MICROPORCESSOR ARCHITECTURE WITH FPGA IMPLEMENTATION FOR UNDERGRADUATE COMPUTER ARCHITECTURE COURSES

Dr. Jonathan Hill
University of Hartford

Abstract

This paper presents a simple yet nontrivial Von Neumann style computer architecture and an corresponding implementation suitable for an undergraduate course in computer architecture. The processor is said to be soft as it is implemented with a field programmable gate array (FPGA). This paper incorporates the latest nod4 implementation called nod4.1 which has significant improvements over nod4.0, including the use of unidirectional busses. The processor architecture itself is worthy of study, including such features as subroutines, stack relative addressing, interrupts, and conditional branching. The processor is able to pre-fetch with some instructions and provides performance comparable to traditional small microprocessors such as the Freescale 68HC11. The documentation is written so that several options are possible for introducing nod4 into the classroom curriculum. In particular, students may investigate the nod4 processor or implement the processor themselves. It is also possible to present the processor architecture entirely without the implementation.

Introduction

This paper presents a simple yet nontrivial Von Neumann style computer architecture and an implementation that undergraduate students may implement as a soft-core processor. Engineers are continually called upon to make decisions regarding what is appropriate for a given application. The *grand vision* serves as a north-star to inspire and help the designer in making decisions regarding a given architecture. The nod4 processor is designed to be a tool for teaching introductory computer architecture principles to undergraduates. The nod4 motto is, "*simple yet nontrivial*." It is classic accumulator based Von Neumann style architecture. The design strives for clarity and is transparent so nothing is hidden from the student. It has an 8 bit address bus and primarily supports unsigned 8 bit integer math.

Relevant references include Mano and Kime[1] as well as Tanenbaum[2]. To implement nod4 the target technology is the field programmable gate array (FPGA). Other than switches, light emitting diodes, and the clock oscillator, the nod4 processor system is implemented entirely in a FPGA. Students are provided with VHDL modules used to make schematic symbols. In this way students use register level or higher schematics. The development tools include a simulator for examining the system cycle by cycle behavior.

The support provided to software is an important concern. Compiler generated machine code makes use of only a few addressing modes and is generally supported by certain processor hardware features. The nod4 architecture has a stack, uses subroutines, and includes stack relative addressing for passing parameters. Other than the possibility of a very simple executive, we have no interest in supporting a formal operating system. At the very least, to perform a context switch requires direct access to the processor stack.

Performance means something different for each application we consider. In executing simple demonstration programs nod4.1 strives for a respectable level of performance comparable to such classics as the Freescale 68HC11 microcontroller. As outlined by Tanenbaum, to avoid expressing the fetch-execute cycle as a binary tree, the microcode is aided with jump ahead rules. Also, in fetching an instruction, two bytes are read from memory, allowing some instructions to pre-fetch the following opcode. With nod4.1 students are exposed to such classic metrics as cycles per second, average cycles per instruction, and integer operations per second.

The nod4 system is broken into three documents. The nod4 architecture document[3] focuses primarily on the assembly language view of the processor. The memory system document[4] and the implementation document[5] together outline the actual implementation. Documentation is also written for optional peripherals. In other courses that have a focus only on architecture, the processor can be presented without the implementation. Courses with a lack of development tools can use the implementation as reference. With the development tools on hand, a project can be assigned to actually implement the project. Otherwise, students could possibly use an existing implementation to investigate the nod4.1 processor, considering changes to the nod4 architecture and implementation. There are many opportunities such as adding peripherals, new instructions, and addressing modes. Each document includes homework exercises.

The nod Series History

The first time I taught a computer architecture course I wrote a hypothetical microprocessor architecture called nod1, which was simply meant to serve as an example. To my surprise I discovered its value in teaching. I found the instruction set and encoding worthy of discussion, serving to contrast with text-book examples. The assembly language is educational without being a burden. Such an example is a benefit in its own right and for this I produced an improved version called nod2 which I used the next two times I taught the course.

First with nod1 and then nod2, students had a project to write a simulator program to model the respective architecture behavior. In reviewing feedback, while the students felt that each of the architectures was useful, the corresponding simulator project was too abstract. I was also concerned that the simulator did not fully help to convey a sense of the fetch-execute cycle. It seems that anything less than an actual implementation would not be acceptable.

