The art of progress is to preserve order
amid change and to preserve change amid order.

Alfred North Whitehead

CHAPTER 1: INTRODUCTION

1.1 A Technology Overview

The past three decades has introduced technology that has radically changed the way in which the world is analyzed and controlled. In the early 1970s, developments in computer architecture and IC fabrication begot the first microprocessor, introduced by Intel Corporation, the 4-bit 4004. A later by-product of the development of the microprocessor is the less well-known and abundantly used microcontroller. These devices are responsible for smart VCRs, clock radios, washers and dryers, video games, telephones, microwaves, TVs, automobiles, toys, vending machines, copiers, elevators, irons, and other intelligent products that are "programmable." [1]

Programmable Logic Devices (PLDs), like Field Programmable Gate Arrays (FPGAs) have a long-standing reputation for being slower, more expensive, less flexible and more time consuming to use for developing embedded systems. In the past,

Application Specific Integrated Circuits (ASICs) have emerged as a cost-optimized, though still expensive, custom solution for bringing programmability and other benefits to a design. In the past couple of years, cutting edge process technologies have significantly reduced die sizes. As a result of these advancements, FPGA vendors are now able to provide a low-cost solution to ASIC designers.

Microprocessors must be used with external resources including RAM, ROM, I/O ports and timers to make them functional. A microcontroller has a CPU in addition to RAM, ROM, I/O ports, and timers all on a single chip.[1] Microprocessors and microcontrollers are widely used in embedded systems. Figure 1.1 shows a microprocessor system contrasted with a microcontroller system.

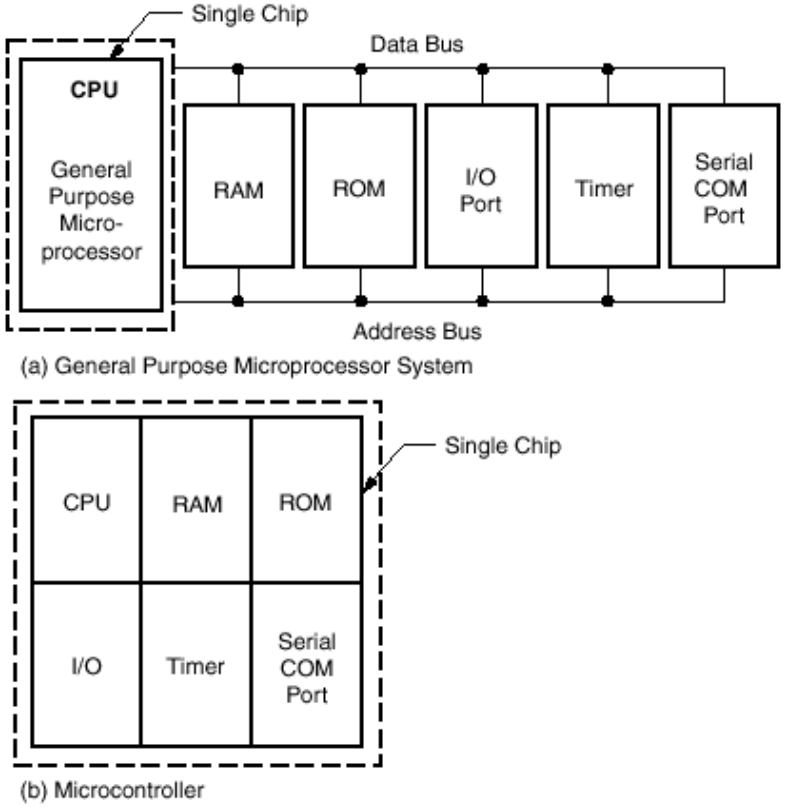


Figure 1.1: A microprocessor contrasted with a microcontroller. [*Source:* Dhir and Mousavi]

Customarily, an embedded system uses a microprocessor (or microcontroller) to perform a single task. Critical considerations for using microprocessors or microcontrollers to produce a system include the amount of space occupied (number of logic cells), power consumption, price per unit, computing power and amount of development time required for integration.

1.2 Hardware/Software Co-design

Traditionally, an embedded system requires the cooperative design of hardware and software. One of the goals of co-design is to shorten the time-to-market while reducing the design effort and costs of the designed products. Therefore, the designer has to take advantage of the target architecture using both software and hardware. Many consider using processors an advantage because software executed by the processor is more flexible and cheaper than a design made completely of hardware. The flexibility of designing parts of the system in software allows late design changes and simplifies debugging. Furthermore, the software may be reused by porting it to other processors. This may reduce the time-to-market and design effort. In the past, it was much cheaper to use microprocessors compared to developing ASICs, because of their high-volume production.[2]

Although software components introduce a degree of simplification with available compilers and code that can potentially be reused, speed is sacrificed. Replacing software with hardware increases the speed of the input-to-output transfer, however, hardware solutions require logic gates; more chip space, and therefore, additional costs. In the past three years, aforementioned advancements in technology and processes have

created affordable FPGAs with large densities that are increasing every 6 months. Designs that would be extremely expensive to implement in hardware and realize the tremendous speed advantage are becoming significantly cheaper. Figure 1.2 shows the reduction in costs and increase in chip densities of a market leader in FPGA design and production, Xilinx Corporation.

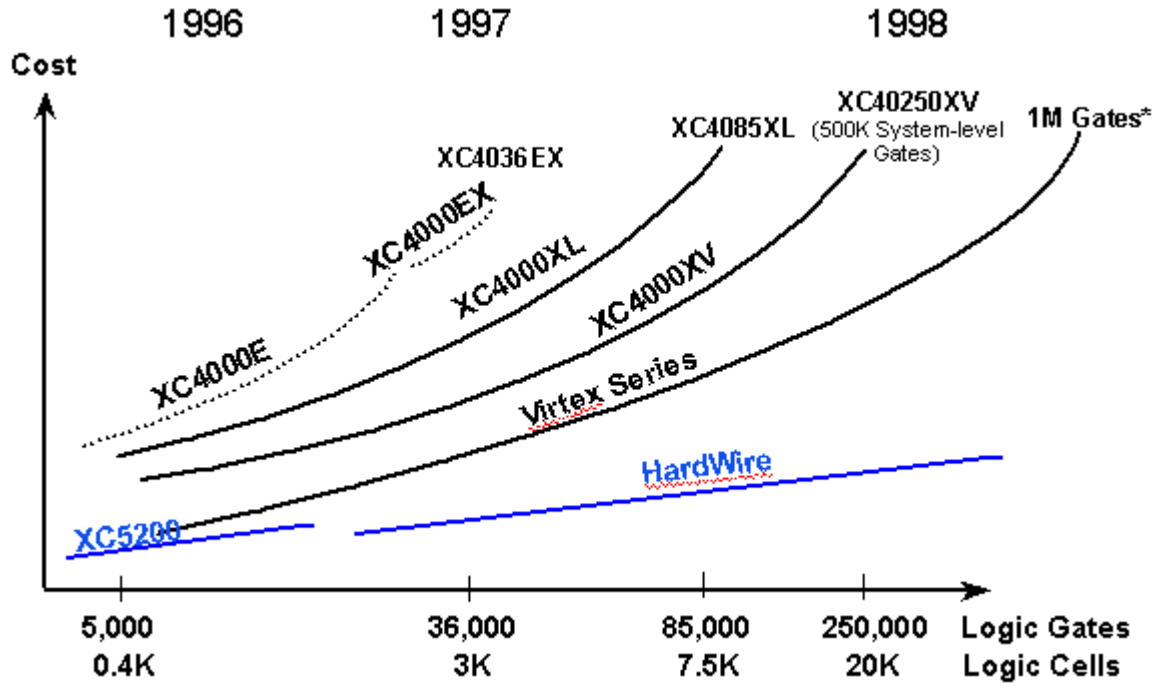


Figure 1.2: A cost reduction and increase in chip density and the number of gates available on FPGA chips provided by Xilinx Corporation. [*Source:* Xilinx, Inc.]

Notice that FPGAs with a capacity of approximately 36,000 gates cost the same in 1997 as a chip that has 1,000,000+ gates available in 1999. Ralf Niemann writes,

> Co-design is an interdisciplinary activity, bringing concepts and ideas from different disciplines together, e.g. system-level modeling, hardware design and software design.[2]

1.3 The Need for a Simplified Co-design Process

The hardware/software co-design process, shown in Figure 1.3, introduces a high-risk step, determining which components will be designed in hardware and which will be designed in software. Furthermore, the microprocessor selected to design the software components must support the formulated software requirements, minimize hardware costs and be *sufficiently* flexible for anticipated design changes that may be introduced during development. In general, these critical factors introduce enough potential risks that many co-designed systems have resulted in over-budget, lengthy and unsuccessful projects due to the lack of ability for the hardware to support unforeseen software requirements and vice versa.

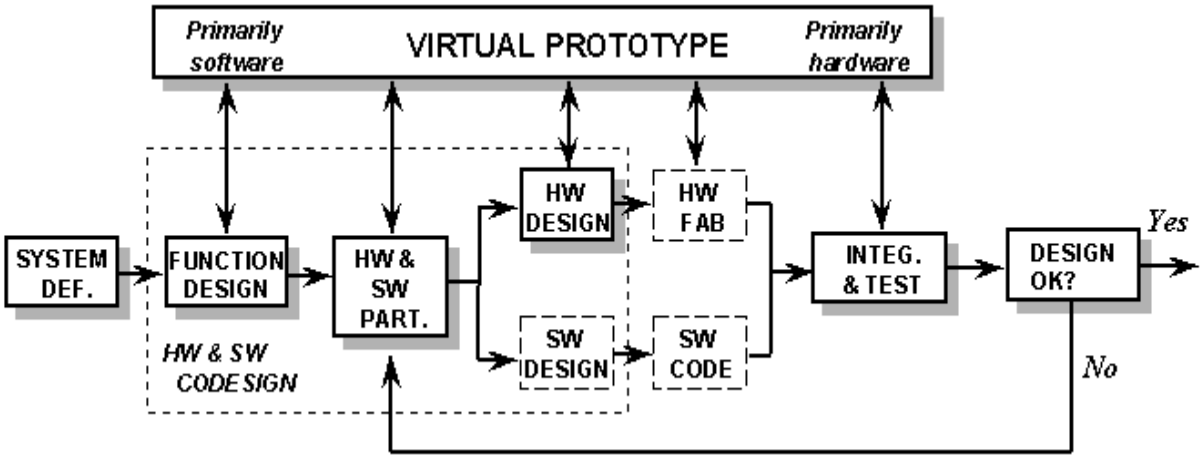


Figure 1.3: The hardware/software co-design process with rapid prototyping. [*Source:* RAASP model for hardware/software prototyping]

The co-design process requires the use of heterogeneous design methodologies. Many design processes, for example, the Unified Software Development Process, aren't completely sufficient for hardware/software co-design.

The Unified Process is an iterative and incremental process that asserts designers to perform several iterations of Requirements, Analysis, Design, Implementation, and Testing. It is also Architecture-Centric using the architecture to understand the system, organize development, foster software reuse, and evolve the system. While developing software under the Unified Process, developers perform iterations concerning the use cases that introduce the highest element of risk that has not been specifically addressed and interface with the boundary actors throughout the process. The system architecture is also developed iteratively. If a change in a use-case or unforeseen requirement surfaces, developers identify the required classes that the change affects and make necessary design changes to accommodate the use-case changes. For hardware/software co-design, a change in a use-case or nonfunctional requirement, such as a timing constraint, may be beyond the capability of the selected microprocessor. This may require the selection of a new microprocessor with the added capability, within cost constraints, to perform the necessary operations at the required speed to realize the change and possibly require software programmers to retrace all of their code and port it to the new microprocessor. In some cases, significant changes to the original architectural plan are required.[3] For software development, this process minimizes risk and has proven to be successful in many large, requirement intensive projects.

The step in the co-design process introducing the most risk is the hardware/software-partitioning step. In this phase, developers decide which components of the overall system architecture will be implemented using software and which will be hardware. It is this step that results in the specifications for the microprocessor that will

be used which, in turn, begins the platform specifications for the software developers. Incidentally, it is also the step that frequently is revisited as timing constraints and requirements present changes that the system needs to accommodate. Changes at this juncture can lead to significant re-development of software and/or hardware.

1.4 A Proposed Solution

What is a fundamental difference between hardware/software co-design and software development? Architecture. In most software development environments, code written in high-level languages are compiled to a fixed instruction set that remains compatible with each new release of a faster, more recent release of the architecture. For hardware/software co-design, the entire system could be developed with components in software and components in hardware. The developers must design the architecture *before* the development of the system can begin. Generally, for software programs, particularly on a PC or mainframe, more functionality requires more disk space, possibly more RAM and more programming time that typically is linearly related to the programming time of the system.

In theory, the best time to actually decide which components can bear to be designed in the less-optimized, slower software environment and which components require the speed of the hardware is after the software has been written and the number of clock cycles required for each function can be computed. The software, however, can be written only after a determination has been made regarding the type of microprocessor on which the software will run. The microprocessor, on the other hand can be selected only

after considering the minimization of implementation cost by selecting a minimal function set required for the microprocessor by the software.

An effective solution to this paradox would be to eliminate packaged microprocessors. In 1994, Andŕe DeHon from the Artificial Intelligence Laboratory at the Massachusetts Institute of Technology included in his paper given at an IEEE workshop on FPGAs for Custom Computing Machines that "for broader application, future microprocessors should dedicate a portion of their silicon real-estate to reconfigurable logic… A single reconfigurable microprocessor design can serve as the principal building block for a wide range of applications including personal computers, embedded systems, application-specific computers, and general- and special-purpose multiprocessors." [4] DeHon's idea for the future of FPGAs and microprocessors was for a microprocessor to contain reconfigurable gates on its chip. Several groups have explored developing unified hardware development processes and tools including formal specification languages such as LUSTRE, KRONOS, POLIS, SIGNAL, REACTIVE C, Synchronous Language (SL), LOTOS, and SDL. If the co-design problem were reduced to a microprocessorless, common platform, co-design would resemble software development and software engineering processes such as the Unified Software Development Process alone could be used to guide the development of a hardware/software system.

References

1. Dhir, A., Mousavi, Saeid, "High-performance Spartan-II 8-bit Microcontroller Solution," *Xilinx White Paper* number 114, version 1.0, March 16, 2000.

2. A general overview of the basic co-design process is given by Ralf Niemann, PhD., from the University of Dortmund, Germany, at *http://ls12-www.informatik.uni-dortmund.de/~niemann/codesign/codesign.html*; Internet; Accessed September 2000.

3. Jacobson, Ivar, Booch, Grady and Rumbaugh, James, The Unified Software Development Process. Addison Wesley Longman, copyright 1999, Reading, Massachusetts.

4. DeHon, Andŕe, "DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century," Artificial Intelligence Laboratory at Massachusetts Institute of Technology, April 1994.

5. Jantsch, Axel, *et al*, "A Software Oriented Approach to Hardware/Software Codesign," International Conf. on Compiler Construction, CC-94, pp. 93 - 102, Edinburgh, Scotland.

