTH machine.

1 Introduction

The N.I.G.E. Machine is a complete computer system implemented on an FPGA development board [1]. It comprises a 32 bit softcore processor optimized for the FORTH language, a set of peripheral hardware modules, and FORTH system software. The system is primarily designed to support the rapid prototyping of experimental scientific apparatus . The N.I.G.E. Machine was presented in a paper at EuroFORTH 2012 [2]. In the conclusions of that paper a number of avenues for further work were suggested. These included improving memory access efficiency, implementing an SD-card interface with a FAT file system, and porting the N.I.G.E. Machine to a higher performance FPGA development board.

As of the date of this paper an SD-card interface and native FAT file system have been successfully implemented and tested. This greatly simplifies the transfer of program and data files between the N.I.G.E. Machine and a PC. The upgrade was relatively straightforward. It was essentially an implementation exercise that did not raise substantial new design issues. Porting the N.I.G.E. Machine to a Digilent Atlys development board with a Spartan 6 FPGA is currently underway.

The goal of improving memory access efficiency was set with the intention of redesigning the connection between the system memory (i.e. the FPGA static RAM blocks (SRAM)) and the softcore CPU. The CPU is a 32 bit processor, but the memory databus between the CPU and SRAM is only 8 bits wide in the original N.I.G.E. Machine design (fig. 1). Widening the databus from 8 bits to 32 bits to match the width of the softcore processor might superficially seem to be a worthwhile and simple capacity increase at the cost of some further FPGA resources. However upon closer examination this design change actually raises a number of interesting implications in terms of conflicting requirements for functionality. As a result, the final preference between the 8 bit and 32 bit memory databus configurations requires deeper consideration of the intended application and the specific hardware on which the N.I.G.E. Machine will be deployed.

Using the N.I.G.E. Machine as an example, this paper explains the background and challenges in optimizing memory access for a 32 bit FORTH processor. It describes the 32 bit databus design that was successfully implemented on the N.I.G.E. Machine, presents performance benchmark comparisons between the 8 bit and 32 bit databus configurations, and discusses the tradeoff decisions guiding which implementation should ultimately be preferred. All of the N.I.G.E. Machine design files and software are available open source [11].

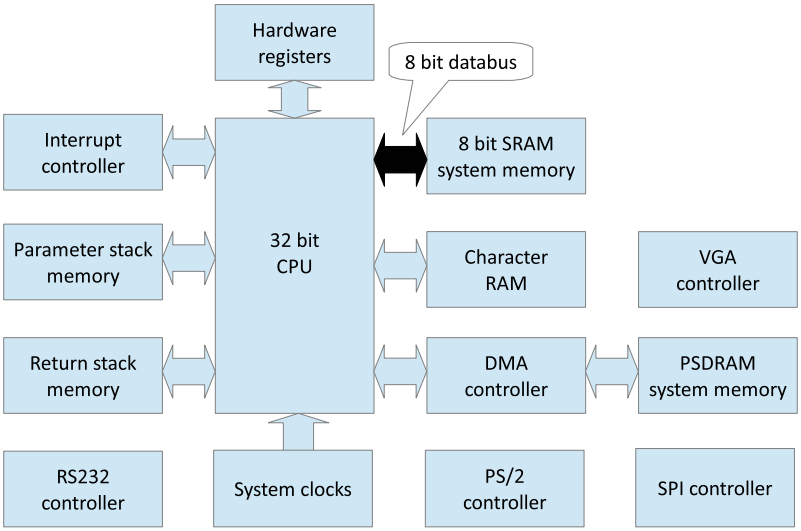

Figure 1: N.I.G.E. Machine system diagram showing the principal components and CPU connections. This is the 8 bit databus configuration (highlighted), as presented at EuroFORTH 2012.

2 Memory access requirements of a FORTH processor

2.1 Review of prior work

A number of softcores have been designed specifically to execute FORTH [3, 4, 5, 6, 7, 8]. Several aspects of the J1[3] directly inspired the design of the N.I.G.E. Machine. In most cases the focus of the design work in these examples has been the CPU itself, while memory access requirements have been less of a consideration. There have been some notable exceptions. For example the RTX 2000 includes an on-chip memory page controller that considerably enhances memory access [10], and Klaus Schleisiek's microcore [4] features pre-incrementing and post-incrementing data memory operations (++!, !++, ++@, @++) that are directly supported by hardware.

Philip Koopman [10], in discussing the characteristics of 16 bit stack machines, makes the point that the fit between the width of the CPU datapath and the FORTH programming model is a key design factor. Koopman observes that an 8 bit CPU is likely be a unsatisfactory from a performance standpoint because too much time would be required in synthesizing 16 bit operations, whilst at the time of writing (1989), specifying a 32 bit CPU might an aggressive specification. The reasonable assumption presumably being made here is that wider datapaths can also access memory across wider databuses, thus increasing processing bandwidth.

Koopman also discusses the limits of memory bandwidth. He makes the point that traditional, register based, processors are very dependent on cache memory. This creates performance bottlenecks that are dependent on the hit ratios of various caches, and the organization of the code produced by the compiler. He envisages stack machines offering an alternative approach to memory organization because of their very fast procedure calls. (Procedure calls are fast on stack machines because there is no need to save the register set in system memory). In the stack machine approach, frequently executed code can be stored in on-chip memory, avoiding the requirement for dynamic memory management. The design of N.I.G.E. Machine follows the approach envisioned by Koopman in that code density is very high, and system memory comprises FPGA SRAM blocks that can be accessed in a single clock cycle.

This paper builds on Koopman's theme of efficient memory access to focus on the specific problem of how best to design the connection between the CPU and system memory on a 32-bit FORTH machine. The first question to answer is, what are the memory access requirements of a 32 bit FORTH processor?

2.2 Impact of system design objectives on memory access requirements

A FORTH processor has some distinct advantages in real-time control applications [9, 10]. The design objectives of the N.I.G.E. Machine reflect its intended role in the real time control of scientific hardware. The principal objectives are listed below and are in turn the main influence on memory access requirements.

