AC 2008-2081: USING AN EDUCATIONAL MICROPROCESSOR ARCHITECTURE AND FPGA IMPLEMENTATION TO INTRODUCE INTERRUPTS

Jonathan Hill, University of Hartford

Dr. Jonathan Hill is an assistant professor on Electrical and Computer Engineering in the College of Engineering, Technology, and Architecture (CETA) at the University of Hartford, located in Connecticut. Ph.D. and M.S. from Worcester Polytechnic Institute (WPI) and Bachelor's degree from Northeastern University. Previously an applications engineer with the Networks and Communications division of Digital Corporation. His interests involve embedded microprocessor based systems.

Using an Educational Microprocessor Architecture and FPGA Implementation to Introduce Interrupts

Abstract

The use of interrupts is an important topic in the use of computers. Interrupts provide the means for a computer to quickly respond to significant real-world events. Unlike polling, which is suitable for interfacing slow peripherals, interrupts provide a more efficient means to interface with devices. Interrupts are generally considered to be advanced and unfortunately, the topic can be a challenge to present to students. In computer architecture in particular, the topic of interrupts is often overly abstracted, which can make learning about interrupts difficult for students.

Following a *trigger* event, an interrupt service routine (ISR) is invoked to provide service. Such an event could be the arrival of data on a serial communications port, or a signal that the brakes in an automobile have been applied. The latency is the actual time for the system to respond. Interrupts are a critical part of the hardware-software interface. With a serial port, it is necessary to quickly process received data, to avoid it being overwritten by more recently received data. Likewise, with brakes there is a constraint on the time the controller has to respond to the external event. Interrupts are also useful in debugging software.

To go beyond the discussion of a generalized abstraction, it is necessary to consider an actual microprocessor. This paper discusses the use of the nod4 soft core microprocessor to introduce the concept of interrupts to undergraduate students. But to strike some kind of balance, nod4 exception handling is fairly generic roughly following the discussion in Tanenenbaum[1] and it also borrows concepts from other processors as well. Apart from the implementation, nod4 is classic accumulator based Von Neumann style architecture and as such is similar to many other processors. This implementation is called soft as it is implemented with a Field Programmable Gate Array (FPGA). The FPGA and modern computer aided design tools provide new opportunities in teaching computer architecture. The grand vision is expressed neatly in the motto that the nod4 processor is a simple yet non-trivial processor designed to be a tool for teaching introductory computer architecture principles to undergraduates.

Introduction

From a computer user's point of view, interrupts simply provide a means for a computer to quickly respond to significant real-world events. From a programmer's point of view, interrupts provide a means for the hardware to almost magically invoke a subroutine to service a device as necessary, freeing up resources that might otherwise be used for polling. Unlike polling, which is generally suitable for interfacing slow peripherals, interrupts provide a more efficient means to interface with devices. Traps such as the software interrupt instruction (swi) are similar to interrupts and are invoked with the same mechanism. The swi instruction is typically used to implement break points for use in program debugging.

In computer engineering computers are thought of as being layered, with physical hardware representing the lowest layer. Invoking an interrupt starts deep within the hardware, working up through the layers, forcing a sudden jump in execution of the machine code instructions that eventually reaches the operating system and application software. Interrupt latency is the overall time for the system to respond. In this way interrupts involve all aspects and all layers of the computer system.

By involving all layers, interrupts should be a useful pedagogical tool. Unfortunately, teaching such a comprehensive topic can be a challenge. In some computer architecture courses the topic of interrupts is often overly abstracted, making learning difficult. In other courses interrupts are introduced in the context of an existing system, however many the details are hidden so that we resort to inductive technique which causes the topic to appear distant.

To go beyond abstract generalities and make the topic of interrupts reachable it is useful to consider an actual system. While on one hand students need an opportunity to closely study such a system, modern systems are exceedingly complex. To strike a balance, the nod4 exception mechanism is designed to be simple and fairly general. Exception handling roughly follows the discussion in Tanenenbaum[1] and is also inspired by other processors, in particular, the PIC[2], 8086[3], PowerPC-405[4], Microblaze[5], and the 68HC11[6].

I have found that a deductive approach tends to be more effective in teaching new concepts than an inductive approach. Interrupts come with the nod4 implementation as *built-in*, no additional functionality is needed, so that students can construct the nod4 system without knowing anything about interrupts. Students are provided with VHDL modules that they use to make symbols and use schematic capture to implement the processor themselves. In fact, the topic of interrupts can be presented entirely at the instructor's discretion. Once constructed, the implementation provides students with hand-on opportunities to investigate interrupts.