6. Arpnikanondt, Chonlameth and Madisetti, Vijay K., "Constraint-Based Codesign (CBC) of Embedded Systems: The UML Approach," Center for Signal & Image Processing (CSIP) Georgia Tech, December 12, 1999, Yamacraw Technical Report #: YES-TR-99-01.

The important thing in science is not
so much to obtain new facts as to discover
new ways of thinking about them.

Sir William Bragg

CHAPTER 2: A SIMPLIFIED RECONFIGURABLE MICROPROCESSOR CORE

2.1 The Microcontroller

Most traditional microprocessors can be categorized as having either a Complex Instruction Set Computer (CISC) architecture or a Reduced Instruction Set Computer (RISC) architecture.[1] Both of these architectures involve a set of registers and multiple addressing modes. A simpler architecture that is easier to implement in an FPGA is a stack-based processor in which all arithmetic and logical operations are performed on the top elements of a data stack.

Forth is a programming language invented by Chuck Moore in the late 1960s while programming minicomputers in assembly language. His idea was to create a simple system that would allow him to write many more useful programs than he could by writing his programs in assembly language. The essence of Forth is simplicity -- always try to do things in the simplest possible way. Forth is a way of thinking about

problems in a modular way. It is modular in the extreme. Everything in Forth is a word and every word is a module that does something useful. There is an action associated with Forth words. The words execute themselves. In this sense they are very object-oriented. Words are sent parameters on the data stack and told to execute themselves. In return, the answers are placed on the data stack; a black box approach.

Forth has been implemented in a number of different ways. Chuck Moore's original Forth had what is called an *indirect-threaded* inner interpreter. Other Forths have used what is called a *direct-threaded* inner interpreter. These inner interpreters get executed every time you go from one Forth word to the next; i.e. all the time. A unique version of Forth called *WHYP* (pronounced *whip*) has recently been described in a new book on using the Motorola 68HC12 microcontroller in embedded systems.[2] WHYP stands for *W*ords to *H*elp *Y*ou *P*rogram. WHYP is what is called a *subroutine-threaded* Forth. This means that the subroutine calling mechanism that is built into the 68HC12 is what is used to go from one WHYP word to the next. In other words, WHYP words are just regular 68HC12 subroutines.

Inasmuch as Forth (and WHYP) programs consist of a sequence of words, the most often executed instruction is a call to the next word. This means executing the inner interpreter (NEXT) in traditional Forths, or calling a subroutine in WHYP. Up to 25% of the execution time of a typical Forth program is used up in calling the next word. To overcome this problem, Chuck Moore designed a computer chip, called NOVIX, in the mid-eighties which could call the next word (equivalent to a subroutine call) in a single clock cycle.[3] Many of the Forth primitive instructions would also execute in a single

clock cycle. The design of the NOVIX chip was eventually sold to Harris Semiconductor where it was redesigned as the RTX 2000.[4] Similar 32-bit Forth engines were also developed. In the late eighties Chuck Moore designed a 32-bit microprocessor called ShBoom that had 64 8-bit instructions and was designed to interface to DRAM.[8] Later Chuck Moore and C. H. Ting designed the MuP21 that has been described by Ting.[9] The WnX microcontroller described in this section is a simplified reconfigurable microprocessor based on ideas developed in these early Forth engines. It is designed using VHDL. Different versions, both simplified and extended have been implemented in a Xilinx FPGA at Oakland University.

2.2 The Simplified WnX Microcontroller

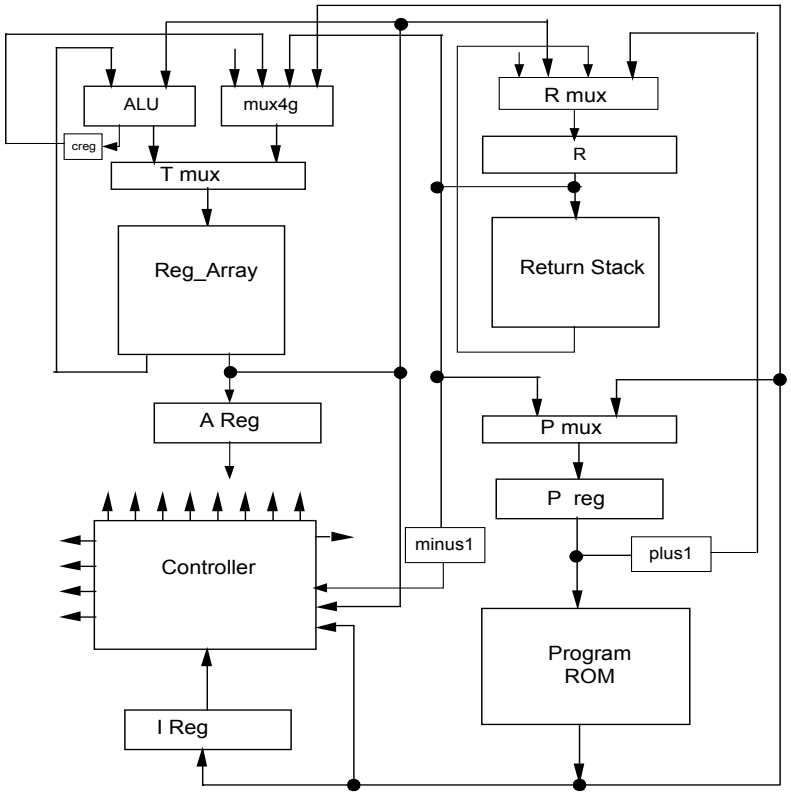


Figure 2.1: A block diagram of the simplified WnX microcontroller.

The WnX is a high-performance microcontroller that can be implemented to perform useful functions on an FPGA. The overall structure of the WnX is shown in Figure 2.1. The data busses in this figure are 8 bits wide and each instruction contains 8 bits. All busses in the microcontroller are defined as *generic* sizes allowing the WnX to be reconfigured to an *n*-bit microcontroller. In addition to the simplified microcontroller, the WnX has extended components such as a multiplier, divider, fuzzy-inference component and interrupt vector tables with an interrupt handler. In successive chapters, the WnX will be reconfigured for a specific co-design application. The WnX instruction set is given in Table 2.1.

Table 2.1: The Simplified WnX Instruction Set

Opcode	Name	Function
00	DUP	Duplicate T and push data stack.
01	DROP	Drop T and pop data stack.
02	SWAP	Exchange T and N1.
03	NIP	Drop N1 and pop rest of data stack.
04	ROT	Rotate top 3 elements on stack clockwise.
05	MROT	Rotate top 3 elements on stack counter-clockwise.
06	OVER	Duplicate N1 into T and push data stack.
07	TUCK	Duplicate T into N2 and push rest of data stack.
08	NOP	No operation
09	TOR	“To-R” Pop T and push it on return stack.
0A	RFROM	“R-from” Pop return stack R and push it into T.
0B	RFETCH	“R-fetch” Copy R to T and push register stack
10	LSL	Logic shift left T
11	ASR	Arithmetic shift right T
12	LSR	Logic shift right T
13	ROTR	Rotate right T (carry unchanged)
14	ROTL	Rotate left T (carry unchanged)
20	ZEROS	Clear all bits in T to ‘0’.
21	PLUS	Pop N1 and add it to T.
22	MINUS	Pop T and subtract it from N1.
23	ANDD	Pop N1 and AND it to T.
24	ORR	Pop N1 and AND it to T.
25	XORR	Pop N1 and AND it to T.
26	INVERT	Complement all bits of T.
27	ONES	Set all bits in T to ‘1’.
28	ZEQUAL	TRUE if all bits in T are ‘0’.
29	ZLESS	TRUE if sign bit of T is ‘1’.

Table 2.1 ***Continued***

2A	CTOT	Push carry bit to top of register stack
2B	1PLUS	Add 1 to T
2C	1MINUS	Subtract 1 from T
2D	MPP	Multiply partial product
2E	SHLD	Shift left T and N1 for division
2F	SUBC	If T > N1, subtract N1 from T and set N1(0) to '1'
40	LIT	Load inline literal to T and push data stack.
31	C@	Fetch the byte at addr T in RAM and load it into T
32	C!	Store the byte in N1 at the address T
41	JMP	Jump to inline address
42	JZ	Jump if all bits in T are ‘0’
43	JNC	Jump if carry is cleared
44	DRJNE	Decrement R and jump if R is not zero
45	CALL	Call subroutine
46	RET	Subroutine return
47	PUSHD	Load external value to T and push data stack
48	JNZ	Jump if all bits in T are not '0'

The data stack in the WnX is a register array designed with four multiplexers combined with four registers. The multiplexer for a stack register switches the output from any of the other three registers to its input as shown in Figure 2.2.

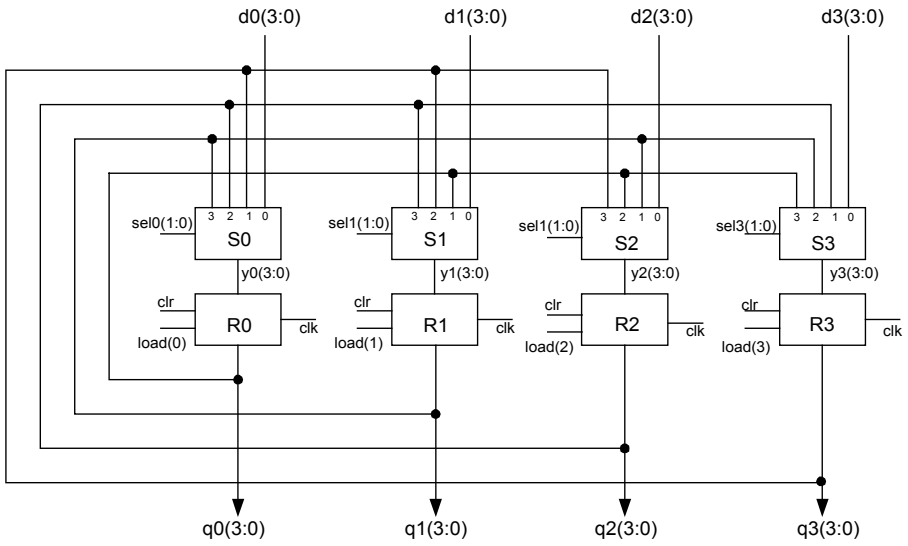


Figure 2.2: A register-array data stack.

The multiplexed-register stack provides flexibility to implement the stack instructions in one clock cycle. Although each of the stack registers have the capability, by design, to be loaded, the WnX only uses the load provided to the register designated as the top of the stack. This design provides an *n* x 4 register array data stack and an *n* x 16 return stack. The return stack is not an array-based stack since the flexibility to manipulate individual return stack data is not necessary to obtain single clock-cycle instructions. The input to the data stack is multiplexed from two sources, the ALU and a 4-input multiplexer. The ALU of the simplified microcontroller performs the operations shown in Table 2.2.

Table 2.2: WnX ALU Operations

ALU Select	Operation
"000"	all zeros
"001"	a + b
"010"	b - a
"011"	a and b
"100"	a or b
"101"	a xor b
"110"	not a
"111"	all ones

The *n*-bit inputs into the ALU are T and N, the top and second elements in the stack, respectively. The 4-input multiplexer provides an external signal, the carry out from the ALU, the top of the return stack, and the current value in the program memory addressed by the program counter. The program for the WnX is stored in a program ROM. The ROM is addressed by the program counter that can be loaded with a value from the return stack or from memory for return-from-subroutine instructions or instructions that may jump to an inline address. The return stack can be loaded with values from the top of the data stack and the program counter plus one. A control unit

and instruction register controls the WnX microcontroller. The microcontroller is the mealy state machine shown below in Figure 2.3.

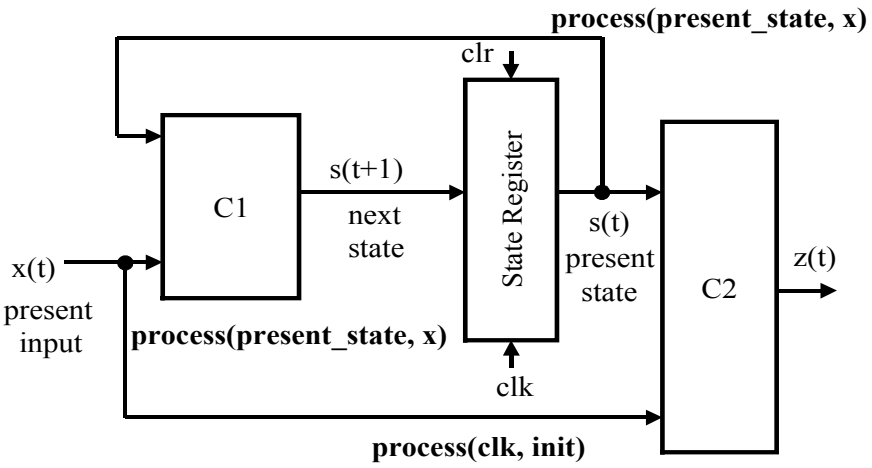


Figure 2.3: The WnX state machine.

This state machine has three states: Fetch, Execute, and Execute-Fetch. A portion of its VHDL implementation is shown in Figure 2.4. The controller begins in the

```
C1: process(current_state, M)
begin
  case current_state is
    when fetch =>
      if M(6) = '1' then
        next_state <= exec;
      else
        next_state <= exec_fetch;
      end if;
    when exec_fetch =>
      if M(6) = '1' then
        next_state <= exec;
      else
        next_state <= exec_fetch;
      end if;
    when exec =>
       next_state <= fetch;
  end case;
end process C1;
```

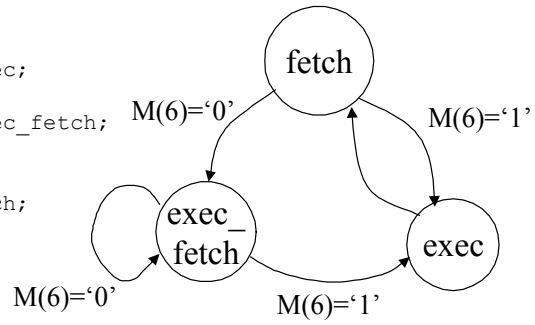


Figure 2.4: A three-state VHDL implementation.

fetch state to 'fetch' the next instruction from the program ROM. If the instruction requires only a single clock cycle to execute, the current instruction is executed and the next instruction is read from the program ROM in the Execute-Fetch state. The instructions continue to be executed and fetched at the same time until it is presented with an instruction that requires more than one clock cycle.

These multi-cycle instructions have been assigned opcodes with a '1' in the 6th bit position, the second most significant bit of the opcode. For instructions requiring two clock cycles, the controller executes the current instruction without 'fetching' the next word from the program ROM. On the second clock cycle, the microcontroller returns to the fetch state to 'fetch' the next instruction.

The top of stack register in the Register Array of the extended microcontroller is an SPI (Serial Peripheral Interface). The SPI top-of-stack register is shown below in Figure 2.5.

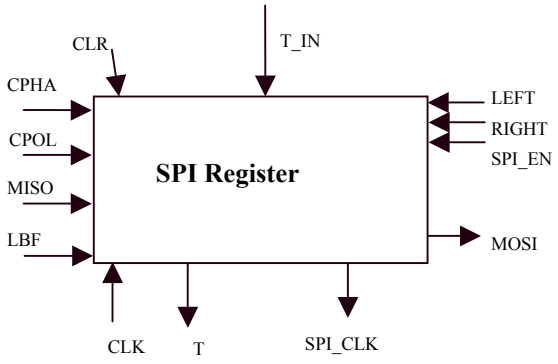


Figure 2.5: The WnX top-of-stack SPI register.