After deciding to have an actual implementation, I considered a number of factors and made some decisions. I cannot require my computer architecture students to know a hardware description language like VHDL and I feel that pure schematic capture techniques are too intensive in this regard. I selected a hybrid approach where students use pre-written VHDL modules to define the blocks in a schematic. In this way students encounter higher-level schematics. This is similar to using MSI parts in that the underlying VHDL code describing the behavior is already provided. Students perform simulation and once '*things look good,*' the design can be configured into a field programmable gate array.

Refining nod2 led to nod3. Given the prior student feedback, I introduced nod3 as example architecture in the same manner that I introduced nod1 as well as nod2. Later in the course, students actually implemented nod3. In the following year, refinements led to the current nod4 architecture and the nod4.0 processor which uses a bidirectional data bus and students hand wrote test benches.

In considering my experience with nod4.0 and student issues, I realized that an improvement was needed. I realized that unidirectional busses are somewhat common with soft core processors. As an example the Microblaze[6] soft core processor makes some use of unidirectional busses. A final revision led to the current nod4.1 processor which uses unidirectional busses and a simpler data path. A simple graphical tool is used with nod4.1 to develop test-bench files. The nod4.0 and nod4.1 processors are both examples of the nod4 architecture.

The nod4 Architecture

To introduce nod4 to students we start with a fairly abstract view, presenting the registers, assembly language, and encoding. The nod4 architecture has an 8-bit data path and an 8-bit address bus. From the programmer's point of view nod4 has the following CPU registers

- A – accumulator
- C – condition code register (Z,C,I) and IID
- S – stack pointer
- X – index register
- PC – program address counter

The A register is primarily for handling data. The C register contains the zero flag (Z), carry/borrow flag (C), and the interrupt enable flag (I). The lower five C register bits store the ID for an interrupting device (IID). The stack pointer maintains the stack data structure. The X register is a fairly general purpose index register. The program counter (PC) can be thought of as referring to the next instruction however due to pre-fetching has a twist discussed later, that the assembly language programmer is less concerned with.

Students typically resist the notion that data is accessed by address. The nod4 syntax is inspired by the Borland Turbo Assembler (TASM) ideal-mode syntax[7], which is more intuitive than most and is helpful in this regard. In particular, square brackets imply *the contents of the address*, which makes the syntax for the addressing modes almost self explanatory.

Until recently, nod4 programs were assembled by hand to produce machine code. A recent change is the development of an actual assembler program called nod4asm. To avoid having to memorize a numeric value, nod4 assembly language programs allow symbols, or symbolic names for values. A label is like a symbol, but the value it represents must be an address. The assembler determines the actual value assigned to each label. An assembly language program is written in lines of text, each with as many as four fields.

- The left-most field contains a label, symbol, or semicolon to start a comment line. Each label or symbol ends with a colon ':'.
- The second field contains either a mnemonic or an assembler directive which is a command directed at the assembler
- The third field, called the operand field may contain instruction data which is dependent on the addressing mode, or data for an assembler directive
- The fourth field is for comments and starts with a semicolon ';'.

The effective address or EA is the location for a memory data access. Four addressing modes are supported, namely implied, immediate, direct, and indexed. With implied addressing (IMP) there is no operand however as with push and pop the EA is implied. An immediate instruction (IMM) follows the mnemonic by the required data. With direct addressing the mnemonic is followed by the EA value. With index addressing the EA is calculated by adding an offset value following the mnemonic to the corresponding index register (S or X). The following is the general format for lines in an assembly language program

```
; here is a comment line
Label:  mnemonic   Operand  ; comment text
Symbol: directive  Data     ; another comment
```

A directive or pseudo-instruction is an explicit command directed at the assembler. The following are the directives, presented in context

```
          ORG   Address
```
Sets the current point of assembly to ‘Address’

```
symbol:  equ   val
```
The symbol is assigned the constant value ‘`val`’.

```
label:   FCB   val1, val2,…
```
Inserts successive byte values into memory. The address of the first or left most value is assigned to the label

```
label:   RMB   n
```
Reserves n bytes without inserting any values. The address of the first byte reserved is assigned to the label.

Table 1: Instruction distribution.