The notion of interrupts can first be presented in steps, with increasing levels of focus. First interrupts can be presented from an abstract point of view using a text book such as Tanenbaum[1]. Next, the nod4 architecture provides a more deductive approach, where students can study an example assembly language program that uses interrupts. The nod4 project[8] is a resource and many of the documents also include homework questions. Finally, students can study an actual implementation of nod4 and see all the fine detail, *clock cycle by clock cycle*, showing exactly how interrupts are invoked.

The nod4 Microprocessor Architecture

The name nod4, pronounced *"node four"* refers to a computer architecture developed for use in undergraduate projects involving computer architecture. The author[7] provides an outline and the complete details are in the project website[8]. The architecture is entirely eight bit in that all registers are eight bits in length. The processor supports unsigned arithmetic and has several notable features such as subroutines, stack relative addressing,

interrupts, and conditional branching. The conceptual difference between computer *architecture* and a computer *implementation* is a key lesson for students to learn.

The nod4 *architecture* outlines aspects of the computer that an assembly language programmer is aware of. The nod4 architecture has an 8-bit data path and an 8-bit address bus. The architecture comprises the CPU registers, the memory map and memory mapped devices, as well as the instruction mnemonics and addressing nodes, as well as the interface to the exception handling mechanism. 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 should be less concerned with.

To express a program that makes use of the CPU registers and memory, it is convenient to have an assembly language. In writing assembly code we will be most concerned with *symbols* and *labels*. A symbol is a symbolic name for a value. A label is like a symbol, except that the value must be an address. The assembly language file format is broken into lines. A line is organized into as many as four possible fields. A comment can be inserted at the end of a line or an entire line can be a comment.

1. The left-most field contains a label, symbol, or a semicolon ';' used to start a comment line. Each label or symbol here is terminated with a colon ':'
2. The second field contains either an instruction mnemonic or a directive, which is a command to the assembler.
3. The third field contains either the operand for an instruction or data associated with an assembler directive. The use of square brackets '[ ]' means the contents of the given address.
4. 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).

A directive is a command intended for the assembler, rather than an instruction for nod4. The following are the directives:

```
         ORG  Address
```
Sets the point of assembly to 'Address'

```
symbol: EQU  value
```
The symbol is assigned the constant value

```
label:  FCB  val1,val2,…
```
Reserves a byte for each value in a comma separated list. The address of the first value is assigned to the label.

```
label:  RMB  n
```
Reserves n bytes but does not assign any specific values to memory. The address of the first byte is assigned to the label

The following summarizes the given assembly language format

```
; Here is a comment line
Symbol: Directive  Data    ; A comment
Label:  Mnemonic   Operand ; Another comment
```

Given such a program, an execution history is a listing of instructions, with relevant values. The execution history shows, based on an understanding of the architecture, how a program is expected to execute, instruction by instruction.

Architectural View of Interrupts

The term *exceptions* collectively refer to *interrupts* and *traps*. *Interrupts* are generally used to service *devices* and *traps* are generally invoked by a software related event. As with most microprocessors, nod4 uses one mechanism to handle both interrupts and traps and unfortunately, quite often the language can become blurry. Details of nod4 exception handling are fairly generic and generally follow the discussion in Tanenenbaum[1] and also borrow concepts from other processors, including the PIC16 architecture[2], the Intel 8086[3], the PowerPC 405[4], Microblaze[5], the Motorola 68HC11[6], and others.

The following is an outline of interrupts as seen in the nod4 architecture. Interrupts are said to be *maskable* as the I-flag in the condition code register is used to *enable* interrupts. As with the 68HC11, the interrupt vector is meant to be permanently written in ROM along with the program start address. As with the 8086, invoking an interrupt only preserves the PC and C registers, the handler is responsible for preserving any other CPU registers that are used. Also like the 8086, an interrupting device provides an identifier code. But like the Microblaze and PIC processors only a single vector is

provided so that the exception handler is also responsible for dispatching execution to the corresponding interrupt handler.

An interrupt is *triggered* at the completion of a machine code instruction when the I-flag and the IRQ interrupt request signal is high. For interrupts the identifier code (IID) is read from the interrupting device. For traps the IID code zero is used. The upper 3 bits of the IID are masked or forced to be zero and written to the condition code register so that only the lower five bits are used. The PIA value is the address of the first instruction in the exception handler. The PIA value is stored in memory at address $01 and is written to the PC register to invoke the handler.