An instruction, SPI, is included in the extended instruction set. Obviously, this instruction requires more than one or two clock cycles. The microcontroller remains in the execute state for the correct number of clock cycles to complete the SPI transmission.

The SPI runs in all four standard SPI modes: active high and low with CPHA = 1 and active high and low with CPHA = 0. Using the SPI interface, the WnX will now accept data directly into and transfer data directly from the top of the stack. This design was synthesized on a Xilinx 4010 FPGA and simulated using the Aldec Active VHDL simulator. A simulation is shown below in Figure 2.6.

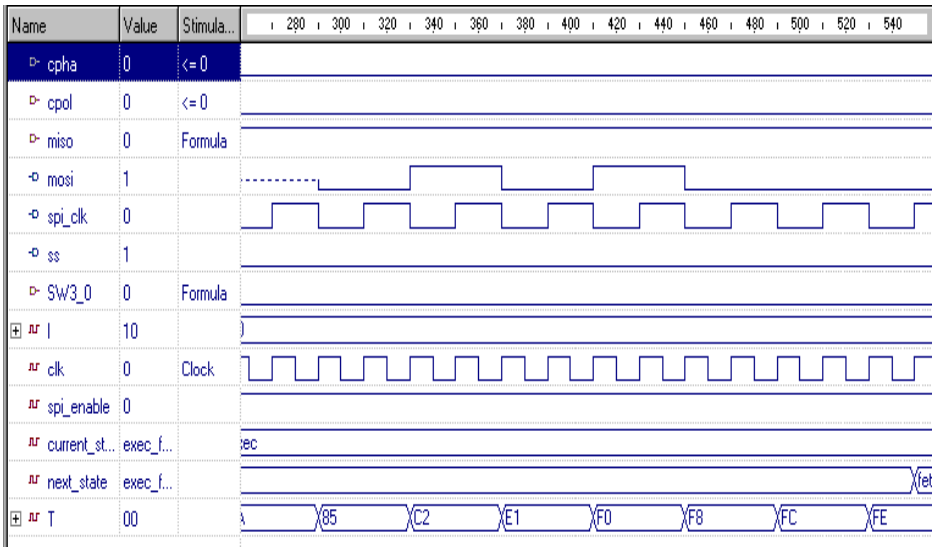


Figure 2.6: The WnX SPI simulation with CPHA=0 CPOL=0.

References

1. Mano, M. M. and C. R. Kime., Logic and Computer Design Fundamentals*, 2nd Ed.*, Prentice Hall, Upper Saddle River, NJ, 2000.

2. Haskell, R. E., Design of Embedded Systems Using 68HC12/11 Microcontrollers, Prentice Hall, Upper Saddle River, NJ, 2000.

3. Golden, J., Moore, C. H., and Brodie, L., "Fast Processor Chip Takes Its Instructions Directly from Forth," *Electronic Design*, March 21, 1985, pp. 127-138.

4. Hand, T., "The Harris RTX 2000 Microcontroller," *Journal of Forth Application and Research*, Vol. 6, No. 1, pp. 5-13, 1990.

5. Koopman, Jr., P., "32-Bit RTX Chip Prototype," *Journal of Forth Application and Research*, Vol. 5, No. 2, pp. 331-335, 1988.

6. Hayes, J. R., Fraeman, M.E., Williams, R. L., and Zaremba, T., "A 32-Bit Forth Microprocessor," *Journal of Forth Application and Research*, Vol. 5, No. 1, pp. 39-48, 1987.

7. Hayes, J. and Lee, S., "The Architecture of the SC32 Forth Engine," *Journal of Forth Application and Research*, Vol. 5, No. 4, pp. 49-71, 1989.

8. Moore, C., "ShBoom on ShBoom: A Microcosm of Software and Hardware Tools," Proc. 1990 Rochester Forth Conference, pp. 21-27, June 12-15, 1990.

9. Ting, C. H., "P Series of Microprocessors," in *More on Forth Engines*, Vol. 22, pp. 1-17, Sept. 1997.

10. Ting, C. H., "P16 Microprocessor Design in VHDL," in *More on Forth Engines*, Vol. 22, pp. 44-51, Sept. 1997.

11. Ashenden, P. J., The Designer's Guide to VHDL, Morgan Kaufmann, San Francisco, 1996.

The shortest answer is doing.

Lord Herbert 1583-1648

CHAPTER 3: A MICROPROCESSOR/FPGA DESIGN COMPARISON: TO FPGA OR NOT TO FPGA

3.1 A Basic Implementation Example – An Ultrasonic Tape Measure

Interfacing a pre-packaged or pre-designed component to build an embedded solution typically requires the use of a microprocessor. If the interface to the microprocessor were eliminated and the design could be implemented using software that optimized the system to hardware and embedded software, the hardware and software would be optimally split after the design has been completed. To this extent, committing to a specific limited microprocessor or an expensive functional-rich microprocessor is not necessary. Furthermore, design changes will be accommodated in software and necessary supporting hardware to manage timing and other constraints along with necessary embedded software would be generated for the system, thus eliminating the

possibility of having to apply backward compatible patches to existing software due to a change in hardware.

As a basic example of a small embedded system implemented using a microprocessor verses implementing the same system using the automatically reconfigurable WnX microcontroller by designing the system in software is an ultrasonic tape measure. An ultrasonic ranging system is available from Polaroid Corporation. It consists of an acoustical transducer and a ranging circuit board. The transducer transmits an ultrasonic pulse when the INIT input is asserted and remains high. This ultrasonic pulse reflects off of the target object and returns to the transducer, which asserts the ECHO output high until the INIT signal is returned to ground. Figure 3.1 shows a block diagram of the Ultrasonic Ranging System.[1]

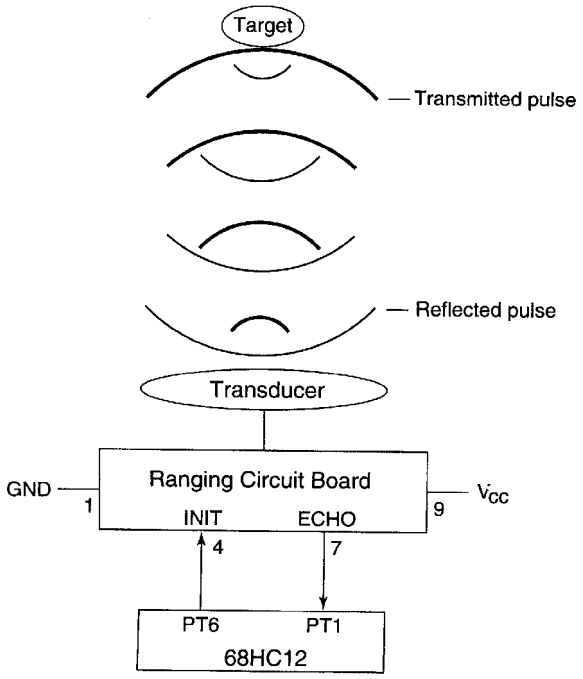


Figure 3.1: Block diagram of the Ultrasonic Ranging System. [*Source:* Haskell]

The transducer can measure up to nearly 36 feet. Since the speed of sound is approximately 1.125 ft/ms or 13.5 in/ms (at 20° C), the sound will travel a distance of 36 feet in about 32 ms. Since the sound waves reflect off of the target object, they must travel a round trip that will take about 64 ms.[1]

In the *Design of Embedded Systems using 68HC12/11 Microcontrollers*, Haskell interfaces the ranging board to the 68HC12 by connecting the INIT pin to an output compare pin on the 68HC12 and using an input capture pin to design an ultrasonic tape measure.

In this chapter, the same Ultrasonic Tape Measure will be designed using the reconfigurable WnX microcontroller and supporting internal hardware/software. Both implementations will be compared with respect to complexity and extensibility. Other microcontrollers may be used to implement the ultrasonic tape measure less expensive and slightly easier than the Motorola 68HC12, however, traditional co-design processes and high-risk decisions are still apparent.

3.2 The Ultrasonic Tape Measure using the Motorola 68HC12

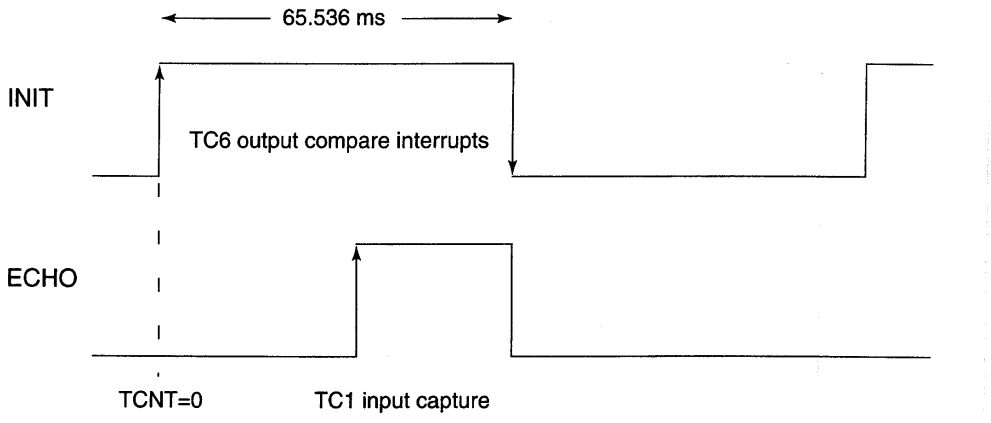


Figure 3.2: Waveforms of the Ultrasonic Ranging System. [*Source:* Haskell]

To expedite the implementation, a particular member of the team, Chuck, uses WHYP words described in Chapter 2 to write the software for the 68HC12. Since the maximum time for the sound wave to make a round trip is about 64 ms, if the timer prescalar on the 68HC12 is set to 8, the timer will overflow every 65.536 ms. Chuck realizes that he must set up an output compare to enable the Ultrasonic Transducer and an input capture that is asserted high when the sound wave returns. The INIT signal shown in Figure 3.2 is produced. Chuck continues, after reading up on the necessary assembly language for the Motorola 68HC12, to design the program shown below in listing 3.1.

Listing 3.1: Ultrasonic Tape Measure in WHYP [*Source:* Haskell]

```
\     Ultrasonic tape measure File: SONAR.WHP
LOAD SPILED.WHP   \ for .leds and SPI words (Fig. 7.11)
HEX

0080    CONSTANT  TIOS          \ Timer Input Cap.\Output Comp. Select
0084    CONSTANT  TCNT          \ Timer Counter Register
0086    CONSTANT  TSCR          \ Timer System Control Register
0088    CONSTANT  TCTL1         \ Timer Control Register 1
008B    CONSTANT  TCTL4         \ Timer Control Register 4
008C    CONSTANT  TMSK1         \ Timer Interrupt Mask Register 1
008D    CONSTANT  TMSK2         \ Timer Interrupt Mask Register 2
008E    CONSTANT  TFLG1         \ Timer Interrupt Flag Register 1
0092    CONSTANT  TC1           \ Timer Input Capture Register 1
009C    CONSTANT  TC6           \ Timer Output Compare Register 6
0B22    CONSTANT  TC6.IVEC      \ Timer Channel 6 interrupt vector

VARIABLE DISTANCE
VARIABLE ECHO

: init.sonar      ( -- )
            F0 TIOS C!        \ PT6 output, PT1 input
            80 TSCR C!        \ enable timer
            03 TMSK2 C!       \ 1 MHz timer clock
            5 TCTL1 LO        \ toggle PT6 (TC6)
            4 TCTL1 HI
            0 TC6 !           \ sync to TCNT
            6 TMSK1 HI        \ enable TC6 int
            3 TCTL4 LO        \ rising edge of PT1
            2 TCTL4 HI ;
```

Listing 3.1 ***continued***

```
INT: TC6.INTSER  ( -- )       \ int on both edges
             6 PORTT ?HI      \ if rising edge
             IF
                02 TFLG1 C!   \  clear C1F flag
             ELSE             \ if falling edge
                1 TFLG1 ?HI   \  if echo
                IF
                   TC1 @      \   get distance
                   DISTANCE ! \   & save it
                   TRUE ECHO ! \   echo=true
                ELSE          \  else
                   FALSE ECHO !      \   echo=false
                THEN
             THEN
                   40 TFLG1 C!       \ clear OC2F flag
RTI;

: SET.TC6.INTVEC        ( -- )
                [ ' TC6.INTSER ] LITERAL
                TC6.IVEC ! ;

: ?distance ( -- n tf | ff )
             ECHO @
             IF
                DISTANCE @
                TRUE
             ELSE
                FALSE
             THEN ;


: sonar.tape      ( -- )
             init.sonar
             SPI.INIT
             BEGIN
                ?distance
                IF             \ if echo
                   135 20000 */      \ convert to inches
                   .4leds            \ display distance
                ELSE
                   SS.LO             \ else
                   EE SEND.SPI \ display dashes
                   EE SEND.SPI
                   SS.HI
                THEN
             AGAIN ;
```

As detailed in the Design of Embedded Systems Using 68HC12/11 Microcontroller[1]:

Output compare 6 is set up to toggle *PT6* and produce an interrupt on each timer match. The value of *TC6* is set to zero so that the match will always occur when *TCNT* rolls over from $FFFF to $0000. Inasmuch as *PT6* is connected to the *INIT* signal on the ranging circuit board, then a new ultrasonic pulse will be transmitted on each rising edge of *PT6* when the value of *TCNT* will be zero. This means that if there is an echo and a rising edge of *PT1* stores the value of *TCNT* in *TC1*, this value will be the total elapsed time since the pulse was transmitted.

The interrupt service routine will be executed on both the rising and falling edge of *PT6*. The routine first checks to see if it was a rising edge. If it was, then this means that an ultrasonic pulse is being sent. The *C1F* flag of input capture 1 is cleared so that we can determine if an echo occurs before the next falling edge of *PT6*. When a falling edge of *PT6* occurs, the ELSE part of the interrupt service routing, *TC6.INTSER* is executed. This will check the *C1F* flag in *TFLG1* to see if an echo occurred. If it did, then the distance value is read from *TC1* and stored in the variable *DISTANCE* and a TRUE flag is stored in the variable *ECHO*. If an echo did not occur, then a FALSE flag is stored in the variable *ECHO*.

The word *?distance (-- n tf | ff)* checks the values in the variables *ECHO* and *DISTANCE* and will return a false flag if an echo has not occurred within the 65.536-ms time-out time. Otherwise, it will return a true flag over the echo time, *n,* corresponding to the total elapsed time from *INIT* going high to *ECHO* going high. Since each tick corresponds to 1.0 μs, then *n*/1000 would be the total elapsed time in milliseconds.

The word *sonar.tape (--)* will continually read the distance from the transducer to the target using the word *?distance (-- n tf | ff)* and display this value on the three-digit seven-segment display.