Mnemonic & Behavior		Addressing Modes				Registers			
		IMP	IMM	DIR	IND	A	C	S	X
`clra`	clear A	–				–			
`inva`	ones comp. A	–				–			
`nega`	negate A	–				–			
`rts`	return from sub.	–						–	
`rti`	return from int.	–				–	–	–	–
`swi`	software int.	–				–	–	–	–
`pshR`	push R	–				o	o	–	o
`popR`	pop to R	–				o	o	–	o
`decR`	decrement R	–				o		o	o
`incR`	increment R	–				o		o	o
`jsr`	jump to sub.		–					–	
jumps(7)	jumps – 7 total		–						
`andR`	bitwise-and w. R		o	o	o	o	o		o
`cmpR`	compare w. R		o	o	o	o	o		o
`orR`	bitwise-or w. R		o	o	o	o	o		o
`addR`	add to R		o	o	o	o		o	o
`stR`	store R			o	o	o		o	o
`subR`	subtract from R		o	o	o	o		o	o
`ldR`	load into R		o	o	o	o		o	o

Table 2: Instruction Encoding Summary.

	Opcode Formats
1.	`0 ACX mmn xx`
2.	`0 10  mmn xx`
3.	`1 ASX mmn xx`
4.	`1 01  mmn xx`

Register Choices	
Reg.	Encoding
A	0 0
C	0 1
S	1 0
X	1 1

Addressing Mode	
Modes	mmn
IMP	0 0 x
IMM	0 1 x
DIR	1 0 x
IND-S	1 1 0
IND-X	1 1 1

The nod4 Instruction Encoding

The first part of an actual machine code instruction is called an opcode. The means by which an opcode conveys an action, addressing mode, and the registers involved is called the encoding. The nod4 encoding is meant to contrast with the principle of the expanding opcode presented by Tanenbaum[2]. The nod4 instruction encoding is formulated from Table 1. The headings IMP, IMM, DIR, and IND refer to implied, immediate, direct, and index addressing modes, respectively. The headings A, C, S, and X refer to the corresponding registers. The '–' symbol means use of an item without a choice and 'o' means a choice among items. Instruction mnemonics use the nameR format where R may refer to a source or destination register. Instructions not ending with R either imply or otherwise do not refer to any registers.

The encoding is not orthogonal and takes advantage of patterns in the instruction distribution. Note that certain registers are sometimes excluded. There is no point in incrementing or decrementing the condition code register C or pushing or popping the S register. In examining the instruction distribution we make several observations:

- The register choices are indicated with three patterns, either none, the set A, C, and X (ACX), or the set A, S, and X (ASX).
- The jump instructions only use immediate addressing with no choice of registers
- Mnemonics that write to memory only make use of direct and indexed addressing modes

Table 2 outlines the encoding. The items ACX and ASX refer to a two bit code that references one of the given registers. The item mmn refers to the addressing mode. The 'x' symbol indicates a bit involved in selecting an instruction from the group. The remaining details of the instruction encoding are in the Architecture document[3].

Example Program

The following program named ex0.asm illustrates the assembly language as well as many of the architecture features. The system has ROM from addresses $00 to $BF and RAM from $C0 to $FB. The first two addresses in memory are reserved for storing the program start address (PSA) and program interrupt address (PIA). Without an interrupt service routine, it is wise to use the PSA as the PIA so that an accidental interrupt will restarts this system. The last four addresses are for peripheral devices.

In the code, the first appearance of each addressing mode type is indicated in the comment field, as in IMM, IND, IMP, and DIR. The push instruction decrements the S register before writing to memory, so that the subroutine return address is written to address $FB. Before returning, the final value is written for display to the output port. To take this example further, consider the online documention[3].

The nod4.1 Implementation

A hardware designer looks at a microprocessor in a different way than the assembly language programmer. Figure 1 is the classic Von Neumann structure I had in mind when I designed the nod4 system. The memory contains executable code and data. The controller produces enable signals to control the actions of the data path, which in return produces status information. Here the input and output (I/O) devices are said to be memory mapped in that the devices are also accessed by address. In the following, each block is considered in turn.