Deterministic execution Avoiding jitter in electronic interfaces is essential for precise control and measurement. This requires that the softcore CPU is designed so that all instructions execute in a fixed number of cycles, including SRAM memory access.

High instruction throughput High instruction throughput translates directly into higher processing performance and shorter interrupt response times. This is especially important on FPGA softcore processors that operate at lower clock rates than comparable dedicated microcontrollers. Single clock cycle instruction throughput is the ideal target.

Maximum code density The fastest memory resources available to a softcore CPU are FPGA SRAM blocks. FPGA SRAM also has the advantage over external memory of deterministic access (i.e. guaranteed single clock cycle read/write). However FPGA SRAM resources are typically limited to a several tens or hundreds of kilobytes. To maximize the use of SRAM as program memory, code density needs to be as high as possible. Ideally instructions should be encoded in a single byte.

Flexible memory access With a 32 bits processor, optimizing the speed and flexibility of memory access requires that CPU instructions are available that read or write memory in byte (8 bit), 16 bit, and 32 bit formats. Flexibility is further enhanced if even address alignment is not required when accessing 16 bit or 32 bit data in SRAM system memory.

2.3 Advantages and limitations of an 8 bit wide databus

FPGA SRAM blocks can be configured in a variety of formats by specifying (with the FPGA design tools) the width (i.e. data size: 8 bits, 16 bits, etc.) and depth (i.e. number of address lines) of the required memory resource. The N.I.G.E. Machine's softcore is a 32 bit CPU, but the system presented at EuroFORTH 2012 incorporated SRAM system memory configured in an 8 bit wide format. Coupling the 8 bit wide SRAM to the CPU in this design is an 8 bit databus and an address bus that references memory at byte-by-byte (fig 1).

This configuration has some advantages: an 8 bit databus naturally facilitates the fetching of single byte instructions, and a byte-by-byte address format avoids misaligned address boundaries. The design of separate CPU instructions that read or write memory in byte, 16 bit, or 32 bit formats is also easily facilitated in this configuration by arranging for the CPU control unit to read or write byte data from/to consecutive memory locations in consecutive CPU clock cycles, as required by the length of the data.

In conjunction with the N.I.G.E. Machine's three stage pipeline [2], the 8 bit databus configuration provides straightforward memory access and single cycle execution for almost all instructions. The important exceptions are those instructions that require access to more than a single byte of memory and which therefore require more than one cycle to execute. These include all of the load literal instructions and the memory fetch and store instructions.

This impact of this limitation becomes apparent when considering the relative frequency of instructions that comprise the FORTH system software (table 1). The most common instruction, which occurs almost twice as frequently as any other, is LOAD.W (or "#.W"), the instruction to load a 16 bit literal to the stack. The ubiquity of the LOAD.W instruction in the FORTH system software reflects the load/store architecture of a stack machine CPU and the subroutine threaded nature of FORTH (LOAD.W is the instruction used to load a subroutine address prior to a JSR (jump to subroutine)). In the N.I.G.E. Machine instruction format, LOAD.W is a three byte instruction comprising an opcode byte followed by the high and low data bytes in big endian format. It takes three CPU clock cycles to execute with an 8 bit databus configuration. On account of the narrow instruction fetch, the most commonly executed instruction is therefore one of the minority of instructions that does not to execute in a single cycle. This is a defeat of the optimization maxim, "make the common case fast", and was the key motivation to consider the design of a wide (32 bit) SRAM databus.

Instruction	**Frequency**
LOAD.W	17.88%
JSR	9.17%
RTS and ,RTS	9.06%
LOAD.B	6.47%

Table 1: The most used CPU instructions in the FORTH system software in the 8 bit databus configuration of the N.I.G.E. Machine that was presented at EuroFORTH 2012. The most frequently used instruction, LOAD.W, is one of the minority of instructions that does not execute in a single cycle.

2.4 Design complications resulting from a 32 bit wide databus

When FPGA SRAM blocks are configured in longword (32 bit) data format, each SRAM memory address references a separate, complete longword. Consecutive memory addresses therefore step through memory in units of 4 bytes at each increment. The first concern in designing a wider databus are the mismatches that arise between the 4 byte wide data format and the width of each instruction (1 byte) and the smallest unit of memory access (also 1 byte). These mismatches have a number of important design implications.

The first implication is that the SRAM address bus cannot directly reference memory at the level of individual bytes or 16 bit words. This is at odds with the design requirement that the CPU should be able to read or write to memory either longwords, words or bytes. For memory read instructions this problem can be circumvented by creating a composite address bus whereby the high bits of the composite address bus are matched directly with the SRAM address bus, and the low 2 bits are used to multiplex individual bytes from within the relevant longword (fig 2).

For write instructions there is, however, no simple solution because when configured in longword format, FPGA SRAM performs data writes complete longwords without the flexibility to specify only individual words or bytes within them.

The second implication is the problem of address boundaries. Suppose that the CPU wishes to read 4 consecutive bytes from memory. Two different situations arise (fig. 3). In the first case, the address of the first byte happens to coincide with a longword address within SRAM. This is aligned access and can be accomplished with a single read instruction to that memory address, with a duration of one clock cycle. However in the second case, the desired 4 bytes may be spread across two consecutive longword addresses in SRAM. This is non-aligned access and it requires the CPU to execute two read instructions from the two consecutive memory addresses, thus taking two clock cycles. Either that, or non-aligned access must be prohibited by the CPU specification.