- The programmer is responsible for assigning the programmer interrupt address (PIA) value at address $01.
- Peripheral devices are configured during system initialization.
- Following system initialization a bit-wise OR instruction sets the I-flag in the condition code register, enabling interrupts.
- A peripheral device asserts IRQ to request service.
- The interrupt mechanism is triggered at completion of the current instruction.
- The interrupting device identifier (IID) code is read.
- The address of the next opcode and C register value are pushed into the stack
- The IID is bit-wise anded with $1F and put into the condition code register
- The PIA value is read from memory and put in the PC register, which effectively invokes the exception handler.

Given that only the PC and condition code registers are pushed into the stack, the assembly language programmer is responsible for preserving and later restoring other registers used by the exception handler. With the PIA as the only vector for exceptions the IID value can be used to dispatch execution to a corresponding interrupt service routine. Traps will have an IID value of zero.

- Before returning from the exception handler, service must be provided in a way so that the requesting device no longer asserts the IRQ signal.
- The 'rti' instruction returns execution from the interrupt handler by popping the condition code register and PC register values from the stack, respectively.

First Example Program

The following example program uses a real-time clock to generate interrupt requests, causing an LED display to count in binary. The period of the real time clock peripheral is defined in hardware and otherwise is not configurable in software. The device is periodic in that it regularly asserts the interrupt request, whether or not the request is cleared. The exception handler is responsible for clearing the request.

The following is true with performing a read from the control/status register (RTCTL):

- bits 7 to 2 - Read as '0'
- bit 1 - RTIE - The device IRQ enable flag, when high requests can be made
- bit 0 - RTF - The real time flag (RTF) state

In performing a write to the control/status register the following applies:

- bits 7 to 2 - Writes are ignored
- bit 1 - RTIE - The device IRQ enable flag, when high requests can be made
- bit 0 - RTFC - Writing a '1' clears the real time flag (RTF)

The following program rtex1.asm demonstrates the real time clock. Here we assume that the period is 100ms, corresponding to a 10Hz rate. The labels RTCTL and LEDS refer to the address of each corresponding device. The full details of the mnemonics and assembly language encoding are in the Architecture document[8]. This example only uses implied (IMP), immediate (IMM) and direct (DIR) addressing.

System initialization is performed starting with the *load stack register* (lds) instruction which initializes the stack. Next, the *load* and *store register A* instructions (lda and sta) clear the count value stored at Count. The square brackets mean the *contents of the given address*, which here indicates direct addressing. The next lda and sta instructions enable and clear the real time interrupt request signal, completing system initialization. The orc instruction performs a bitwise OR with the C register, enabling the exception mechanism.

The exception handler starts at the address ISR. Note that with only one source of exceptions, there is no need to examine the IID to dispatch execution to this interrupt service routine. Likewise, there is only one place in the main program where interrupts can occur, that is just before or after the jump-always (jmp) instruction executes. Given that the jmp instruction forms a very tight loop to itself, the exception handler need not preserve the A register value which it uses. In more practical examples, the ISR must preserve and restore the registers that it uses.

At the top of the ISR the real time interrupt request signal is cleared the same way as during initialization. Next, the load, add, and store A register (lda, adda, and sta) instructions increment the count and update the value displayed by the LEDs. Finally, the return from interrupt instruction (rti) returns execution back to the main program code. The rti instruction is different from a return from subroutine (rts) instruction as the condition code register must also be pulled from the stack.

```
; rtex1.asm - Jonathan Hill - Fall 2007
; Demonstrate the real-time clock interrupt
RTCTL: EQU  $FD
LEDS:  EQU  $FC

       ORG  $00       ; Point of assembly
       FCB  Start,Isr ; PSA,PIA

Start: lds  $FC       ; (IMM) Init stack
       lda  $00       ; Load zero and
       sta  [Count]   ;  (DIR) clear the count
       lda  $03       ; Load value to
       sta  [RTCTL]   ;  enable device
       orc  $20       ; Enable interrupts

Done:  jmp  Done      ; Hang out here

Isr:   lda  $03       ; Load value to
       sta  [RTCTL]   ;  reset flag
       lda  [Count]   ; Get count and
       adda $01       ;  increment value
       sta  [Count]   ;  store count
       sta  [LEDS]    ;  write to LEDs
       rti            ; (IMP) Done for now

       ORG  $C0
Count  RMB  1         ; Counter value
```

The nod4.1 Microprocessor Implementation

In contrast, the *implementation* is how the computer is actually constructed. Students are provided with VHDL modules that they use to make symbols and use schematic capture to implement the processor themselves. In this way students are exposed to register transfer level and higher level schematics. Constructing the given nod4.1 implementation is similar to drawing schematics using MSI parts. While students are not expected to know VHDL, they are welcome to investigate what the VHDL modules contain. The nod4 architecture strives for clarity and is transparent so that nothing is hidden from the student.