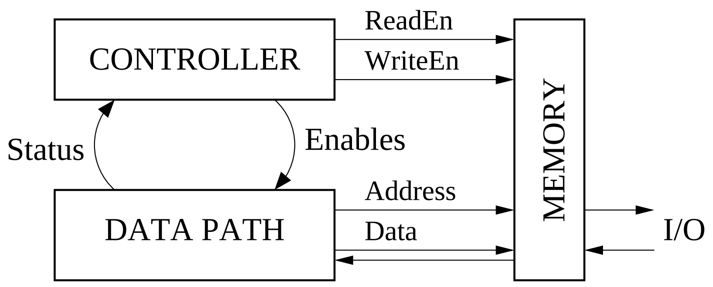

Figure 1: Processor system overview.

```
; ex0.asm – Jonathan Hill – Dec. 7, 2007
; Demo nod4 program showing some features
TOS:    EQU  $FC         ; top of stack
OUTP:   EQU  $FC         ; output port
        ORG  $00         ; set origin
        FCB  Start, Start ; PSA, PIA

Start:  lds  TOS         ; (IMM) init. stack
        ldx  Valx        ; address of val.
        jsr  Absval      ; call sub.
Done:   jmp  Done        ; all done

Absval: lda  [X+0]       ; (INDX) get val.
        cmpa $80         ; compare endval
        jlo  Posval      ; already pos?
        nega             ; (IMP) form opposite
Posval: sta  [OUTP]      ; (DIR) store val
        rts              ; sub. Done
Valx:   FCB  $37         ; a value
```

Memory systems is a topic that itself is worthy of several lectures. As such, discussion of the nod4.1 memory system is split off to a separate document[4]. Students learn from the architecture document[3] and memory system document that the nod4 memory system has three regions comprised of ROM, RAM, and device registers. They also learn that ROM is suitable for permanent executable code and constant data, and that RAM is suitable for variables and the stack. Device registers provide access to the peripheral devices.

For discussions of memory types and memory maps to be more than a simple exercise, students learn about what a simple memory bus is. In reality *the bus* comprises several busses that together convey address, data, and control information. Students learn what address decoding is and how a memory access is performed. In studying the memory system document[4], students discover how each region is a manifestation of a device, on a bus, mapped by the address-decode logic to a range of addresses. Besides memory, so-called memory mapped peripheral devices are accessed by address, as part of the memory system. These are general principles that aid further learning. With an understanding of the basics, students can appreciate more advanced memory systems.

Figure 2 is the nod4.1 memory system. Each block corresponds to a small bit of VHDL code that students are welcome to explore. Unlike symbols that refer to conventional discrete logic devices, these symbols are simply part of a larger description. A key point with FPGAs is being able to tailor to an application. While a 192 byte ROM or a 60 byte RAM may not be practical by itself as a discrete device, the FPGA has the necessary resources. The tools simply allocate the required FPGA resources and automatically route the corresponding logic.

In Figure 2 the signal ax is the address bus, which conveys the address for a memory system access. The signal do is the data bus from the CPU and di is the data bus to the CPU. Three-state logic allows multiple devices to share the di bus. Devices also share the read (rd) and write (wr) control lines. Enables are produced by the memena component so that each device