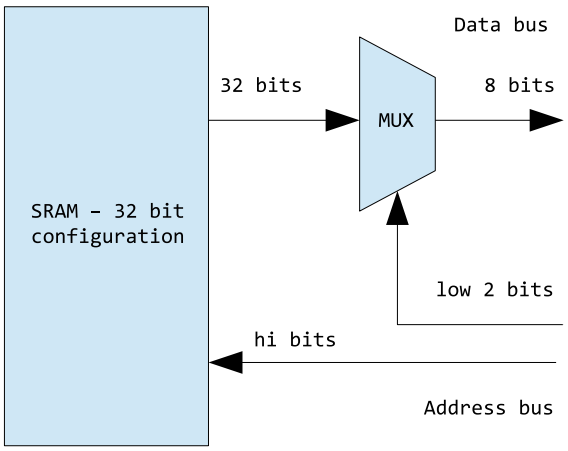

Figure 2: Illustrative scheme for reading individual bytes from 32 bit width RAM. There is no equivalently simple, single step scheme for writing individual bytes to 32 bit width RAM.

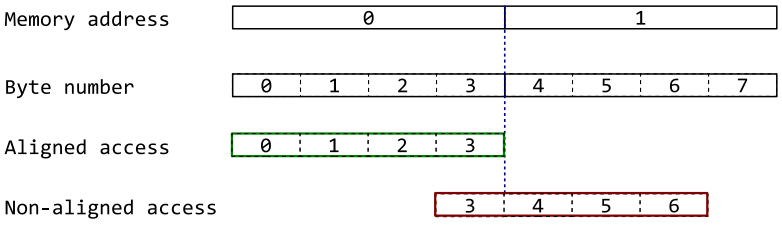

Figure 3: Aligned and non-aligned memory access operations require different treatment.

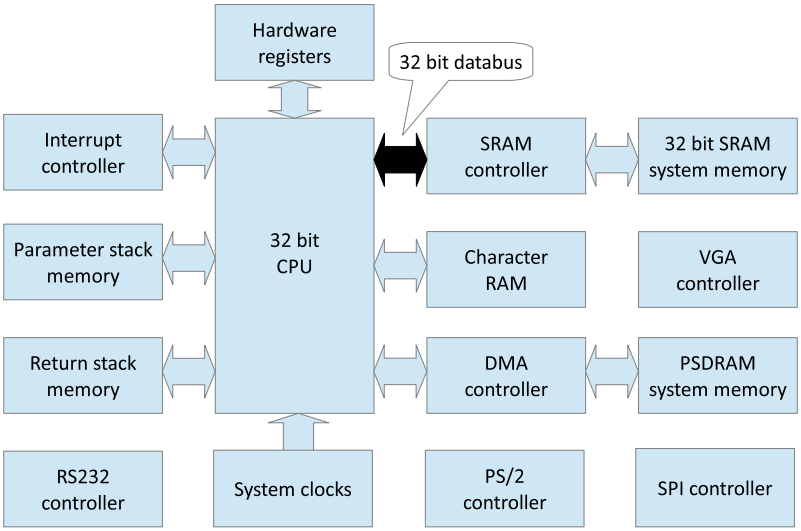

Figure 4: N.I.G.E. Machine system diagram showing the principal components and CPU connections. This is the 32 bit databus configuration, as highlighted.

Neither of these constraints are attractive. Prohibiting non-aligned memory access, especially at the longword level, decreases the flexibility available to the programmer and the FORTH compiler, and wastes space in memory. Requiring the CPU to switch between single cycle and two cycle memory access modes depending on address alignment would mean that instruction execution would no longer be deterministic, thus introducing jitter into signals being generated in real time. Fixing the duration all memory read accesses at the worst case of two-cycles would solve the problem of non-deterministic execution, but at the expense of halving the throughput of all load/store instructions.

These issues are a particular concern for FORTH processors. The real-time control applications of a FORTH processor mean that deterministic execution is often sacrosanct. At the same time, FPGA based softcore processors are inevitably clocked a much lower frequencies than general purpose CPU's and so high instruction throughput is essential. FPGA SRAM resources are also usually limited to tens or hundreds of kilobytes and FORTH has a natural advantage in the very small code size of its applications compared with other high level languages. For memory efficiency reasons, byte level memory access and byte sized instruction coding is therefore also highly desirable.

Fortunately by leveraging a particular capability of FPGA SRAM resources it is possible to design a 32 bit wide memory architecture which circumvents almost all of these constraints. That FPGA SRAM feature is **dual ported memory access**. It is possible to configure FPGA SRAM resources with two independent address and data buses that read or write to separate memory locations in the same clock cycle. The 32 bit N.I.G.E. Machine memory architecture leverages dual ported SRAM to provide a 32 bit wide memory databus whilst maintaining byte level memory access, deterministic execution and single cycle throughput for most instructions.

3 Design for 32-bit wide system memory access

Figure 4 shows the N.I.G.E. Machine system diagram in 32 bit databus configuration. The design is described in detail below.

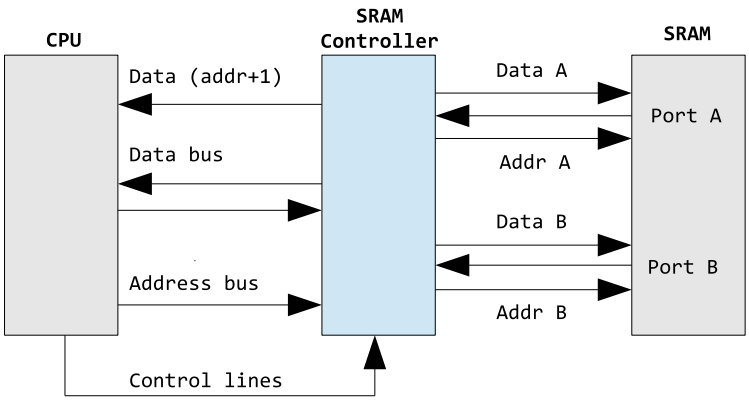

Figure 5: Block diagram of the SRAM memory controller showing the connection to dual ported SRAM blocks

3.1 SRAM memory controller

The key component in the N.I.G.E. Machine's 32 bit memory datapath is the SRAM controller that sits between the CPU and dual-ported SRAM, as shown in fig 5. It provides byte, word and longword read/write access to system memory.