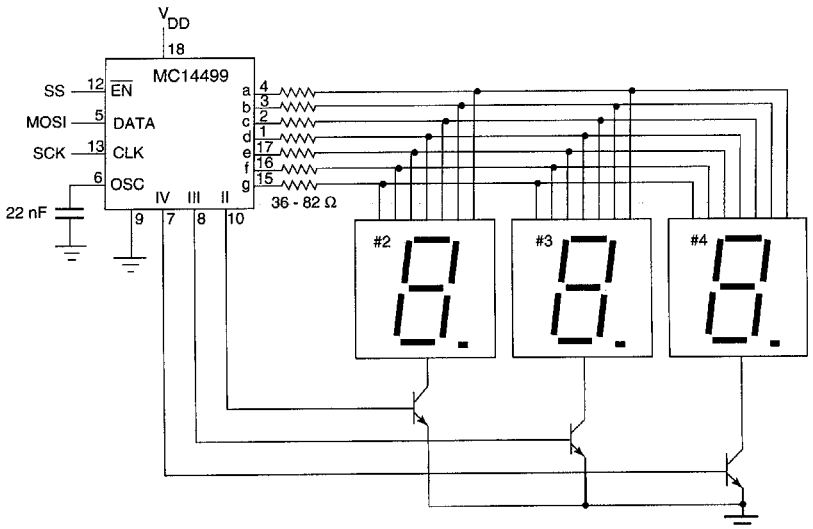


Figure 3.3: Connecting the MC14499 to three common-cathode seven-segment displays. [*Source:* Haskell]

In addition to writing the WHYP code shown in Listing 3.1, Chuck also needed to write the WHYP code to interface with three external common-cathode seven-segment displays. Chuck selected a standard IC, the MC14499 to interface to these displays. The displays and interface are shown in Figure 3.3.

Listing 3.2, below, shows the WHYP code written to interface with the MC14499 and the Motorola 68HC12.

Listing 3.2: Displaying up to Four Seven-Segment LEDs Using the MC14499 [*Source:* Haskell]

```
\     4 LEDs Using the MC14499 Decoder/Driver with Serial Interface
\     File: SPILED.WHP

LOAD SPI.WHP
LOAD STRING.WHP
DECIMAL

: pack2          ( addr -- c )
                 DUP C@
                 4 LSHIFT                \ addr c1
                 SWAP 1+ C@              \ c1 c2
                 15 AND OR ;

: .4leds          ( n -- )
                 SS.LO
                 10 BASE !
                 (U.) 4 SWAP -           \ addr #blanks
                 FOR               \ addr
                   1- 15 OVER C!         \ store F for blank
                 NEXT
                 DUP pack2 SEND.SPI      \ 1st digit
                 2+ pack2 SEND.SPI       \ 2nd and 3rd digit
                 SS.HI ;
```

In addition to the LED functions that are designed to use the Motorola 68HC12's SPI port, Chuck pulled the two files loaded before the WHYP code written in Listing 3.2 from his previous WHYP library. Remember that Chuck used his collection of existing WHYP source to expedite the project. Listing 3.3 shows the code used from the library to use the 68HC12's SPI port.

Listing 3.3: Basic WHYP SPI Words [*Source:* Haskell]

```
\     Serial Peripheral Interface
HEX
00D0  CONSTANT   SP0CR1       \ SPI Control Register
00D2  CONSTANT   SP0BR        \ SPI Baud Rate Register
00D3  CONSTANT   SP0SR        \ SPI Status Register
00D5  CONSTANT   SP0DR        \ SPI Data Register

: SPI.INIT  ( -- )           \ Initialize SPI port
            40 PORTS C!      \ SS lo, sclk lo, MOSI hi
            E2 DDRS C!       \ SS lo when DDRS7 set
            04 SP0BR C!      \ 250 KHz (/32)
            54 SPCR C! ;     \ CPHA = 1, CPO; = 0

: ?SPI.DONE       ( -- f )         \ Is SPI data sent?
            7 SP0SR ?HI ;

: SEND.SPI        ( c -- )
            SP0DR C!         \ send char
            BEGIN
               ?SPI.DONE     \ wait till sent
            UNTIL ;

: SS.HI     ( -- )           \ set SS high
            7 PORTS HI ;

: SS.LO     ( -- )           \ set SS low
            7 PORTS LO ;
```

Additionally, Chuck used a single word from his extensive "string" library. This WHYP word is shown below in Listing 3.4.

Listing 3.4: A Single WHYP Word from the String Library [*Source:* Haskell]

```
…
: (U.)          ( u -- addr len )
                0 <# #S #> ;
…
```

Without Chuck's library, the words used would have had to be written in 68HC12 assembly language. The original assembly language source for select WHYP words is shown in Listing 3.5.

Listing 3.5: 68HC12 Assembly Source for Select WHYP Words [*Source:* Haskell]

```
;       @       ( a -- w )
AT
      LDY     0,X             ;Y = a
      MOVW    0,Y,0,X         ;w = @Y

;       C!      ( c b -- )
CSTOR
      LDY     2,X+            ;Y=b
      LDD     2,X+            ;D = c
      STAB    0,Y             ;store c at b
      RTS

;       TRUE = -1 = $FFFF
TRUE
      LDD     #-1
      STD     2,-X
      RTS

; SWAP ( w1 w2 -- w2 w1 )
; Exchange top two stack items.
SWAP
      LDD 0,X
      MOVW 2,X,0,X
      STD 2,X
      RTS

;       + ( X Y -- X+Y )
PLUS
      LDD     2,X+
      ADDD    0,X
      STD     0,X
      RTS

;       (CREATE)        ( +++ )      Run time for CREATE
PCREATE
      LDAA    #15              ;code 15
      JSR     OUTPUT
      PC1     JSR     INWDY           ;read word
      CPY     #6              ;if 6, exit
      BEQ     PC2
      JSR     0,Y             ;else, execute sub
      LDAA    #6
      JSR     OUTPUT          ;send ACK
      BRA     PC1
      PC2     RTS
```

It is obvious that all of the WHYP words that Chuck had in his library for the Motorola 68HC12 provided a tremendous asset to the project.

3.3 The Ultrasonic Tape Measure using the Reconfigurable WnX Microcontroller

A block diagram of an implementation using an FPGA and the reconfigurable WnX described in Chapter 2 is shown in Figure 3.4. As outlined in section 3.2, the transducer can measure distances up to about 36 feet and the sound wave emitted by the transducer requires a maximum of 64 ms to reflect off of the target object and return to the transducer. If a nominal clock frequency of 3.90625 kHz (125 kHz with a clock divider of a factor of 2^5 or 32) for all synchronous components in the design is used, an

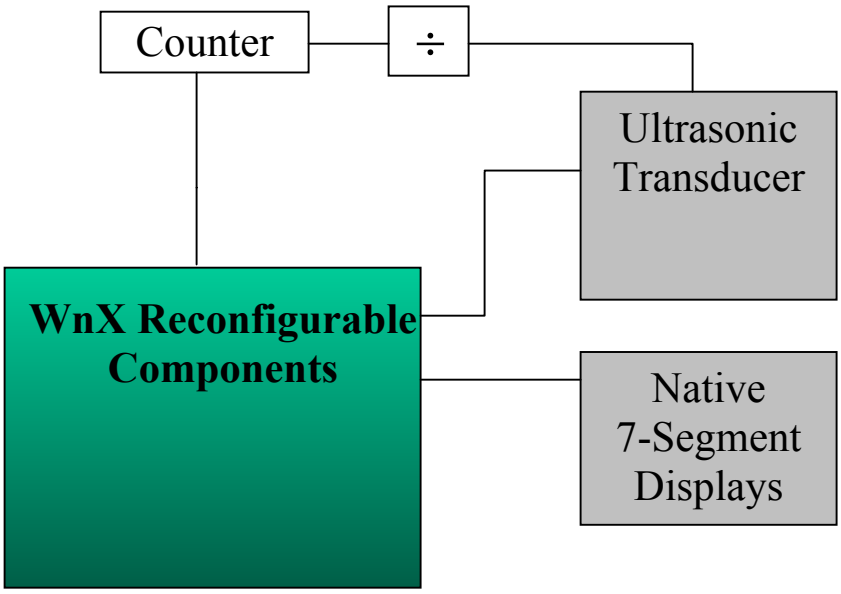


Figure 3.4: A block diagram for the ultrasonic tape measure using an FPGA without an external microcontroller.

8-bit counter will overflow every 65.536 milliseconds. In this case, an 8-bit WnX could be generated, therefore, this example will use a W8X. If precision or speed were crucial

constraints, a 16-bit W16X could be generated and a 16-bit counter value with a much faster clock could be used to directly latch the 16-bit counter value into a single top-of-stack register instead of manipulating 16-bits in two registers. Upon the assertion of the ECHO pin, the value of the counter will latch into the top-of-stack register and cause an interrupt in the reconfigurable W16X. This interrupt runs a subroutine to transfer the counter value from the register to the top-of-stack and perform the appropriate instructions required for the calculations to convert the counter value to the distance in inches.

The counter overflows at 256, therefore an 8-bit counter is used. Since each clock period is 256 μs and *n* is the value of the counter from the time the INIT signal went high to the time the ECHO signal went high, the total elapsed time in milliseconds can be obtained by the expression 256*n*/1000. The total distance in inches corresponding to the total time elapsed can be calculated by multiplying *n* by 256 (256 μs per clock period) and 135 (13.5 inches/ms) and dividing by 10,000 (to convert to milliseconds and compensate for the factor of 10 in the numerator). Since the sound wave reflected off of the target object and returned to the transducer, the distance from the transducer to the object is 1/2 of the total distance. Therefore, Equation 3.1 shows the distance between the transducer and the target object.

$$\frac{n * 256 * 325}{20{,}000} \qquad \text{Equation 3.1}$$

An optimal microcontroller is generated; in this case an 8-bit microcontroller, along with necessary components. Among the necessary components is a 16-bit x 8-bit

divider that will yield an 8-bit result and an 8-bit remainder. A simplified expression shown in Equation 3.2 can be obtained by factoring the numerator and denominator of Equation 3.1.

$$\frac{n*104}{25} \qquad \text{Equation 3.2}$$

The simplified expression in Equation 3.2 requires a 15-bit (for simplicity let's use 16-bit) numerator and a 5-bit (for simplicity let's use 8-bit) denominator. Therefore, necessary hardware or software will be generated to support an optimal divider. If timing was a crucial constraint, combinational ALU operations could be generated. Each calculation including complete transmission of the value to the seven-segment displays has 255 clock cycles to complete. Since this is more than enough clock cycles, such a stringent timing constraint is not necessary for this application.

The interrupt routine generated for the reconfigurable W8X is given in Listing 3.6. Notice that using a reconfigurable microprocessor reduces the code required to perform a specifically designated task. In this case, the W8X performs one task; it acts as an interrupt handler for the ultrasonic tape measure. The W8X is used more as a math co-processor with no microprocessor!

In addition to the application-specific code for the interrupt service routine shown in Listing 3.6, the following support subroutines, shown in Listing 3.7, are also generated automatically. Only required support subroutines are generated and program ROM sizes are minimized after the required code has been generated. This optimizes the overall system design.

Listing 3.6: The Entire Interrupt Routine for the Ultrasonic Tape Measure

```
--APPLICATION SPECIFIC SOURCE

00    DINT              --Wait for interrupt
01    JNI, X"00",       --Jump zero to DINT and keep waiting
03    LIT, X"68",       --Load a decimal 104 as the multiplier
05    PUSHD,            --Obtain the counter value as the multiplicand
06    CALL, X"0014",    --Call the built-in multiplication routine
08    LIT, X"19",       --Load a decimal 25 as the divisor
0A    ROT,              --Rotate the top 3 elements clockwise - divisor
                        --on bottom
0B    CALL, X"0021",    --Call the built-in division routine
0D    NIP,              --Not interested in the remainder, drop N1 and
                        --pop rest of stack
0E    WSPI0,            --Call the Write SPI instruction to transfer
                        --the 8-bit data out to the Binary to BCD
                        --converter to the Seven-Segment Displays
0F    CLI               --Clear/Reset the Interrupt
10    JMP, X"00"        --Return from interrupt
```

External RAM can be easily used when larger amounts of memory are required, however, the lower the number of interfaces, the more like software development embedded system co-design becomes.

Listing 3.7: Built-in Support W8X Subroutines

```
--MULTIPLICATION SUBROUTINE (12 clock cycles) Built-in
12    LIT, X"00", --Load a 0 to the top of stack and push data stack
13    MPP, MPP, MPP, MPP, MPP, MPP, MPP, MPP,
                  --Multiply partial product eight times
1B    2NIP        --Clean up the data stack (=ROT, DROP)
1C    RET,

--DIVISION SUBROUTINE (11 clock cycles) Built-in
1D    MROT,       --Rotate the top 3 elements counter-clockwise -
                  --Divisor on top
1E    SHLD, SHLD, SHLD, SHLD, SHLD, SHLD, SHLD, SHLD,
                  --Shift left for division 8 times
26    S2NIP       --Clean up the data stack (=ROT, DROP, SWAP)
27    RET,
```

Once the 8-bit distance is on the top of the stack (after interrupt service routine address 0D), the WSPI instruction is called to transfer serially the data to the built-in

binary-to-BCD converter. The FPGA architecture lends itself very well to serial code conversion, such as binary-to-BCD conversion. Data is entered serially into a register in one format and retrieved from the same register in a different format as a parallel output.[2]

For those who are interested, binary-to-BCD conversion is performed in a modified shift register that successively doubles its BCD contents. The binary data are shifted into the converter serially, most significant bit first. Subsequent bits are shifted serially into the converter. The conversion is complete when all of the binary input has been shifted into the register, at which time the BCD result is available. The available output, as shown below in Figure 3.5, does not require serial transfer, instead, it is immediately available in parallel. To remain a valid BCD number when doubled, a BCD digit of 5 or greater must not be shifted, but must be converted into a proper BCD representation, along with a 1 being shifted into the next higher digit.[2] This provides a component-based solution for the generator to produce a binary-to-BCD converter for an *m*-bit binary input requiring *m* clock cycles to convert the number into a BCD representation. This BCD representation is mapped directly to an on-chip built-in Seven-Segment Decoder. A detailed example is given in Appendix C.

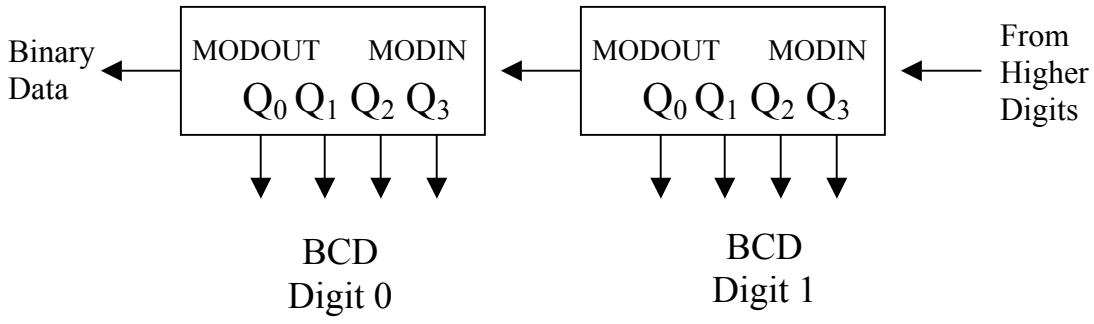


Figure 3.5: The binary-to-BCD converter. [*Source:* Alfke]

The FPGA implementation of the ultrasonic tape measure requires minimal code and little, if any, waste of chip resources. Once the design has been generated, an appropriately sized FPGA can be selected. In addition to offering a minimal solution, the FPGA implementation does not require any additional hardware except for the Polaroid Ultrasonic Transducer and the native Seven-Segment Displays. All timing measurements, computations, conversions, and decoding takes place in the Field Programmable Gate Array.