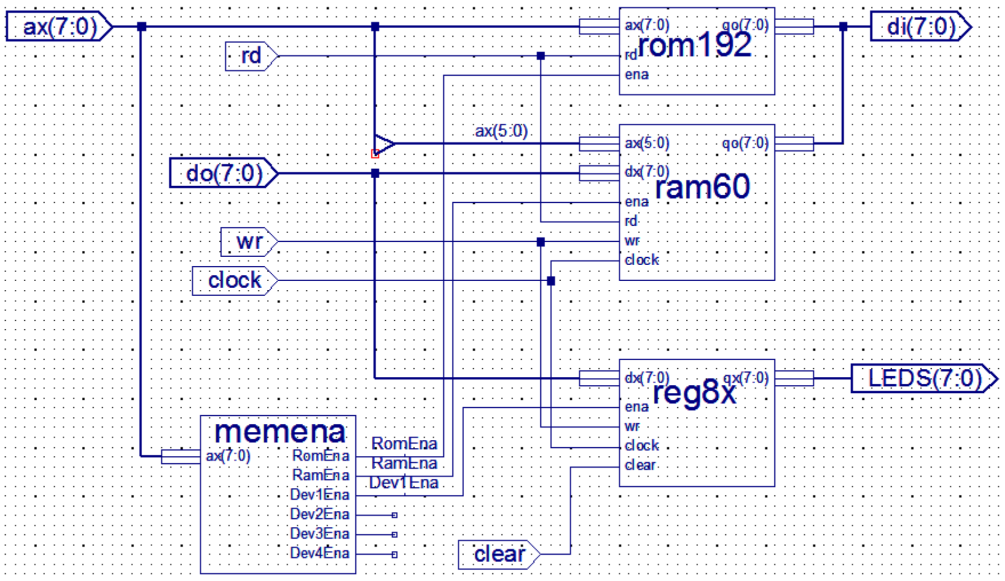

Figure 2: Memory system for nod4.1

appears separately in a region of memory. The memory used in the RAM is called asynchronous-read, synchronous-write and is similar to conventional static RAM, except that a write, as in Figure 3 is committed at the rising clock edge. As with static RAM, in performing a read from memory as in Figure 4 there is a delay to the arrival of valid data. Students can consider different ways to implement the enable logic.

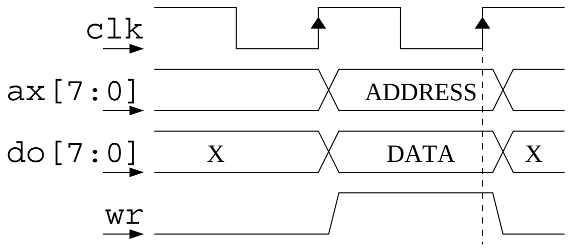

Figure 3: Memory write cycle.

The data path in Figure 5 is what performs the work of the microprocessor. The data path includes all the visible registers, hidden registers, arithmetic logic unit, and all the so called *interconnect plumbing* including multiplexers. The data bus register (DX), temporary register (ND) and instruction register (IR) are all said to be hidden from the assembly language programmer's view. The action of the data path is directed by enable signals (not shown here) produced by the controller. The arithmetic logic unit (ALU) is the real worker in the data path. Students learn that the so-called program counter (PC) is not a counter. The data path in return provides the controller with *status* information, namely the values in the DX, C and IR registers.

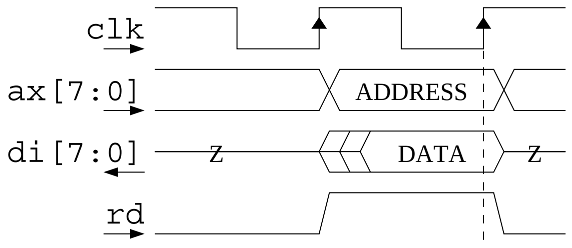

Figure 4: Memory read cycle.

In performing a read from memory the DX register loads the value. Care is taken so that the actual value fetched is only used inside the data path once it is actually loaded into DX. Using a buffer register in this way, so that the data bus is not directly inspected by logic is significant. This means that the fetch and decode phases of the fetch-execute cycle will not be combined. This is an elementary form of

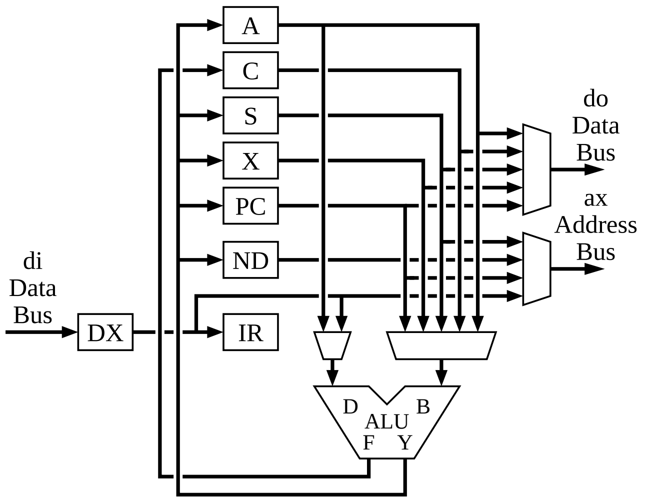

Figure5: The nod4.1 data path.

pipelining, though we normally do not think of it that way. The arrangement shortens the overall read path and allows for a higher clock frequency.