3.1.1 Read functionality

The functionality of the SRAM controller during a memory read is shown in fig. 6. The memory address provided by the CPU points to an individual byte address in memory. The SRAM controller splits this address into two parts: the lowest 2 address bits and the remaining (high) address bits. The high address bits point to the longword address within which the selected byte address lies. The lowest 2 address bit can be interpreted as the offset of the byte address from the zeroth byte of the longword address. The SRAM controller passes the high address bits directly to SRAM port A. It also adds 1 to the high bits address (equivalent to adding 4 to the byte level address) and passes this address to SRAM port B. One clock cycle later, the SRAM read operations occur on both ports simultaneously, and the SRAM controller will have a total of eight contiguous bytes available to it from SRAM ports A and B combined. The offset multiplexer selects four contiguous bytes out of these eight according to the lowest 2 bits of the address specified by the CPU. Finally, the size multiplexer takes a 2 bit control signal from the CPU control unit and selects either a single byte, a single word, or the full longword from the output of the offset multiplexer. In the case of selecting a byte or a longword, the multiplexer shifts the relevant bits to the low end of the output longword and pads the high end with zero.

Table 2 illustrates some worked examples of the SRAM controller read functionality.

3.1.2 Write functionality

The functionality of the SRAM controller during memory write mode is shown in fig. 7. Essentially a write to SRAM now takes place over two cycles, during which time the CPU must hold the address

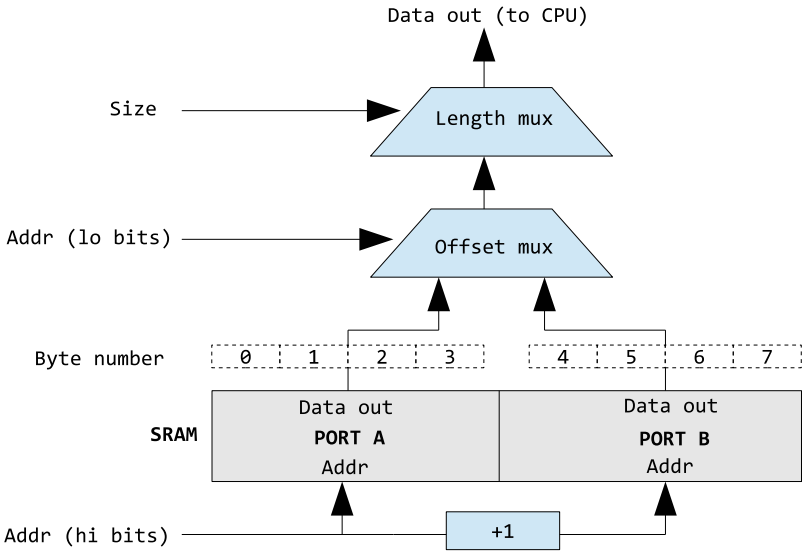

Figure 6: Details of the SRAM controller read functionality. The offset multiplexer select the appropriate longword from the 8 bytes available from SRAM ports A and B. The length multiplexer either passes through the longword or left pads a word or byte output with zero bits, according to the selected size.

	CPU address	**CPU size request**	**Port A address**	**Port B address**	**Offset**	**Memory bytes at output**
A)	0	longword	0	1	0	`[00][01][02][03]`
B)	0	word	0	1	0	`[--][--][00][01]`
C)	7	longword	1	2	3	`[07][08][09][10]`
D)	7	byte	1	2	3	`[--][--][--][07]`

Table 2: Worked examples of SRAM controller read functionality. In case (A) the CPU is reading a longword from memory address 0. The first four bytes in memory appear at the output, in big endian format. In the case (B), the CPU is also reading from memory address zero, but a word. In this case the size multiplexer has shifted the word at memory address zero to the low end of the output databus and filled the high bits with zero (indicated [–] in the table). Cases (C) and (D) illustrate a read from memory address 7. In these cases port A reads longword memory address 1 (byte memory address 4) and port B reads longword memory address 2 (byte memory address 8). The offset of 3 selects a longword starting at third byte on port A.

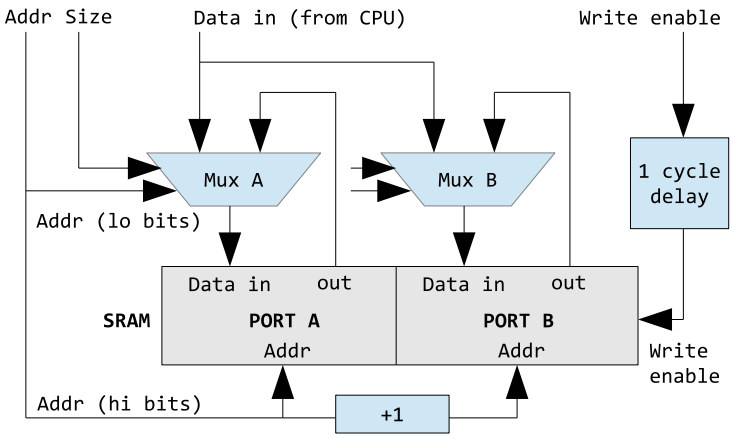

Figure 7: Details of SRAM controller write functionality. A write operation takes place over two cycles. In the first cycle the existing memory contents are read and overlaid at the appropriate position with the data from the CPU. In the second cycle the memory contents are updated.

and data constant on the memory bus. In the first cycle, the existing contents of SRAM memory at the relevant addresses are read and multiplexed with the write data from the CPU. Multiplexing takes into account both the memory address offset, and the size of the data being presented by the CPU (longword, word, byte). The result is that the appropriate overlay of the CPU write data onto the existing memory contents becomes available at the end of the first cycle. In the second cycle the outputs of the multiplexers are written to SRAM.

As with SRAM read functionality both of the dual SRAM ports are active. The low two bits of the address presented by the CPU form the address offset used by the multiplexers, while the high address bits are used to access two consecutive longwords in SRAM. A single cycle delay on the write enable signal from the CPU defers the SRAM write to the second cycle.