3.4 Scalability – A Field of Dreams

Clearly, a reconfigurable solution offers many advantages and simplifies the design verses using a packaged microprocessor. There are, however, more benefits than meet the eye. Consider the scalability in the case of the simple example of the ultrasonic tape measure. In section 3.2 and 3.3, this tape measure was designed using a Motorola 68HC12 and a reconfigurable WnX on an FPGA, respectively. Suppose that after designing the measuring system, a larger system was desired, for example, a simple system comprised of four (4) Ultrasonic Tape Measures that would have a wider range than a single transducer. This system is shown in Figure 3.6.

Both implementations could be extended relatively easily to implement this system. The Motorola 68HC12 has 8 input captures and 8 output compares. If Chuck were to implement a system of four transducers, he could reuse the same code that he used to design the first single-transducer system and make minor changes to the referenced input capture and output compare ports. Likewise, the reconfigurable WnX FPGA system could easily reuse its design by simply downloading four identical designs

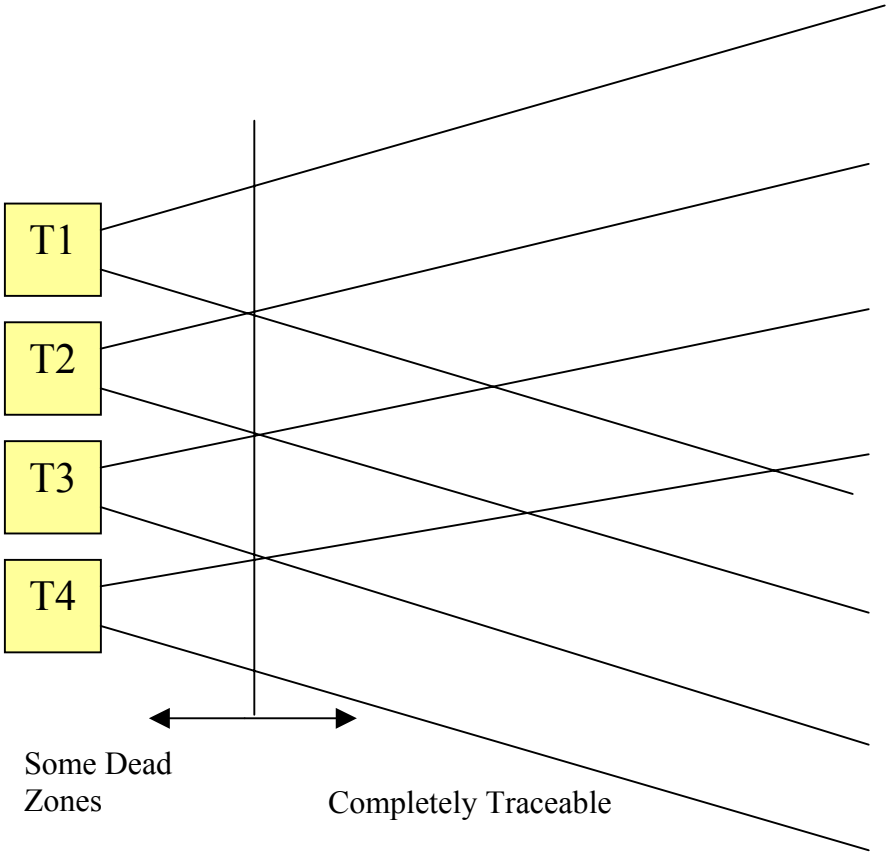


Figure 3.6: Four ultrasonic tape measures to form a span of measured area.

to the chip that operate in parallel. In this case, both implementations are relatively similar in code/time costs. The FPGA implementation still optimizes the design for space and an FPGA could be purchased to hold all four designs, in high volume, for less than $2.49. Furthermore, each successive system requires an additional MC14499 driver IC for the seven-segment display while the FPGA system generates internal decoders for the additional displays.

Suppose that only one value was desired, a non-zero distance given by whichever transducer had such data or zero if all values are zero (theoretically, if an object fell in the range of more than one transducer, both transducers would generate the same distance

value and therefore, either could be displayed). For the FPGA implementation, the BCD outputs (12 bits, 3 digits each having 4 bits) could easily be multiplexed to the internal, built-in Seven-Segment Decoder with the VHDL code listed in Listing 3.8. For a similar result using the Motorola 68HC12, a more complicated approach is required. The 68HC12 shifts its data out the SPI port directly to an MC14499 and the seven-segment display. Externally, special circuitry would have to be created to shift the data in and determine which data were non-zero and hence, which data to shift to the MC14499 and display. Internally, similarly extensive code considerations could be made to determine which value to send out a single SPI with no extra external requirements.

Listing 3.8: A Multiplexed Output in the FPGA Implementation

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity Value_Select is
    port (
        trans1: in STD_LOGIC_VECTOR (11 downto 0);
        trans2: in STD_LOGIC_VECTOR (11 downto 0);
        trans3: in STD_LOGIC_VECTOR (11 downto 0);
        trans4: in STD_LOGIC_VECTOR (11 downto 0);
        out_sig: out STD_LOGIC_VECTOR (11 downto 0)
    );
end Value_Select;