The controller is essentially a state machine that uses status information to direct the actions of the data path to provide the desired cycle-by-cycle behavior. Based solely on the controller it is possible to cause the behavior of the data path to be like that of an entirely different processor. The processor controller is microcoded to both emphasize how the fetch-execute cycle behaves like an interpreter, and also to provide opportunities to experiment with the implementation. Students can also consider the performance of instructions by counting microcode instructions.

Figure 6 outlines the microcode by representing related blocks of code as states. The actual microcode is listed in the nod4.1 implementation document[5]. Starting at init, the program start address or PSA is loaded in the PC register. In fetch1 the opcode is fetched from memory. In fetch2 the opcode is decoded and a second byte is fetched from memory. With the opcode and the following byte fetched, implied and immediate type instructions can be executed. Direct and indexed instructions access data at the corresponding effective address or EA. The access-EA code calculates the effective address (EA) as necessary and reads data from or writes data to the EA. Once the instruction is executed, as necessary the interrupt code prepares for an interrupt.

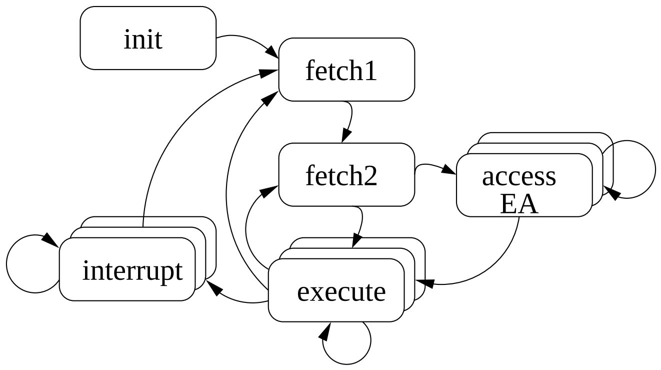

Figure 6: Microcode overview.

In starting at fetch1, the first instruction has two fetches performed. For immediate, direct, and extended addressing instructions, fetch2 obtains the operand. For implied instruction however, the fetch2 produces the next opcode so that most implied instructions treat the second fetch as a pre-fetch. In pre-fetching the next opcode, the current implied instruction or the next instruction can be thought of as executing in one less clock cycle.

The use of two fetches follows our motto of "*simple yet nontrivial*." The choice to arbitrarily fetch two bytes in sequence from memory in this fashion has less to do with implied instructions, and more to do with the rest. By immediately fetching a second byte, regardless of addressing mode, the microcode is simpler and no time is used to decide if a second fetch is required, so all instructions execute faster. The idea of pre-fetching and having implied instructions execute faster yet is a happy coincidence.

The downside is that the exact meaning of the PC register is less clear. Once fetch2 is complete, the PC register contains the address of the current opcode plus two, which could be the next instruction or something after that. Thus the PC is more of a fetch counter. Normally this is not a problem as we know the situations where the PC is expected to refer to the next opcode in memory. All jump instructions are two bytes long so that in executing a jump to subroutine (JSR) instruction, the PC refers to the return address, and will properly be pushed onto the stack.

The nod4 architecture has one vector for supporting exceptions. The lower five bits of the C register store the interrupt identifier (IID) value, where zero is reserved for the software interrupt instruction (swi). In invoking an interrupt, the previous completed instruction may not be two bytes long. In completing most implied instructions, the pre-fetching must be first be undone before jumping to the interrupt service routine (ISR), so that the correct return address is pushed onto the stack. The swi instruction in particular is one byte long.

Considering Student Feedback

Prior to nod3, our standard college course questionnaire asked students a number of detailed questions that provide students with opportunities to make comments. Based on these comments I discerned that many students felt the nod processor architecture was helpful but that writing a simulator program was too abstract. Because of the feedback I was motivated to have students actually implement nod series processors. More recently a questionnaire was first e-mailed out to students who were exposed to the versions up till nod4.0. From a typically small course, a smaller response was received. Two studied nod2, three studied nod3, and two studied nod4.0. Despite that two of the students wrote the nod2 simulator program, their responses were very similar to those who implemented at least a significant part of a nod3 or nod4.0 processor. Nine of the students first exposed to nod4.1 were given the questionnaire in class. Some questions asked students to reply with a numerical answer and others asked for a statement. Students were also welcome to make any comments they wished.