3.1.3 Dual data output

In addition to providing non-aligned (byte addressable) longword access for any given memory address, SRAM controller is also configured to output the memory contents at the next following address. This is databus is labeled "Data (addr + 1)" in fig. 5. The purpose of this data is to expedite the execution of load literal CPU instructions. The format of a load literal instruction is a single instruction byte followed by a longword, word, or byte of data. By making available to the CPU the contents at the next memory address beyond the current instruction, the literal data can be multiplexed directly into the datapath during single cycle execution of a load literal instruction.

3.2 Datapath

Minimal changes were required to the CPU datapath design to accommodate the 32 bit memory databus since the datapath width is already 32 bits.

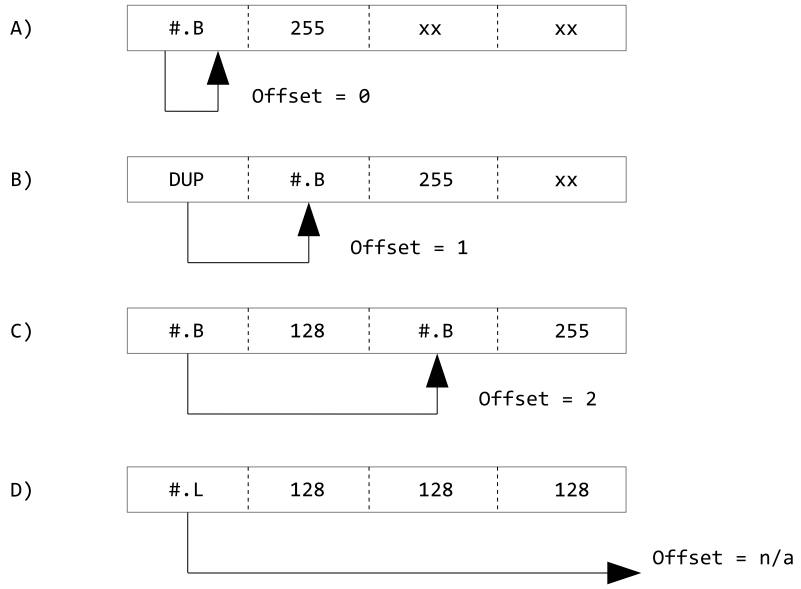

Figure 8: Identification of the next instruction byte on the databus by the program counter logic. In case (A) there is no currently executing instruction and the offset to the next instruction is zero. The next instruction is a load literal byte instruction of length 2. In case (B) the currently executing instruction is one byte in length and the offset to the next instruction is one byte. In case (C) the currently executing instruction is a load literal byte instruction of length 2 bytes, and this is also the offset to the next instruction. Case (D) illustrates that when the currently executing instruction is load literal longword of length 5 the next instruction lies beyond the width of the 4 byte databus and the offset cannot be calculated.

In the 8 bit databus configuration of the N.I.G.E. Machine, memory read data is made available to the 32 bit datapath on a 32 bit accumulator register that is managed by the control unit finite state machine. The register functions to accumulate the required data byte by byte over the required number of memory read clock cycles. In the 32 bit databus design memory the accumulator is not required because read data is available directly from the SRAM controller.

The datapath also has direct access to the SRAM blocks that hold the parameter and return stacks. The stack databuses are always 32 bits wide. The memory holding the parameter and return stacks is dual ported and is also available to the CPU over the system memory databus. When FPGA SRAM blocks are configured with a 32 bit databus on one port (for direct stack access) and with an 8 bit databus on the other port (for system memory access in 8 bit databus format) the SRAM memory layout is little endian by default. The N.I.G.E. Machine is big endian format and hence the 4 bytes of the longword must be reversed when read over the stack databus. This is a minor detail that does not affect performance, but the complexity is avoided in the 32 bit databus configuration.

3.3 Control unit

The control unit required a more considerable redesign to accommodate the 32 bit databus and optimize instruction execution.

3.3.1 Program counter logic

The principal impact of the 32 bit memory datapath on the control unit is that the program counter ("PC") logic needs to be reconfigured to process variable length instructions that execute in a single cycle. In the N.I.G.E. Machine instruction set instructions longer than a single byte only occur when literal data is provided in the second and subsequent bytes (the instruction itself is always fully specified by the initial byte). These include the load literal and branch instructions.

In the 8 bit databus configuration, variable length instructions are executed over several cycles with each successive instruction byte being read from memory in successive cycles. The program counter is therefore hardwired to step in units of a single byte in all circumstances except when a branch or jump occurs. This considerably simplifies the program counter logic. In the 32 bit datapath format, instructions that are longer than one byte and which include literal data also need to be executed in a single cycle. Therefore the program counter logic must decode the length of each instruction and advance by the relevant number of bytes during a single cycle. The process for PC update is as follows.

Firstly the program counter logic must determine the location of the next instruction (i.e. the instruction that will be executed after the currently executing instruction). The control unit is configured so that the currently executing instruction always corresponds to the byte at the lowest memory address on the 32 bit datapath. (This is the highest order byte of the whole longword on a big endian machine such as the N.I.G.E. Machine.) The size of the currently executing instruction always known to the control unit finite state machine and made available as an output (labeled as the "offset", fig 8). The program counter logic refers to the offset to identify which byte of the longword corresponds to the next instruction. For example, the majority of instructions are encoded as single bytes and so the next instruction is the next byte. The load literal byte and word instructions are two or three bytes in length respectively and so the next instruction is two or three bytes ahead respectively. Offsets are not calculated for branch or jump instructions since these require that the PC be diverted rather than incremented.

Once the instruction byte of the next instruction has been identified, that instruction byte is multiplexed to the second stage of the PC logic which determines the instruction length. Finally, in the third stage of the PC logic, the length of the next instruction is added to the current value of the PC, so the the PC will be appropriately updated in the next cycle.