architecture Value_Select_arch of Value_Select is
begin
      if trans1 > 0 then
            out_sig <= trans1;
      elsif trans2 > 0 then
            out_sig <= trans2;
      elsif trans3 > 0 then
            out_sig <= trans3;
      elsif trans4 > 0 then
            out_sig <= trans4;
      else
            out_sig <= "000000000000";
      end if;
end Value_Select_arch;
```

Although both implementations offer similar scalability for the expanded system proposed above, the FPGA implementation offers a significant cost and resource reduction. It may, in fact, seem as though the primary difference in scalability between the FPGA implementation and the implementation using the 68HC12 is in production cost. However, consider further expanding the system by requiring 3-dimensions to be covered. In other words, four ultrasonic tape measures for each dimension, X, Y and Z. This expansion is shown in Figure 3.7.

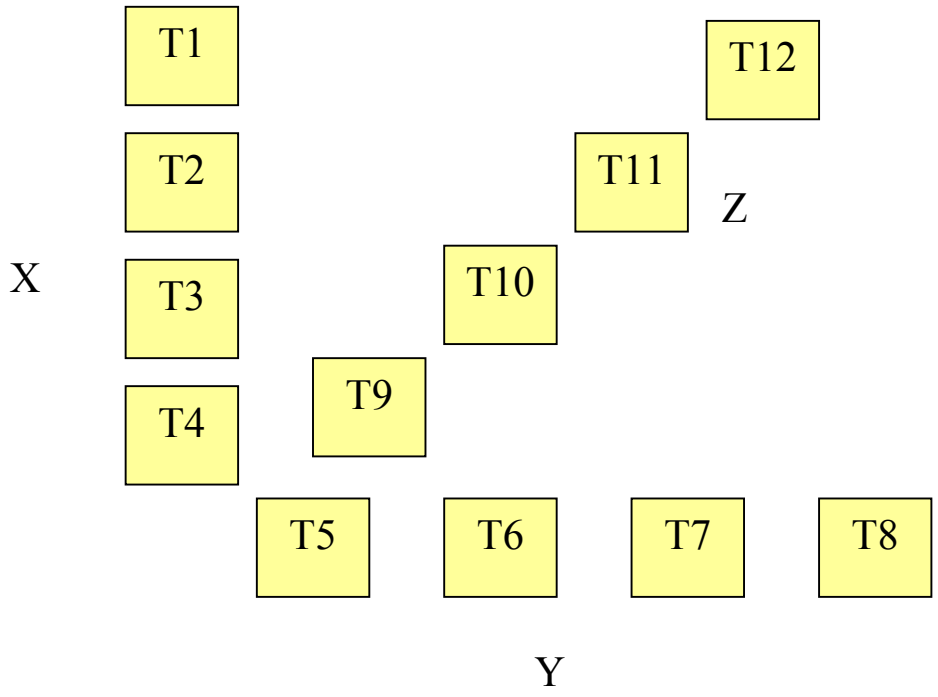


Figure 3.7: Three dimensions of ultrasonic tape measures.

We have clearly exceeded the capabilities of the 68HC12. The Motorola 68HC12 only has 8 input captures and 8 output compares. For this expansion, we would require an additional microcontroller, perhaps another 68HC12. Or at this point, it may be better

to seek a microcontroller that has 12 input captures, output compares, and SPI ports (if necessary) and port all of the existing software and libraries to the new microcontroller's assembly language.

The FPGA implementation simply could reproduce 12 of the subsystems that successfully implemented the single tape measure. If a single output were desired, the VHDL code in Listing 3.8 could easily be expanded by adding more cases to the 'IF' statement. However, more gates cost more money, in particular, millions of gates in a single chip can become very expensive. Even the FPGA implementation seems to have limits. Recall that a primary difference between hardware and software introduced at the end of Chapter 1 is Architecture. A software system can grow fundamentally larger because its raw architectural requirements are disk space and memory. Using traditional co-design tactics, the architecture requires a restructuring or reconsideration of interactivity if a system is to grow larger, before software changes can be considered.

3.5 What this Means for the Future – A Big Impact

It is true of the FPGA implementation as well, after a reasonably priced chip will no longer accommodate the design, larger chips can cost several thousand dollars. This limitation is however, in a sense, a matter of point-of-view. Consider connecting many chips together to form a field of gates bearing a larger array of gates for synthesizing a design. An example of connecting four Xilinx 40xxXL Series FPGAs is shown in Figure 3.8.

Using a high-level language to design a hardware/software system, which generates necessary components such as a reconfigurable microprocessor, may yield a

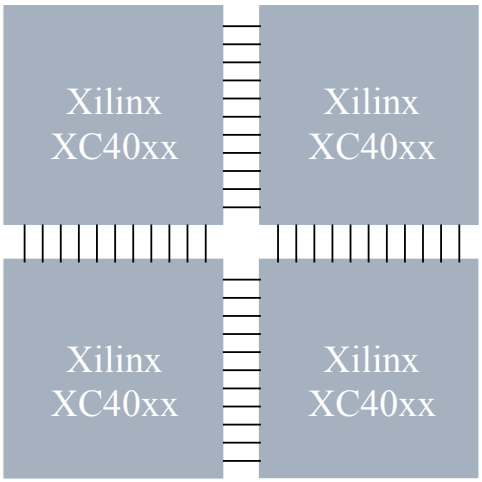


Figure 3.8: A field composed of Xilinx FPGAs connected to form a larger synthesis target.

design that is too big to fit in an affordable single chip. Connecting FPGAs together to form a larger synthesis target can be an effective solution. In addition to generating an optimal set of components, an optimal implementation plan for synthesizing over several chips complete with pin-connection requirements could be generated. According to Xilinx, "there is no limit to the number of devices in the daisy chain and XC2000, XC3000, XC4000, and XC5200 devices can be mixed freely with only one constraint: the lead device must be a member of the highest family in the chain."[4] Using a field of FPGAs as a larger synthesis target provides a base-line architecture for a high-level compiler to synthesize to where most design changes result in a need for more gates, something that can be easily and inexpensively provided by the designer. Inexpensive, that is, relative to the costs of handling the potential hazard of having to select different hardware and redesign the software to accommodate the change. A hypothetical example of a multi-chip implementation is shown in Figure 3.9.

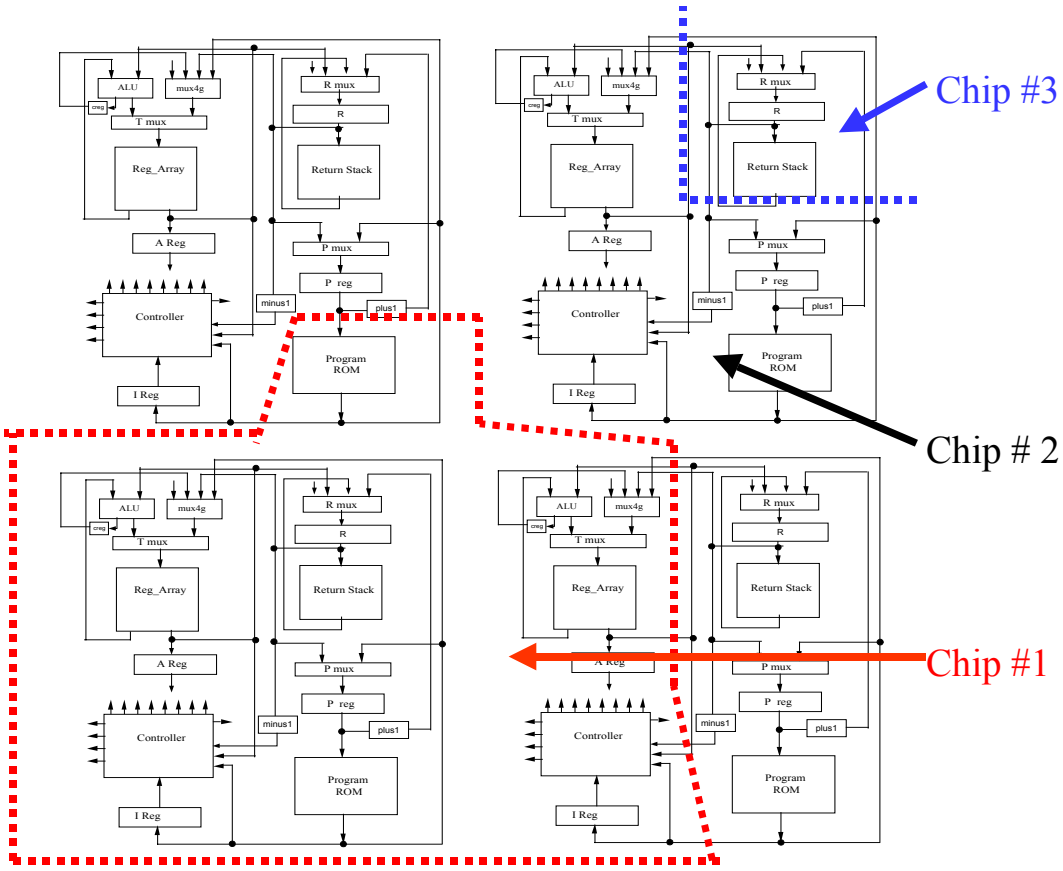


Figure 3.9: Three 16-bit reconfigurable microcontrollers implemented over multiple Xilinx 4010 series FPGAs.

In the mid-1950's, 5 Megabytes (MB) of hard drive space became available for $50,000.00. In the mid-1980's, 10 MB of disk space cost approximately $800.00. In the mid-1990's, 1,000 MB (known as a gigabyte; GB) cost $850.00. By 1999, 27 GB (27,000 MB) of space was offered to the general public at $400.00. Last month, October 2000, Maxtor released an 82 GB hard drive for $518.00.[5] Software designers for large systems in the mid-1980's targeted to a 30 MB hard drive (baseline architecture). If a software system required more than the maximum affordable hard disk space available at that time, often disk raid systems would be used. This methodology paid off in the long

run since only 3 years later, a disk drive bigger than the raid system used for the original design was available at a cost less than the raid system. Furthermore, in the early 1990's, a mere 6 years after the software in this hypothetical example was designed, hard disk space was available in abundance and at increasingly low prices.

Similarly, in the early to mid-1990's, Field Programmable Gate Arrays were available with a limited number of gates and at a high cost. This small number of gates with such a high price tag disinterested most ASIC designers whose requirements compared with the available FPGAs resembled the R.M.S. Titanic compared to a door opening. Today, FPGAs are available with 1 or more million gates with costs ranging in the low to mid thousands of dollars. On the other hand, an FPGA like a Spartan XL with 5,000 gates, for example, is available in high volume for $2.49 each.[6] Similarly, a field of FPGAs, similar to the one shown in Figure 3.8, made of 20 Spartan XLs totaling 100,000 available gates would cost approximately $60.00 to construct. FPGAs are expected to be available within the next year with more than 20,000 at the cost of a single Spartan XL. Systems designed using the 20-FPGA field could be implemented using 5 of the new 20,000 gate chips for ¼ of the cost of the original FPGA configuration. Future releases of technology, as in the case of the hard drive, would provide a means for designs implemented on an array of FPGAs to be implemented in an inexpensive single chip.

Current hardwired ASICs and Microprocessors, such as the Motorola 68HC12, are not updateable. If a new version of a microprocessor or ASIC is released to introduce fixes to known problems with the device and/or new features, an entirely new chip needs

to be manufactured. The new chip needs to physically replace the old chip; in every deployed embedded system, the chip would need to be replaced. This replacement could cost millions of dollars.

Last year, in 1999, Xilinx released a technology that started what is now called Xilinx Online. To support the development of such downloadable designs Xilinx released JBits API, a Java-based tool set based on an applications programming interface (API) that allows designers to write information directly to Xilinx FPGAs. JBits API makes it possible to create Java logic applets that can be used to send hardware updates via the Internet.[6] With this technology, FPGAs can be reprogrammed over the Internet. Updates and/or new features can be implemented online – for free.

References

1. Haskell, Richard E., Design of Embedded Systems Using 68HC12/11 Microcontrollers. Prentice Hall, Upper Saddle River, NJ, copyright 2000.

2. Alfke, Peter, and New, Bernie, "Serial Code Conversion between BCD and Binary," *Xilinx Application Note* number 029, October 27, 1997. Version 1.1.

3. Hayes, John P., Computer Architecture and Organization, McGraw-Hill Series in Computer Science, Boston, copyright 1998, third edition.

4. Alfke, Peter, "Configuring Mixed FPGA Daisy Chains," *Xilinx Application Note* number 091, November 24, 1997. Version 1.0.

5. Smith, Ivan, A statistical review of hard drives is offered at Nova Scotia's Electronic Attic, et al. URL: http://www.alts.net/ns1625/winchest.html (Accessed November 2000), elaborated in Appendix A.

6. Xilinx Production Information, "The Spartan XL: Rome wasn't Built in a Day," Xilinx Corporation, available online at http://www.xilinx.com/xlnx/xil_prodcat_landingpage.jsp?title=Spartan, Last Accessed November 2000.

7. Clarke, Peter, "Xilinx Launches Support for Internet Reconfigurable Logic," *EE Times*, May 21, 1999.

8. "Xilinx Online Field Upgradable Capability Powers New Remote Server Management System From Apex: Spartan FPGAs reconfigured for digital signal processing functions," Press Release from Xilinx, Inc., August 2, 1999.

Any sufficiently advanced technology
is indistinguishable from magic

Arthur C. Clarke

CHAPTER 4: RECONFIGURABLE HIGH-SPEED APPLICATIONS – BEYOND THE MICROCONTROLLER

4.1 Artificial Intelligence on a Chip

Thousands of successful approaches for processing data to provide predictive outputs, classify data and solve NP-Hard problems have been employed over the last half-century. Most of these techniques have been implemented using various different programming languages that have evolved over time to develop software that runs on a standard CISC microprocessor. The most critical benchmarks for many of these algorithms that provide solutions to complex data problems are accuracy, precision and speed. In some cases, there are several speed benchmarks, for example, the amount of time it takes to train a neural network and how fast a predication can be generated using test data. In the case of the neural network training times typically far exceed testing times by several multiples.

Another important contribution to data processing used to locate global maximums and minimums or to maximize the accuracy of a decision system by optimizing the systems parameters, is the genetic algorithm. Genetic algorithms are parallel-search procedures that are applicable to both continuous and discrete optimization problems.[4] Genetic algorithms are stochastic and less likely to get trapped in local minima and facilitate both structure and parameter identification in complex inference systems. The genetic algorithm operates by generating somewhat random solutions called *chromosomes* to form a set of possible solutions called a *population*. In each *generation*, or iteration of the algorithm, the genetic algorithm constructs a new population using genetic operators such as mutation and crossover. Each generation yields a solution that is the same or better solution than the previous generation.[4] In terms of speed, the faster a genetic algorithm can crossover and mutate chromosomes to produce the next generation, the better the result will be after time, *t*.

Many different types of genetic algorithms have been developed with different schemas for crossover and mutation and varying fitness functions. Many use innovative twists that, for data of a particular domain, enhance the outcome. All of which depend on the speed of execution of the algorithm.

In 1995, Scott, Samal, and Seth implemented a hardware based genetic algorithm. According to Scott, *et al*, "because a general-purpose genetic algorithm engine requires certain parts of its design to be easily changed (e.g. the function to be optimized), a hardware-based genetic algorithm was not feasible until field-programmable gate arrays were developed." This work builds upon other research in reconfigurable hardware

systems which improved system performance by mapping some or all software components to hardware using reprogrammable hardware.[8] Figure 4.1 shows the HGA (hardware genetic algorithm) system.

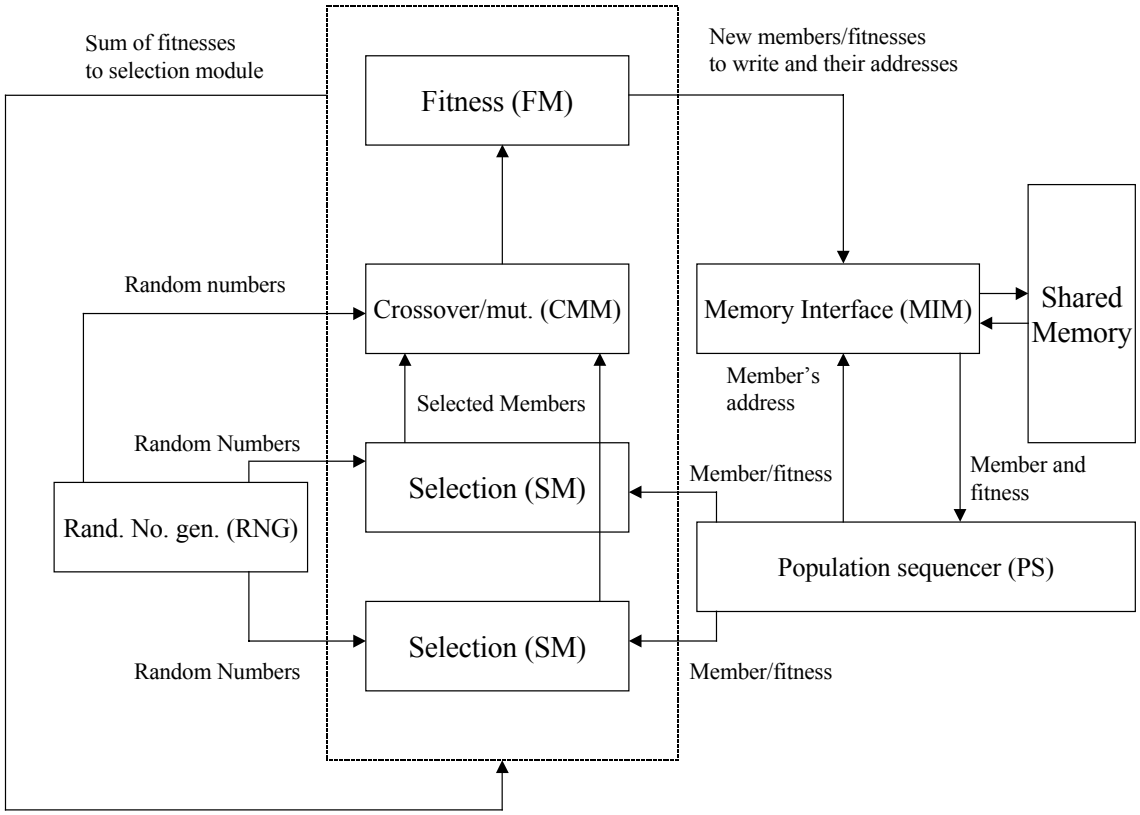


Figure 4.1: Box-level schematic of the overall HGA system. Some lines have been omitted for clarity. [*Source:* Scott, Samal and Seth]

The HGA system significantly improved the performance of the genetic algorithm. On average, the HGA prototype used 6.802% as many clock cycles as the software genetic algorithm. Table 4.1 details this performance boost. The first eight HGA tests were run on the prototype synthesized to three interconnected Xilinx XC4005 FPGAs and the last six tests were run on a VHDL simulator. All input/output times were removed from the comparisons.[8]

Table 4.1: Performance of the Software GA and the Hardware GA. [*Source:* Scott, Samal and Seth]

Fitness Function *f(x)*	Number of Generations	SGA Clock Cycles	HGA Clock Cycles
x	10	97064	5636
x	20	168034	10622
x + 5	10	99825	5585
x + 5	20	170279	10945
2x (add)	10	101019	5390
2x (add)	20	170241	10659
2x (mult)	10	101555	5390
2x (mult)	20	170668	10659
x^2	10	334210	22892
x^2	20	574046	45019
$2x^3 - 45x^2 + 300x$	10	342806	22586
$2x^3 - 45x^2 + 300x$	20	589863	44503
$x^3 - 15x^2 + 500$	10	333701	21362
$x^3 - 15x^2 + 500$	20	579176	44317

Other artificial intelligence and data processing techniques would benefit significantly by a high-speed hardware implementation as well.

4.2 A High-Speed Digital Image Processing Example

An example of a commonplace application in digital image processing is edge detection. Edge-detection operators examine each pixel neighborhood and use the slope and often the direction of the gray-level transition as metrics. There are several methods available for this examination, most of which are based upon convolution with a set of directional derivative masks.[1] The Roberts Edge Operator, Sobel Edge Operator, Prewitt Edge Operator, and Kirsch Edge Operator are all common differential or convolution operators for finding edges.

In addition to the edge-detection operators, image edges may also be detected using an artificial intelligence pattern recognition approach. One such approach was explored using a neuro-fuzzy classification tree.[3] In a binary tree classifier a decision is made at each non-terminal node of the tree based upon the value of one of many possible attributes or features. If the feature value is less than some threshold then the left branch of the tree is taken, otherwise the right branch is taken. The leaves, or terminal nodes, of the tree represent the various classes to be recognized. Since the classes that we are interested in for edge-detection are distinct (an edge pixel or a non-edge pixel), we will refer to the tree as a classification tree.

Fuzzy classification trees used in this paper have the following basic characteristics. A K-S distance associated with a fuzzy cumulative distribution function is used to select the optimum feature and threshold at each node in the tree. Each training sample can belong to more than one class with different degrees of membership. The test at each non-terminal node in the tree is considered to be a fuzzy set allowing a test sample to follow multiple paths through the tree, terminal nodes are evaluated using a defuzzification process to determine the best classification of the test data.[5] Each fuzzy membership function is characterized by two regions, a linear fuzzy region, $a-\Delta a \leq x \leq a+\Delta a$, where $f(a) = 0.5$, that ranges over the real interval (0,1) and a crisp region, $x < a-\Delta a$ or $x > a+\Delta a$, that maps to either 0 or 1. Determining which Δa will produce the best results at each node is an obstacle introduced by using fuzzy classification trees. In the

case of edge detection, Δa is a parameter that can be increased or decreased depending on the types of edges that are to be detected.

For this experiment, the black and white image shown in Figure 4.2 was used. The objective was to train a neuro-fuzzy classification tree to intelligently differentiate between an edge and a non-edge pixel.

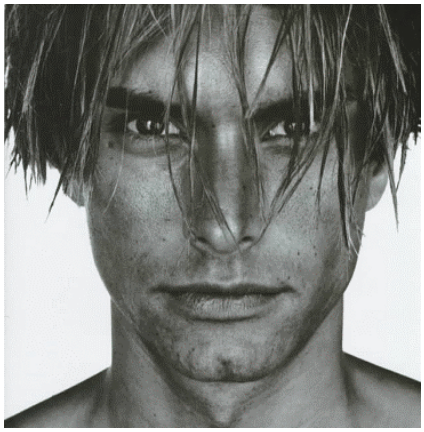


Figure 4.2: A grayscale photograph selected at random for edge-detection using a neuro-fuzzy classification tree (GIF format).

First, the edges of the face were traced in orange for training the neuro-fuzzy classification tree. These traces were used as the training data. The traced image is shown below in Figure 4.3.

Figure 4.3: Orange edge trace used as training data for the neuro-fuzzy classification tree.

Note that few pixels were used to represent an 'edge' pixel. The metric that was used for edge-detection was the difference between the numerical pixel value and the numerical pixel value of each pixel in its neighborhood. These differences were used as features for each pixel. The boundary pixels, those that are missing neighboring pixels on one or more sides, were not used. Listing 4.1 below is the Java Application used to convert the picture file from the Graphical Interchange Format (GIF) to the feature vectors and corresponding class (0 for non-edge pixel and 1 for edge pixel).

Listing 4.1: Java Application for Calculating Metrics from GIF Picture File

```
import java.awt.*;
import java.awt.image.PixelGrabber;
import java.io.*;

public class ImageTest {

      public static final int OUTLINE_PIXEL = -22;

        public static void processImage(String infile, String outfile,
                                        String aifile) {

          Image image = Toolkit.getDefaultToolkit().getImage(infile);
          PrintWriter fout = null;
          PrintWriter aiout = null;
          try {
            fout = new PrintWriter(new FileOutputStream(outfile));
            aiout = new PrintWriter(new FileOutputStream(aifile));
          }
          catch(IOException f) {
            System.out.println("Error opening output file.");
            System.exit(0);
          }
          try {
            PixelGrabber grabber = new PixelGrabber(image, 0, 0,
                        -1, -1, false);
            if (grabber.grabPixels()) {
              int width = grabber.getWidth();
              int height = grabber.getHeight();
              if (bytesAvailable(grabber)) {
                byte[] data = (byte[]) grabber.getPixels();
                // process grayscale image...
                System.out.println("Processing b&w image...\n");
                int i = 0;
```

Listing 4.1 ***continued***