Implementations of nod4 thus far use microcode in the controller to provide a clear way to investigate the cycle by cycle behavior of the implementation. Figure 1 gives a fairly close conceptual outline of the fetch-execute cycle by representing related blocks of microcode as states. Starting at init, the program start address or PSA is read from memory and is loaded in the PC register.

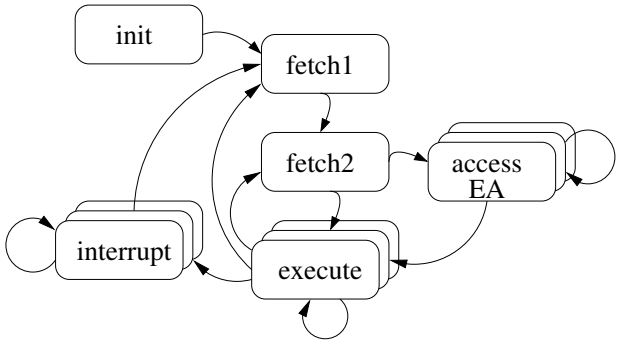

Figure 1: Microcode overview

The fetch1 state reads an instruction opcode and fetch2 reads a second byte from memory. At this point, 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 states calculate the effective address (EA) as reads or writes data as necessary. Once executed, as needed the interrupt states prepares for an interrupt.

Immediate, direct, and indexed addressing instructions use both bytes fetched from memory. Most implied instructions treat the second fetch as a pre-fetch of the following instruction. In pre-fetching the following opcode, the current implied instructions appear to execute in one less clock cycle. The choice to arbitrarily fetch two bytes in sequence from memory in this fashion actually has less to do with implied instructions and more to do with the rest. By immediately fetching a second byte, regardless of addressing mode, then no time is used to decide if a second fetch is required, so that all instructions execute faster. The idea of pre-fetching so that 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 register has the role of a fetch counter. Normally this is not a problem as we know the situations where the PC register 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 register will refer to the return address, and will properly be pushed onto the stack.

In invoking an interrupt, the previous completed instruction may not be two bytes long. In completing an implied instruction, the pre-fetching must be first be undone before jumping the machine code to the exception handler, so that the correct return address is pushed onto the stack. Figure 2 illustrates the flow through microcode to invoke the exception handler. The states with vertical internal bars are the entry points.

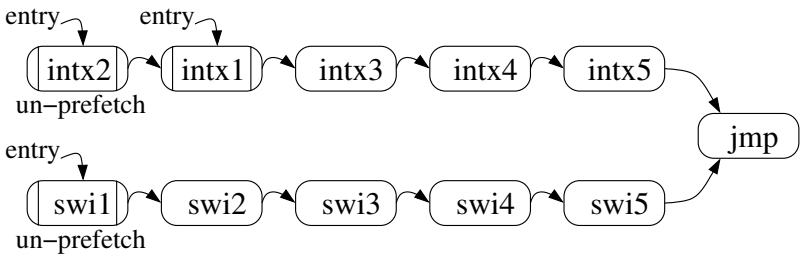

Figure 2: Microcode flow invoking the exception handler

State intx1 is entered after an immediate, direct, or extended instruction is completed. State intx2 is entered after a pre-fetch-type implied instruction is completed. State swi1 is entered when the swi instruction is executed. States intx2 and swi1 un-pre-fetch the following opcode as necessary by decrementing the PC register. Hence, intx1 is entered when a non-pre-fetch-type instruction is completed. Full details of the microcode are in the Implementation document[8].

Interrupt Latency

As outlined earlier, interrupts provide the means to quickly respond to significant real-world events, so that interrupt latency is a critical parameter. Following a request, the *worst case interrupt latency* is the worst case delay till the corresponding ISR actually provides service. Interrupt latency is the sum of several delay values. In many real time systems the interrupts are disabled at least some of the time. Interrupts may be disabled during execution of a critical section, and are disabled when the exception handler itself is invoked. Once interrupts are enabled, the exception handler must wait for the current instruction to complete. As outlined in the previous section, the mechanism requires six clock cycles to invoke the exception handler. Once invoked, the exception handler provides a decision tree to dispatch execution to the corresponding ISR.