3.3.2 Four stage pipeline

Implementation of the program counter logic is made more complex by the fact that the N.I.G.E. Machine CPU is a pipelined design. As a result the program counter needs to decode the instruction length before, and independently of, the rest of the instruction execution logic. This requires an extra stage at the beginning of the pipeline, which is now 4 stages long as illustrated in fig. 9. The pipeline stages are:

1. "READ INSTRUCTION SIZE". In this example the pipeline is being started afresh (following a jump, branch or reboot) and there is no currently executing instruction. The "offset" is therefore zero. During clock cycle #1 the PC logic identifies the next instruction byte according to the scheme described above. (in this case 53, corresponding to the load literal byte instruction which is 2 bytes long).

2. "FETCH OPCODE". On the rising edge of clock cycle #2, SRAM system memory reads the instruction byte at the current PC address and extracts its opcode. A "new" PC address is determined by adding to the PC the instruction size increment calculated in the previous cycle, in this case 2 bytes.

Component / clock cycle	Cycle #0	Cycle #1	Cycle #2	Cycle #3	Cycle #4
Program counter	0	0	2		
Offset		0			
Next instruction byte		53			
Length of next instruction		2			
Instruction byte			53		
Opcode			53		
Microcode				1191	
Datapath combinatorial logic				255	
Datapath synchronous logic register					255

Figure 9: Illustration of the execution pipeline for the CPU instruction to load a literal byte with value 255. Executing variable length instructions in a single cycle requires an extra stage in the pipeline (clock cycle #1 in this illustration).

3. "DECODE AND COMPUTE". On the rising edge of clock cycle #3, SRAM microcode memory within the control unit takes the opcode as a lookup address and returns the corresponding microcode value (1191). During the same clock cycle the combinatorial logic in the datapath is configured according to the microcode value through its control signals. The value of the datapath computation becomes available as the combinatorial output, in this example the literal value loaded is 255.

4. "SAVE". On the rising edge of clock cycle #4, the output of the datapath in combinatorial logic (i.e. the result of the computation in the previous pipeline stage) is written into the synchronous logic register that holds the value of the top of stack.

3.3.3 Instruction throughput

The number of clock cycles required to execute each instruction in both the 8 bit and 32 bit databus configurations of the N.I.G.E. Machine is scheduled in table 3. The differences in throughput between the two configurations results from two opposing factors. (i) The 32 bit databus configuration reduces the number of clock cycle required to execute instructions that load, fetch, or save access word and longword data in SRAM system memory because the full width of the data can be handled concurrently. However (ii) the additional pipeline stage adds an extra cycle to all instructions that require the restart of the pipeline.

The following analysis of instruction throughput speaks from the perspective of the 32 bit datapath configuration / the changes made from the 8 bit format.

- The load literal byte and load literal word instructions now execute in a single cycle. However the load literal longword instruction actually requires two cycles to execute. This is because the length of that instruction (5 bytes) means that whilst it is being executed, the next instruction byte is not visible on the datapath to the PC update logic (fig 8), and hence an extra cycle must be added with no instruction to restart the pipeline.

- Branches also require a restart of the pipeline because of the change to the PC. However the extra cycle this entails is offset by the fact that the whole two byte instruction can be read and decoded in a single cycle. As a result the total execution cycle count is unchanged at 3 cycles.

Instruction	**Mnemonic**	**Cycle count (8 bit databus)**	**Cycle count (32 bit databus)**
Load literal byte	LOAD.B (or #.B)	2	1
Load literal word	LOAD.W (or #.W)	3	1
Load literal longword*	LOAD.L (or #.L)	5	2
Branch (conditional or unconditional)*	BEQ / BRA	3	3
Jump to subroutine (address on stack)*	JSR	2	3
Jump to subroutine (literal address)*	JSL	n/a	3
Return from subroutine*	RTS	2	3
Fetch/store byte in SRAM*	FETCH.B / STORE.B	2	3
Fetch/store word in SRAM*	FETCH.W / STORE.W	3	3
Fetch/store longword in SRAM*	FETCH.L / STORE.L	5	3
Fetch/store in PSDRAM	FETCH.[] / STORE.[]	variable	variable
Multiply (signed/unsigned)	MULTS / MULTU	6	6
Divide (signed/unsigned)	DIVS / DIVU	43/42	43/42
?dup	IFDUP	2	2
All other instructions		1	1

Table 3: Clock cycles per instruction in the N.I.G.E. Machine softcore CPU in both 8 bit and 32 bit databus configurations. Instructions marked * require a restart of the pipeline following their execution. Most, but not all, instructions are faster in the 32 bit configuration.

- The JSR instruction is now one cycle longer due to extra cycle to restart the pipeline. This might imply a considerable performance penalty in executing FORTH code, which is heavily subroutine dependent. However the 32 bit databus configuration permits the inclusion of a new instruction, JSL, that provides a considerable efficiency. This instruction is a "jump to subroutine" with the subroutine address as a 24 bit literal value. Previously, the typical FORTH code to execute a subroutine branch comprised (i) #.W, to load the subroutine address onto the stack, followed by (ii) JSR. This combination requires a total of 5 cycles. The JSL instruction accomplishes the same result in 3 cycles. However the advantage of 2 cycles on a subroutine call is offset by the fact that an RTS instruction is also one cycle longer due to the lengthened pipeline. The net difference is that subroutine calls are now one cycle faster overall. As a side note, the introduction of the JSL instruction did not necessitate significant rewriting of the N.I.G.E. Machine system software. The system software is written in assembly language, and the macro assembler implements either style of subroutine call with a macro, "CALL", appropriate to whichever version of the hardware is being compiled for.
- Fetches and stores to SRAM system memory of all datasizes now execute in three cycles, compared with two, three, and five cycles for byte, word and longword data previously. In FORTH terms, C@ and C! are slower than before, W@ and W! are unchanged, and @ and ! are faster than before.
- Other instructions that do not access SRAM system memory and do not restart the pipeline are unchanged. Fetch and store to the external pseudo-static dynamic RAM ("PSDRAM") takes place through the PSDRAM controller and timing depends on the arbitration of the bus with other users of PSDRAM memory such as the VGA controller.