```
            fout.println(data.length);
            fout.println("Width: " + width);
            fout.println("Height: " + height);
            for (i = 0; i < data.length; i++)
              {
                if (((i % width) == 0) && (i != 0))
                  fout.println("");
                fout.print(data[i] + " ");
              }

          int[][] metrix = new int[data.length][8];
          int recordClass = 0;

          //We want to fill a two-d array with metrics
          //for each pixel (not on a boundary) we want to compute
          //the difference between it and its neighbor.


          aiout.println("8\t" + ((height * width) - (2 * width) -
                       (2 * height - 4)));

          for (i = width; i < ((height - 1) * width) - 1; i++)
          //   looks like 0      ...   width - 1
          //          width  ...   2 * width - 1
          //                 ...
          //(height - 1) * width ...   height * width - 1
          {  //begin processing at the width + 1 pixel
             //we need to skip the top and bottom rows and the
             //leftmost pixel in each row and the rightmost pixel
             //in each row
             if ( ((i % width) != 0) && (((i+1) % width) != 0) )
                //this is not a leftmost or rightmost pixel
             {
                metrix[i][0] = data[i] - data[i - 1];     //left
                metrix[i][1] = data[i] - data[i + 1];     //right
                metrix[i][2] = data[i] - data[i + width - 1];
                                                  //lower left diag
                metrix[i][3] = data[i] - data[i + width]; //below
                metrix[i][4] = data[i] - data[i + width + 1];
                                                  //lower right diag
                metrix[i][5] = data[i] - data[i - width - 1];
                                                  //upper left diag
                metrix[i][6] = data[i] - data[i - width]; //above
                metrix[i][7] = data[i] - data[i - width + 1];
                                                  //upper right diag

                if (data[i] == OUTLINE_PIXEL)
                      recordClass = 1;
                else
                      recordClass = 0;
```

Listing 4.1 ***continued***

```
                  aiout.println(metrix[i][0] + "\t" + metrix[i][1] +
                                "\t" + metrix[i][2] + "\t" +
                                       metrix[i][3] +
                                "\t" + metrix[i][4] + "\t" +
                                       metrix[i][5] +
                                "\t" + metrix[i][6] + "\t" +
                                       metrix[i][7] +
                                "\t" + recordClass);
                }
              }  //all metrics have been created for all interior pixels

              }
              else {
                int[] data = (int[]) grabber.getPixels();
                // process color image
                System.out.println("Processing color image...\n");
              }
            }
          }
          catch (InterruptedException e) {
            e.printStackTrace();
          }
          fout.close();
        aiout.close();
          }

          public static final boolean bytesAvailable(PixelGrabber pg) {
            return pg.getPixels() instanceof byte[];
          }

          public static void main(String[] argv) {
            if (argv.length > 2) {
              processImage(argv[0], argv[1], argv[2]);
              System.exit(0);
            }
            else {
              System.err.println("usage:  java ImageTest <infile>
                  <outfile> <aifile>");
              System.exit(1);
            }
          }
}
```

After using the data to train the neuro-fuzzy classifier, the image data for the image shown in Figure 4.2 was used as the test picture. Figure 4.4 shows the result of the

edge detection using the neuro-fuzzy binary tree with a fuzzy percent (Δa) of 0.10. Different fuzzy percents were used yielding more edge pixels and less edge pixels.

Figure 4.4: Classified edges with $\Delta a = 0.10$.

The primary interest was to use this technique to perform real-time edge detection in hardware. Using a microprocessorless implementation, a real-time edge detector was synthesized to a Xilinx Spartan XCS10 FPGA. Figure 4.5 shows a block diagram of the FPGA system.

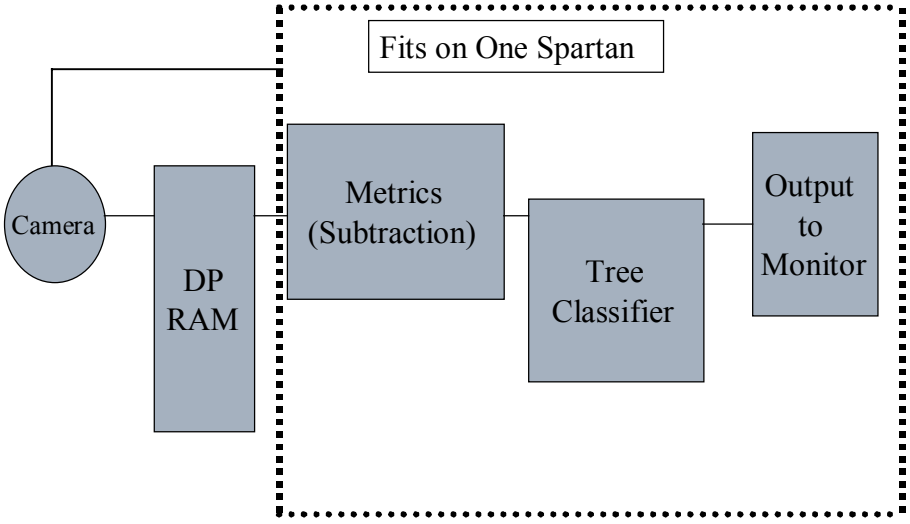


Figure 4.5: A microprocessorless implementation of the real-time edge detector using a neuro-fuzzy binary tree classifier.

For the camera, the OV5017 Camera shown in Figure 4.6 was connected to a Digilab prototyping board made by Digilent.[7]

Figure 4.6: A mounted OV5017 CMOS camera with a 7.4 mm lens.

The camera was a 384 x 288 pixel camera. The pixel data was an 8-bit integer. The first pixel that the metric could be computed for was pixel number 386 after pixel number 771 has been transmitted, therefore a 771 x 8 Dual Port RAM was required. A 5-volt asynchronous dual port RAM from Cypress was used. The camera and the Dual Port RAM were the only components that were not implemented in the FPGA. The METRICS component computed the difference between the pixel and its eight neighboring pixels.

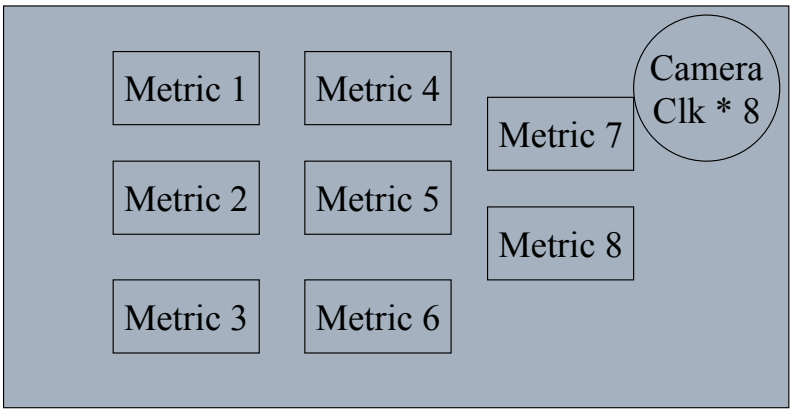


Figure 4.7: Component for calculating the metrics for edge detection

Since the camera array size is 384 x 288 pixels at 50 frames per second (default), the camera clock frequency output in pin PCLK, is 5.5296 MHz. The boundary pixels that are missing one or more of the eight nearest neighbors are not considered. Therefore, the first pixel that has all eight neighbors is pixel 386 (assuming the first pixel is pixel 1). The metrics, however, cannot be calculated until all eight of the neighbors for pixel 386 have been scanned. The eight neighbors from the upper-left diagonal to the lower-right diagonal for pixel 386 are pixels 1, 2, 3, 385, 387, 769, 770, and 771. In general, the metrics for pixel, p_i of an m x n-pixel image are given by:

$$
\begin{aligned}
M_1 &= p_i - p_{i-m-1} \\
M_2 &= p_i - p_{i-m} \\
M_3 &= p_i - p_{i-m+1} \\
M_4 &= p_i - p_{i-1} \\
M_5 &= p_i - p_{i+1} \\
M_6 &= p_i - p_{i+m-1} \\
M_7 &= p_i - p_{i+m} \\
M_8 &= p_i - p_{i+m+1}
\end{aligned}
$$

These metrics, therefore, can only be computed for pixel 386 after pixel 771 has been scanned. After pixel 771 has been scanned, the METRIC component calculates the metric for pixel 386 by accessing the eight neighboring pixels from the asynchronous dual port RAM. The clock driving the METRIC component must be eight times PCLK to continuously access the eight neighbors for every pixel scanned. For this experiment, the METRIC clock shown in Figure 4.7 is approximately 50 MHz (greater than the calculated 44.2368 MHz). Each time a pixel is scanned after and including pixel 386, the METRIC component is enabled and the metrics for a pixel are calculated at 50 MHz. Shortly after the metrics have been calculated combinationally, another pixel has been

scanned by the camera and is written to the dual port RAM. Once again, the METRIC counter that tracks the current pixel number is incremented and the metrics are read and presented to the TREE CLASSIFIER component shown in Figure 4.5. After the metrics for pixel 386 have been computed, pixel 1 is obsolete and may be replaced by the next pixel scanned from the camera. Since the edge detector operates in real-time, that is, pixels are classified as fast as the camera scans the image pixels, no more than 771 pixel data need to be maintained in the dual port RAM. The tree classifier is combinational with a registered output controlled by PCLK. Each time a pixel is read, a pixel has been classified as an edge pixel or a non-edge pixel. Non-edge pixels are displayed as one color and edge pixels are displayed as another color. The neuro-fuzzy classification tree algorithm[3] was altered to produce a VHDL file representing the decision built from the training data. The VHDL component is shown below in Listing 4.2.

Listing 4.2: VHDL Decision Tree Component

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity decision_tree is
    port (
        x1: in STD_LOGIC_VECTOR(6 downto 0);
        x2: in STD_LOGIC_VECTOR(6 downto 0);
        x3: in STD_LOGIC_VECTOR(6 downto 0);
        x4: in STD_LOGIC_VECTOR(6 downto 0);
        x5: in STD_LOGIC_VECTOR(6 downto 0);
        x6: in STD_LOGIC_VECTOR(6 downto 0);
        x7: in STD_LOGIC_VECTOR(6 downto 0);
        x8: in STD_LOGIC_VECTOR(6 downto 0);
        z: out STD_LOGIC
    );
end decision_tree;

architecture behav of decision_tree is
begin
  dt_1: process(x1, x2, x3, x4, x5, x6, x7, x8)
```

Listing 4.2 ***continued***

```
      begin
         if ((x4 < 0 and x5 < -6 and x8 < -45)
            or (x4 < 0 and x5 >= -6 and x8 < -45)
            or (x4 >= 0 and x8 < -45)
            or (x2 < -40 and x4 >= -1 and (x8 >= -45 and x8 < -4))
            or (x1 < -54 and x2 >= -14 and (x8 >= -45 and x8 < -4))
            or (x1 >= -54 and x2 >= -14 and x5 < 0 and x7 < 0
                and (x8 >= -45 and x8 < -4))
            or (x1 >= -54 and x2 >= -14 and x5 < 0 and x7 >= 0
                and (x8 >= -45 and x8 < -4))
            or (x1 < -77 and x8 >= -4)
            or ((x1 >= -77 and x1 < -47) and x8 >= -4)
            or ((x1 >= -47 and x1 < 8) and x3 < 8 and x6 < 3
                and x7 >= 18 and x8 >= -4)
            or ((x1 >= -47 and x1 < 8) and x3 >= 8 and x8 >= -4)
            or (x1 >= 8 and x5 < -10 and x8 >= -4)
            or (x1 >= 8 and (x5 >= -10 and x5 < 1) and x8 >= -4)
            or (x1 >= 8 and x4 < 3 and x5 >= 1 and x7 < 1
                and x8 >= -4)) then
            z <= '1';
         else
            z <= '0';
         end if;
      end process dt_1;
end behav;
```

Each pixel is classified as an edge or non-edge pixel determined by the value of *z*, the output bit of the tree classifier shown in Listing 4.2. If *z* is high, the pixel is considered to be an edge, if *z* is low, the pixel is a non-edge.

The last component in the system is the OUTPUT component. This component is responsible for maintaining the proper timing for outputting the pixel data (color1 or color2) to the monitor. For this experiment, the monitor output required three basic signals, a vertical-sync signal, a horizontal-sync signal and a single color signal. For this experiment, we used yellow and black as our edge and non-edge pixels, respectively. The horizontal-sync signal is used to signify the beginning of a row of pixel data when asserted high. This signal must be brought low again within 25.17μs and must remain

low a minimum of 0.94μs after the last pixel and stay low for 3.77μs. A new line of pixels can begin a minimum of 1.89μs after the horizontal-sync pulse ends. A single line occupies 25.17μs of a 31.77μs window. The remaining 6.6μs of each line is the horizontal blanking interval. Similarly, negative pulses on the vertical-sync mark the

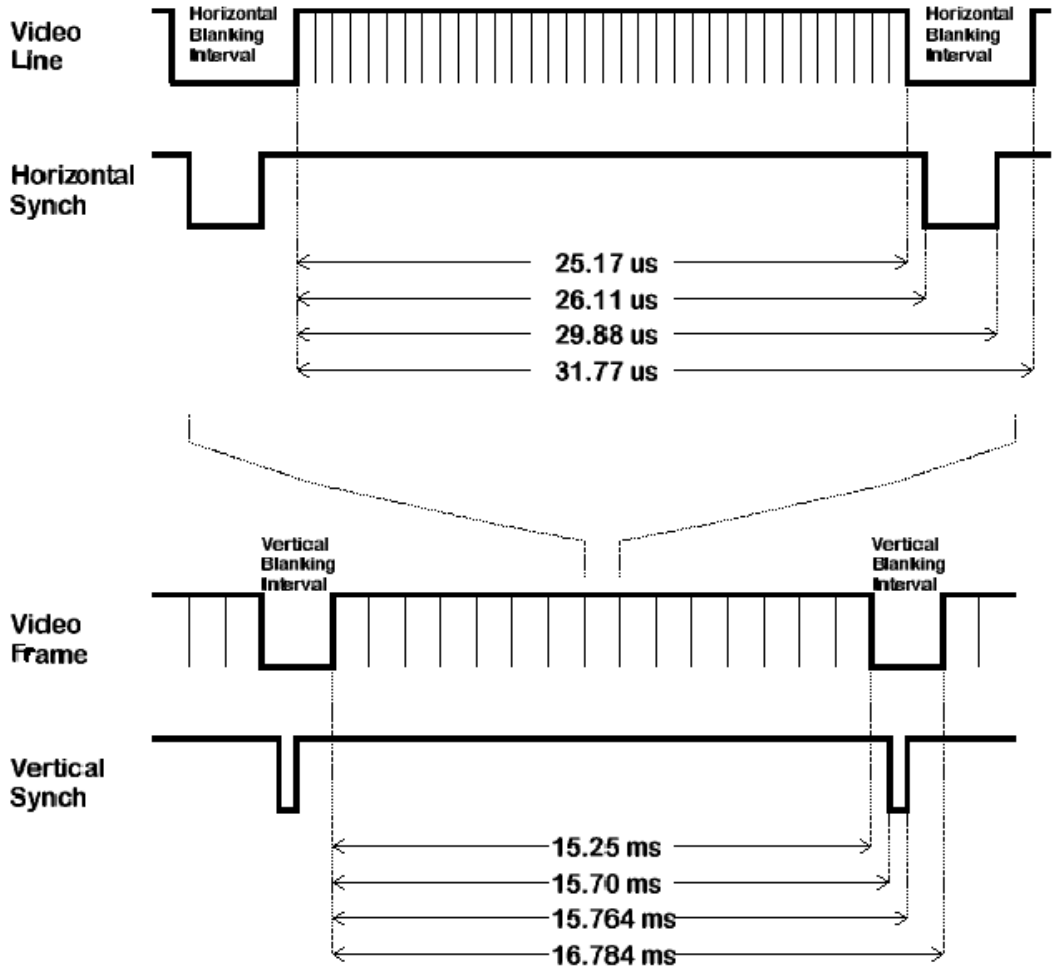


Figure 4.8: VGA signal timing [*Source:* Van den Bout]

start and end of a frame of lines to ensure that the monitor displays the lines between the bottom and top edges of the visible monitor area. The lines are sent to the monitor within a 15.25ms window. The vertical-sync drops low a minimum of 0.45ms after the last line

and stays low for 64μs. The first line of the next frame can begin a minimum of 1.02ms after the vertical-sync pulse ends. A single frame occupies 15.25ms of a 16.784ms interval. The other 1.534ms of the frame interval is the vertical blanking interval. Figure 4.8 illustrates this timing.[6]

To maintain the same signal frequency as the camera, we can use the camera's output horizontal- and vertical-sync signals. Since the first pixel to be classified is pixel 386, after the camera has scanned pixel 771, compensation must be made in the horizontal and vertical synchronization signals. The horizontal-sync signal has dropped low (active low) for the beginning of the 3rd line with the first pixel, 769. The classification for pixel number 386 is latched into the OUTPUT component as the camera is inputting pixel 772 in the first memory location in the dual port RAM. Therefore, the first pixel of the line to be displayed on the monitor is actually three pixels after the horizontal-sync signal has dropped.