With a 50MHz clock, each bus cycle consumes 20ns. The worst case interrupt latency is the sum of the longest number of cycles for which interrupts are disabled, the largest number of cycles to execute an instruction, the number of cycles to invoke the exception handler, and finally the number of cycles to dispatch execution to the ISR which provides service. Other than the longest number of cycles for which interrupts are disabled, the microcode provides a cycle by cycle account of the nod4 processor behavior.

Handling Multiple Interrupt Sources

In considering the handling of multiple interrupt sources, several related topics arise related to interrupt identifier codes (IID), interrupt priorities, and dispatching execution to the corresponding handler. First off, while the processor has only one interrupt request input (IRQ) the IID code has five bits. With the IID code zero reserved for the swi instruction, there can be as many as 31 interrupt request sources. Given that a device produces an IID to identify the source, it is possible for a device such as a UART to produce one of several IID codes, based on the service being requested.

With only one exception handler present, to handle multiple interrupts, the programmer is responsible for providing code to dispatch execution to the corresponding ISR, based on the received IID value. Such code is basically a simple decision tree. Next consider the actual devices. A single device connects with signals as shown in Figure 3. The device asserts IRQ as necessary to request service. When the processor is able, it acknowledges the request by asserting the IRA signal. In the same bus cycle the device puts its IID code into the di memory system data bus.

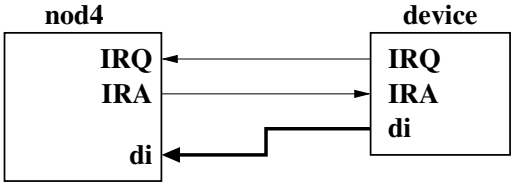

Figure 3: Single interrupt requestor

In systems with two or more devices, the interrupt priority encoder (IPE) establishes priorities between devices. In receiving multiple requests, the IPE determines which device will receive service first. In Figure 4, the IPE receives interrupt requests from two devices. Inside the IPE an OR gate combines the device requests, producing the IRQ signal passed to the processor. When the processor asserts IRA, the IPE passes the acknowledgement to the highest priority device making a request. It is given here that device0 has priority over device1. As with the case with one device, the acknowledged device uses the di memory system data bus to send its IID code to the processor.

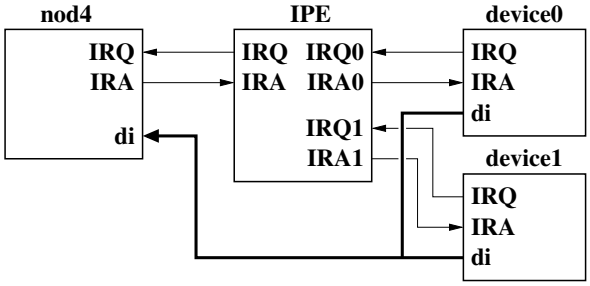

Figure 4: Two interrupt requestors

Summary

The nod4 processor architecture and implementation provide opportunities for students to investigate interrupts. Details of interrupts with nod4 are fairly generic. The concept roughly follows the discussion in Tanenbaum[1] and also borrows concepts from several other commercial processors. The nod4 architecture provides students with a means to investigate interrupts from an assembly language programmer's point of view. Students are provided with VHDL modules that they use to make symbols and use schematic capture to implement the processor themselves. In this way students are exposed to register transfer level and higher level schematics. With nod4 implemented, students can investigate the cycle by cycle behavior of how interrupts are actually invoked.

Bibliography

1. Andrew S. Tanenbaum, Structured Computer Organization, copyright 2006 by Pearson Education, Inc.
2. Peatman, John, Design With PIC Microcontrollers, copyright 1997 by Prentice Hall
3. Walter Triebel and Avtar Singh, The 8088 and 8086 Microprocessors: Programming, Interfacing, Software, Hardware, and Applications, 4th Edition, copyright 2002 by Prentice Hall
4. PowerPC 405 architecture, documents available from http://www.xilinx.com/ipcenter/processor_central/embedded/architecture.htm
5. Microblaze architecture documents available from http://www.xilinx.com/products/design_resources/proc_central/microblaze_arc.htm
6. Peter Spasov, Microcontroller Technology, the 68HC11 and 68HC12, copyright 2004 by Pearson Prentice Hall.
7. Jonathan Hill, "*Microprocessor Architecture with FPGA Implementation for Undergraduate Computer Architecture Courses*," Computers in Education, published by the ASEE, Jan.-Mar. 2008
8. Jonathan Hill, "*Project page for nod4*," http://uhaweb.hartford.edu/jmhill/projects/nod4/index.htm