In summary, whilst for most instructions the softcore CPU throughput has been increased in the 32 bit datapath design, it is clear that there have been tradeoffs in certain cases. To assess

Instruction	Frequency	Clock cycle difference
JSL	10.2%	-2
#.W	9.8%	-2
RTS and ,RTS	8.7%	+1
BRA and BEQ	8.7%	0
#.B	7.3%	-1
DUP	4.8%	0
DROP	4.1%	0
FETCH.L	3.5%	-2
FALSE	3.3%	0
SWAP	3.3%	0
OVER	3.2%	0
STORE.L	3.1%	-2
R>	2.6%	0
+	2.6%	0
FETCH.B	2.6%	+1
>R	2.1%	0
STORE.B	1.9%	-1

Table 4: Relative instruction frequency for the 80% most common instructions in the N.I.G.E. Machine system software (counted by code frequency rather than execution frequency). The clock cycle difference values are the instruction duration difference in moving from the 8 bit to 32 bit databus configuration. Negative values indicate that the 32 bit configuration is faster. The most used instruction, JSL, is 2 cycles faster in the 32 bit databus configuration.

whether the 32 bit datapath configuration should be expected to lead to higher performance overall, it is also necessary to examine the frequency of instruction usage. Table 4 schedule the most frequently used instructions in the N.I.G.E. Machine system software. Despite the fact that some instructions are slower in the 32 bit datapath configuration, taking into account which instructions are most common, the results suggest that average instruction throughput should be improved in this configuration.

4 Results

4.1 Hardware implementation

Implementation of the 32 bit datapath format on the Nexys 2 FPGA development board proved more challenging than anticipated. Whilst synthesis was satisfactory in the electronic simulator, the new design was not able to place and route to meet the timing constraints of a 50MHz a clock speed. This was in spite of considerable optimization work with the FPGA design tools. The reason for the slower timing in the 32 bit bus configuration was revealed by analyzing the place and route results and identifying the longest signal path. This is the logic required to operate the variable instruction length program counter logic in the control unit. The steps involved are shown schematically in fig 10. As described in the section discussing the control unit, these steps are necessary if the N.I.G.E. Machine is to execute variable length instructions in a single clock cycle.

Table 5 summarizes the best achievable timing of the N.I.G.E. Machine in both 8 bit and 32 bit

	N.I.G.E. Machine (8 bit databus)	N.I.G.E. Machine (32 bit databus)
Best achievable timing (ns)	18.22	22.97
Equivalent clock frequency (MHz)	54.89	44.84

Table 5: Best achievable synthesis and place and route timing for the N.I.G.E. Machine on the Nexys 2 development board with a Xilinx XC3S1200E FPGA. The 32 bit databus configuration fails to make timing for a 50MHz clock speed.

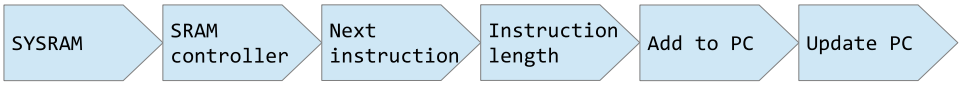

Figure 10: Schematic of the longest signal path in the N.I.G.E. Machine 32 bit databus configuration that is responsible for limiting the best achievable timing to more than 20ns

databus formats. The best achievable clock frequency in 32 bit format was 44MHz. However it is not possible to operate the N.I.G.E. Machine at arbitrary clock frequencies (say 40MHz), because there are also timing constraints imposed by the peripheral components. In particular the VGA controller should operate at 25MHz or 50MHz on account of to the VGA signal specification, and the system clock needs to be synchronized at a multiple of the VGA clock frequency.

For the purpose of comparative benchmarking, the 32 bit databus configuration N.I.G.E. Machine was successfully implemented at 50MHz by reducing the depth of SRAM memory to 4K only. (By reducing memory depth, the number of SRAM block multiplexers is reduced and therefore also the signal time for an SRAM read. The saving was enough to compensate). However this workaround has limited scope, since whist the benchmarks can be run in under 4K of memory, the restricted in memory size is too severe for a general purpose microcomputer.

4.2 Performance benchmarks

A series of benchmarks were run to compare the performance of the N.I.G.E. Machine in both the 8 bit and 32 bit databus configurations at 50MHz. The benchmarks were based on a number of standard FORTH tests [12], minimally adapted to run in an embedded environment. As a baseline comparison, the benchmarks were also run on a Intel i7 desktop PC at 2.8GHz using VFX FORTH. Tables 6 and 7 show the results.

The N.I.G.E. Machine in 32 bit databus format is on average 14% faster in primitive operations and 20% faster in applications than in 8 bit databus format. However the speed increase varies according to the application. Eratosthenes's sieve is only 4% faster while the eight queens problem is 29% faster. This is because not all instructions are faster in the 32 bit datapath format and so the instruction mix of an application is also important. However for real life applications the benchmark average should be a good guide.

The N.I.G.E. Machine in 32 bit databus configuration is approximately 150x slower on average than an Intel i7 PC running VFX FORTH, but again the range varies from 120x for random numbers to almost 300x for quicksort. However the N.I.G.E. Machine was clocked at 50MHz while the PC was clocked at 2.8GHz, a difference of 56x. Allowing for the difference in clock speed, the N.I.G.E. Machine was only on average 3x slower than the PC.