The questions are listed below along with the average values. Questions 2 and 4 are similar and are a general gauge of student satisfaction with the nod series processors. In the ‘Until’ group, other than one student, who in question 2 indicated indifference, all students at least agreed and on average moderately strongly agreed that the nod series is helpful and helped their understanding. Only one student in the ‘nod4.1’ group disagreed in question 2 and the rest agreed at some level.

Questions 5 and 6 are contrasting in that I was concerned with the size of the project. Question 5 asks if there is educational value in the exercise. Of all the questions, number 5 has the highest score in both groups. In contrast, question 6 proposes that students be given a completed processor to study. I am surprised by the cool response in the ‘Until’ group and that on average there is a slight disagreement with the question. Two students gave similar comments that their learning resulted from having to complete either the simulator or an actual implementation. While the ‘nod4.1’ group differs, the question 6 average is also lowest.

Undergraduate computer engineers take this course, and it appears that implementing such a microprocessor is welcome and may account for some of the cool reply to question 6. Perhaps having a completed system available would make nod4 more accessible to other students as well. In particular, straight electrical engineering students and computer science majors may benefit.

In examining the comments made to questions 2 and 3, students indicated that yes, the nod series processors helped in their understanding of what a data path is, what micro-coding is, and what the fetch-execute cycle is and what instruction encoding and decoding is. One student asked that some method be used for nod4 to introduce larger computers. Another student commented that if additional material is added to the course, then nod4 would probably find even more use. To summarize, the feedback for the nod series processors is positive.

0 Disagree Strongly	1 Disagree	2 Neutral or Indifferent	3 Agree	4 Agree Strongly

	Question	Until nod4.0	First nod4.1
1.	Which nod series processor did you study?	– –	– –
2.	Overall the nod processor helped to introduce computer architecture related topics and is a benefit to the ECE335 class in itself. Also list a topic that nod4 helped your understanding	3.43	3.33
3.	Is there a computer architecture topic that nod4 can be used to better introduce?	– –	– –
4.	The nod4 processor implementation or architecture helped me to better understand the internals of microprocessors and the fetch-execute cycle	3.57	3.11
5.	The nod3 and nod4 processors involved having students implement a significant part of a microprocessor. Do you see this exercise as having educational value?	3.71	3.67
6.	Suppose that rather than having students implement a complete4 processor, a completed processor was provided to students to study in detail. Having such a completed would further improve my understanding of microprocessors.	1.71	3.00

Conclusion

The field programmable gate array (FPGA) and modern computer aided design tools provide new opportunities in teaching computer architecture. This paper presents a simple yet nontrivial Von Neumann style computer architecture called nod4 and a soft core implementation called nod4.1 that is suitable for an undergraduate course in computer architecture. As of this writing the nod series processors have reached a level of maturity where few if any modifications are planned. The processor architecture itself is worthy of study, including such features as subroutines, stack relative addressing, interrupts, and conditional branching. The architecture and implementation documents are written so that several options are possible for introducing nod4 into the classroom curriculum. In particular, students may investigate the nod4.1 processor or implement the processor themselves. It is also possible to present the processor architecture entirely without the implementation.

Bibliography

1. M. Morris Mano and Charles Kime, Logic and Computer Design Fundamentals, third edition, copyright 2003 by Prentice Hall.

2. Andrew S. Tanenbaum, Structured Computer Organization, copyright 2006 by Pearson Education, Inc.

3. J. Hill, nod4 Architecture, copyright Sep 25, 2006, http://uhaweb.hartford.edu/jmhill/projects/nod4/index.htm

4. J. Hill, nod4.1 Memory System, copyright Sept. 20, 2007.

5. J. Hill, nod4.1 Implementation, copyright Oct 23, 2007.

6. Xilinx, MicroBlaze Processor Reference Guide, Embedded Developers Kit EDK 9.2i, UG081 (v8.0)

7. Tom Swan, Mastering Turbo Assembler, second edition, copyright 1995 by Tom Swan, published by Sams.

Biographical Information

Dr. Jonathan Hill is an assistant professor in the College of Engineering, Technology, and Architecture (CETA) at the University of Hartford, Connecticut. He received the Ph.D. and M.S. from Worcester Polytechnic Institute and B.S. from Northeastern University. Previously an applications engineer with the Networks and Communications division of Digital Corporation, his interests involve embedded microprocessor based systems.