Benchmark	**Iterations**	**N.I.G.E. Machine (8 bit bus)**	**N.I.G.E. Machine (32 bit bus)**	**PC i7 VFX FORTH**
Primitives		ms	ms	ms
DO LOOP	1,000,000	260	260	-
+	1,000,000	340	300	-
*	1,000,000	440	420	-
/	1,000,000	1,240	1,200	
/MOD	1,000,000	1,240	1,200	16
*/	1,000,000	1,420	1,400	-
Array fill (1000 items)	1,000,000	9,008	7,207	15
		13,948	11,987	31
Applications				
Eratosthenes sieve	3000	19,680	18,884	110
Fibonacci recursion	1	44,272	37,947	265
Quick sort	1,000	10,924	9,171	31
Random numbers	1,000	48,795	35,643	296
Bubble sort	100	41,089	33,387	218
Eight queens	50	37,774	26,968	141
		202,534	162,000	1,046

Table 6: Benchmark timing results for the N.I.G.E. Machine in 8 bit and 32 bit datapath configurations, and the same tests run on an Intel i7 PC using VFX FORTH.

Benchmark	**N.I.G.E. Machine (32 bit) / (8 bit)**	**N.I.G.E. Machine (32 bit) / PC i7**
	%	multiple
Eratosthenes sieve	96%	172
Fibonacci recursion	86%	143
Quick sort	84%	296
Random numbers	73%	120
Bubble sort	81%	153
Eight queens	71%	191
Total	80%	155

Table 7: Relative benchmark timing results for the 32 bit datapath format N.I.G.E. Machine compared to the 8 bit datapath format N.I.G.E. Machine and a PC i7 running VFX FORTH . The 32 bit datapath configuration is on average ~20% faster than the 8 bit configuration.

5 Discussion

At the outset it was expected that the main challenge in widening the memory datapath from 8 to 32 bits would be to design appropriate logic to maintain deterministic execution, single-cycle instruction throughput, byte-sized instruction format, and flexible memory access. Two major components had to be developed to accomplish these objectives. Firstly, developing an SRAM controller that leveraged the dual ported SRAM blocks available on the FPGA. Secondly, an adaption to the control unit to facilitate the execution of variable length instructions in a single clock cycle.

However these components were included at the expense of additional logic levels and a longer signal path. This reduced the maximum achievable clock frequency. Whilst the 32 bit memory databus configuration completes benchmarking tests in approximately 20% less clock cycles that the 8 bit configuration, the maximum clock frequency that can be achieved at implementation is also roughly 20% less (~40 MHz c.f. ~50MHz)

Perhaps in retrospect this tradeoff should have been anticipated. It is similar to the tradeoff between the RISC (reduced instruction set computing) and CISC (complex instruction set computing) approaches to CPU design, and occurs for similar reasons. RISC designs utilize less logic but can operate at a higher clock speed compared to CISC designs that have more sophisticated instruction set functionality.

The question remains as to which general approach is more appropriate for a FORTH softcore such as the N.I.G.E. Machine? Either a "CISC like", 32 bit databus with the ability to execute variable length instructions in a single cycle, resulting in instruction execution that completes in few clock cycles. Or a "RISC like", 8 bit databus matched to the instruction size with fewer layers of logic, resulting in a higher implementable clock frequency?

The answer to this dilemma may be to look more carefully within the CPU at the differing needs of the control unit and the datapath. The datapath within the N.I.G.E. Machine's softcore CPU is 32 bits wide. Matching the 32 bit datapath to a 32 bit memory databus optimizes execution speed by maximizing data transfer bandwidth. On the other hand, the control unit can operate at a higher clock speed when there is no need to execute variable length instructions in a single cycle. A hybrid design can be envisaged that maintains the 32 bit memory databus matched to the 32 bit datapath, but reverts to a control unit that processes instructions on a byte-by-byte basis. Such a hybrid design could be expected to have the following characteristics:

- Maximum clock speed no slower then the 8 bit databus configuration (i.e. $>$ 50MHz)
- Fetch/store instructions execute as fast as with the pure 32 bit databus configuration
- The fast JSL (jump to subroutine literal address) instruction would be available
- The pipeline would revert to 3 stages, eliminating the extra clock cycle restart penalty of the 4 stage pipeline
- Load literal instructions revert to the slower speed of the 8 bit databus configuration

Based on the results discussed above, such a hybrid design is likely to prove a better performer than either the pure 8 bit or 32 bit databus configurations. Development along these lines is an attractive avenue for further work on the N.I.G.E. Machine.

6 Conclusion

This project set out to widen the N.I.G.E. Machine's memory databus from 8 bits to 32 bits. In doing so it was found that neither configuration is absolutely better than the other. The tradeoffs

between them concern maximizing clock speed versus maximizing instruction throughput. This result parallels the differences between the RISC and CISC approaches to CPU design. In the case of the N.I.G.E. Machine, a hybrid memory databus that addresses the differing needs of the CPU control unit and datapath is likely to be the optimum performance strategy. Further work will be undertaken on the N.I.G.E. Machine to implement such an approach.

Whilst it is recognized that differing processor designs have differing design tradeoffs at a detailed level, some general conclusions about the strategy for optimizing memory access design for a 32 bit FORTH processor can be drawn from these project results. A FORTH processor is likely to be optimized for the efficient execution of a basic set of stack and memory operations, subject to embedded control objectives such as deterministic execution and high code density. Maximum clock speed is achieved with simple control unit logic. Given these considerations, it is likely desirable to match the width of the memory databus to the control unit to the width of a single instruction (8 bits on the N.I.G.E. Machine). On the other hand, a FORTH processor is a fetch/store architecture and so data bandwidth will be maximized by matching the width of the memory databus to the width of the CPU datapath (32 bits on the N.I.G.E. Machine). The best overall approach is therefore likely to adopt a hybrid databus design, whereby the needs of the CPU control unit and datapath are separately identified and addressed.

References

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

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

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

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

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

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

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

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

[9] Stephen Pelc, "Programming FORTH", MPE, 2011

[10] P. J. Koopman, Jr., "Stack computers: the new wave", Halsted Press, 1989

[11] The author, Github open source repository https://github.com/Anding/N.I.G.E.-Machine

[12] MPE benchmark suite for 32 bit Forth systems, http://www.mpeforth.com/arena/benchmrk.fth