The timing for the vertical-sync signal is also off. The vertical signal dropped when the camera began scanning the first line. The first line that we will output to the screen is the second line (since the border is ignored). The vertical-sync signal, when the output begins, is currently timed for the third line (beginning with pixel 769). This means that the vertical signal is timed for the beginning of the third line while we are outputting the pixels on the second line, one line behind. The compensation for these timing issues is made in the OUTPUT component. Two counters, one for the horizontal signal and another for the vertical signal with a synthesized 1-bit RAM array

accommodate the time lags. The timing diagram for the OV5017 CMOS camera is shown in Appendix B.

References

1. Castleman, Kenneth R., Digital Image Processing, Prentice Hall, New Jersey, 1996.

2. Bezdek, J. D., Pattern Recognition with Fuzzy Objective Function Algorithms, Plenum Press, New York, 1981.

3. Haskell, R. E., "Neuro-Fuzzy Classification and Regression Trees," *Proc. Third International Conference on Applications of Fuzzy Systems and Soft Computing*, Wiesbaden, Germany, October 5-7, 1998.

4. Jang, J.-S. R., Sun, C.-T., Mizutani, E., Neuro-Fuzzy and Soft Computing, Prentice-Hall, Inc., Upper Saddle River, NJ, 1997.

5. Haskell, R. E., "Regression Tree Fuzzy Systems," *Proc. ICSC Symposium on Soft Computing, Fuzzy Logic, Artificial Neural Networks and Genetic Algorithms*, University of Reading, Whiteknights, Reading, England, pp. B.1–B.6, March 26 - 28, 1996.

6. An overview of horizontal and vertical sync signals is given in Van den Bout, David, *VGA Signal Generation with the XS Board*. Technical reference for the XS prototype board.

7. Cole, Clint, Digilab Circuit Board User's Manual, Washington State University, Digilent, Inc., January 2000.

8. Scott, Stephen D., Samal, Ashok, and Seth, Sharad, "HGA: A Hardware-Based Genetic Algorithm," *Proceedings of the 1995 ACM/SIGDA Third International Symposium on Field-Programmable Gate Arrays*, pp. 53-59.

9. Schmit, Herman and Thomas, Don, "Hidden Markov Modeling and Fuzzy Controllers in FPGAs," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 214-221, 1995.

10. OmniVision Technologies, Inc. "OmniVision Confidential Preliminary Product Specification for the OV5017 CMOS Camera," October 27, 1997, version 1.6.

The most incomprehensible thing
about the world is that it is comprehensible

Albert Einstein

CHAPTER 5: SUMMARIZING A SOFTWARE DESIGN APPROACH WITH A RECONFIGURABLE SOLUTION

5.1 Conclusion

Traditionally, hardware/software co-design introduces the high-risk step of hardware/software splitting shown in Figure 1.3. If a change in specification or unexpected condition or situation occurs a revision in the hardware/software splitting step is often required. This change may require part or all of the software to be rewritten or ported to the newly selected hardware, an expensive and time-consuming process that, in some cases, leads to complete failure of the co-design project.

For software development, different software engineering and management techniques and tools, such as the Unified Development Process and Rational Rose, are available for a software project. These methods and tools offer a great deal of benefit to

software development that is *necessary* but not *sufficient* for equivalent hardware/software co-design projects.

A microprocessorless solution or an embedded system design in software that generates a reconfigurable microcontroller for the specific application was demonstrated. This method of hardware/software co-design leaves the "splitting" to the end, after the entire application has been designed. Additionally, the compiler is responsible for presenting optimal splitting decisions. Many years ago, ASICs were designed at the diffusion layer/p-n junction level of abstraction. After several years of collaboration and hundreds of millions of dollars of invested capital, compilers, such as VHDL, Verilog, and ABEL, compile, optimize, and place high-level hardware designs on a variety of FPGAs. This advancement dramatically increased the speed at which an embedded system could be developed and opened new doors for solving larger problems at a higher level with an ASIC. Similarly, a compiler that optimizes a high-level embedded system design and generates optimally configured microcontrollers when necessary offers a similarly dramatic advancement in the design of embedded systems.

Rapid advancements in transistor density have made smaller chips available with more gates at decreasing costs. Xilinx and other FPGA designers/manufacturers project significant increases each quarter following the current exponential trend. FPGAs can be connected together to form a daisy chain or field of chips for larger designs. Designs requiring multiple FPGAs will soon fit into fewer chips until finally a single chip, similar to the trend of hard disk space over the last 15 years.

Such a compiler introduces the opportunity to easily implement software algorithms in hardware. An example of a real-time edge detector designed using neuro-fuzzy decision trees was presented as a software algorithm and as a system implemented in hardware that was designed using a high-level software language and external memory, without a microcontroller. In this example, an inexpensive microprocessorless solution successfully employed a novel software technique to detect edges in real-time.

The reconfigurable solution offers other important advantages in addition to cost and time savings, among these are the use of software development disciplines in co-design projects and the prospect of compiling software implementations to a hardware solution. One such advantage is remote updatability. Field Programmable Gate Arrays can be updated via the Internet. Design changes can be made easily and inexpensively for bug-fixes and upgrades with enhancements. This expedites the introduction of new technology into the marketplace and adds to the already significant cost and time savings.

5.2 Future Research

This thesis provided examples and a proof-of-concept using a reconfigurable extended WnX microcontroller. A compiler and environment to determine optimal controller requirements and optimize place and routing over many chips for a general design would require a team of engineers and a significant investment.

In the beginning, software was implemented over arrays of expensive storage devices. Later, storage devices decreased in price and increased storage space in small increments. Over time, storage space has become abundant and inexpensive and is virtually transparent to a software designer. In the beginning, hardware was designed at

the p-n junction level. Later, microprocessors and microcontrollers were introduced, both extended and reduced with an assembly language and expensive FPGAs with few gates were produced.

Over time, high-level languages such as VHDL, Verilog, and ABEL were introduced to design, simulate, and synthesize hardware designs on the increasingly useful FPGA. Now, since FPGAs are being produced with rapidly increasing numbers of gates at low costs, the hardware/software split can be rejoined in a software environment.

APPENDIX A

STATISTICAL INFORMATION FOR HARD DRIVES FROM 1956 – PRESENT

SOURCE: NOVA SCOTIA'S ELECTRONIC ATTIC, ET AL.
URL: http://www.alts.net/ns1625/winchest.html ***(ACCESSED NOVEMBER 2000)***
MAINTAINED BY: IVAN SMITH

Prediction:
The cost for 128 kilobytes of memory
will fall below US $100 in the near future.

Creative Computing magazine
December 1981, page 6

The column headed "W" shows the warranty duration in years. The "Price of Drive" is the retail price, sales taxes extra. The "Cost per megabyte" is the retail price, all taxes included. Prices are in Canadian currency, except prices marked "U$" which are in United States currency. These examples have been selected from hard drives advertised for sale, to show the lowest available per-megabyte cost.

Source	Manufacturer	W y	Capacity	Price of Drive	Cost per MB
	1956				
Note 0	IBM		5 megabytes	U$50,000	U$10,000
	1980 January				
	Morrow Designs		26 megabytes	U$5000	U$193
	1980 July				
Note 34	North Star		18 megabytes	U$4199	U$233
	1981 September				
	Apple		5 megabytes	U$3500	U$700
	1981 November				
	Seagate		5 megabytes	U$1700	U$340
	1981 December				
Note 31	VR Data Corp.		6.3 megabytes	U$2895	U$460
Note 32	Morrow Designs		10 megabytes	U$2999	U$300
Note 33	Morrow Designs		10 megabytes	U$2949	U$295
Note 31	VR Data Corp.		19 megabytes	U$5495	U$289
Note 33	Morrow Designs		20 megabytes	U$3829	U$191
Note 33	Morrow Designs		26 megabytes	U$3949	U$152
Note 32	Morrow Designs		26 megabytes	U$3599	U$138
	1982 March				
	Xebec				U$260
	1983 December				
Note 35	Corvus		6 megabytes	U$1895	U$316
Note 35	Corvus		10 megabytes	U$2695	U$270

Note 35	Xcomp		10 megabytes	U$1895	U$190
Note 35	Corvus		20 megabytes	U$3495	U$175
Note 35	Davong		10 megabytes	U$1650	U$165
Note 35	Xcomp		16 megabytes	U$2095	U$131
Note 35	Davong		21 megabytes	U$2495	U$119
	1984 March				
Note 37	Percom/Tandon		5 megabytes	U$1399	U$280
Note 38	not known		5 megabytes	U$1349	U$270
Note 37	Percom/Tandon		10 megabytes	U$1699	U$170
Note 38	not known		10 megabytes	U$1599	U$160
Note 37	Percom/Tandon		15 megabytes	U$2095	U$140
Note 38	not known		15 megabytes	U$1999	U$133
Note 37	Percom/Tandon		20 megabytes	U$2399	U$120
Note 38	not known		20 megabytes	U$2359	U$118
	1984 May				
Note 36	Tecmar		5 megabytes	U$1495	U$299
Note 36	Corvus		6 megabytes	U$1695	U$283
Note 36	Corvus		11 megabytes	U$2350	U$214
Note 36	Comrex		10 megabytes	U$1995	U$200
Note 36	CTI		11 megabytes	U$1995	U$181
Note 36	Davong		10 megabytes	U$1645	U$165
Note 36	Corvus		20 megabytes	U$3150	U$158
Note 36	Davong		15 megabytes	U$2095	U$140
Note 36	Davong		21 megabytes	U$2495	U$119
Note 36	Pegasus (Great Lakes)		10 megabytes	U$1075	U$108
Note 36	Pegasus (Great Lakes)		23 megabytes	U$1845	U$80
	1985 July				
Note 30	First Class Peripherals	1	10 megabytes	U$710.00	U$71
	1987 October				
Note 39	Iomega		10 megabytes	U$899	U$90
Note 39	Iomega		20 megabytes	U$1199	U$60
Note 39	Iomega		40 megabytes	U$1799	U$45
	1988 May				
Note 1			20 megabytes	U$799	U$40
Note 1			30 megabytes	U$995	U$33
Note 1			45 megabytes	U$1195	U$27
Note 1			60 megabytes	U$1795	U$30
Note 1			250 megabytes	U$3995	U$16
	1989 March				
Note 56	Western Digital		20 megabytes	$899.00	$53
Note 56	Western Digital		40 megabytes	$1199.00	$36
	1989 September				
Note 11					$12
	1990 September				
Note 11					$9
	1991 September				
Note 11					$7
	1992 September				
Note 11					$4
	1993 September				
Note 11					$2

Source	Manufacturer	Way	Capacity	Price of Drive	Cost per MB
	1994 September				
Note 11					95¢
	1995 January				
Note 2	Seagate	5	1.0 gigabyte	$849	85¢
Note 2	Seagate	5	1.7 gigabytes	$1499	88¢
Note 2	Seagate	5	2.1 gigabytes	$1699	81¢
Note 2	Seagate	5	2.9 gigabytes	$2899	99¢
	1995 April				
Note 24			240 megabytes	$250.00	$1.26
Note 24			420 megabytes	$320.00	92.2¢
Note 24			520 megabytes	$380.00	88.4¢
Note 24			850 megabytes	$470.00	66.9¢
Note 24			1.0 gigabyte	$625.00	75.6¢
Note 24			1.2 gigabytes	$680.00	68.6¢
	1996 June 10				
Note 3	Western Digital	3	1.6 gigabytes	$399.99	29.5¢
	1996 August 14				
Note 4	IBM	3	1.76 gigabytes	$379.99	26.3¢
Note 4	Maxtor		2.0 gigabytes	$439.99	25.9¢
	1996 September				
Note 5	Quantum		2.5 gigabytes	$440.00	20.7¢
Note 5	Quantum		3.2 gigabytes	$469.00	17.3¢
	1997 August 13				
Note 6	Western Digital		2.1 gigabytes	$329.99	18.1¢
Note 6	Western Digital		3.1 gigabytes	$399.99	14.8¢
Note 6	Western Digital		4.0 gigabytes	$490.99	14.1¢
	1997 August 24				
Note 7	Western Digital	3	2.1 gigabytes	$279.99	15.3¢
Note 7	Western Digital	3	3.1 gigabytes	$329.99	12.2¢
Note 7	Maxtor	3	3.5 gigabytes	$359.99	11.8¢
Note 7	Maxtor	3	4.3 gigabytes	$439.99	11.8¢
Note 7	Western Digital	3	5.1 gigabytes	$459.99	10.4¢
	1997 September 5				
Note 8	Maxtor	3	7.0 gigabytes	$669.99	11.0¢
	1997 November 29				
Note 9	Western Digital		3.2 gigabytes	$289.00	10.4¢
Note 9	Quantum		3.2 gigabytes	$285.00	10.2¢
Note 9	Quantum		4.3 gigabytes	$379.00	10.1¢
Note 9	Western Digital		4.3 gigabytes	$365.00	9.76¢
Note 9	Quantum		6.4 gigabytes	$475.00	8.54¢
Note 9	Western Digital		6.4 gigabytes	$445.00	8.00¢
	1997 December 3				
Note 10	Western Digital	3	5.1 gigabytes	$449.99	10.1¢
Note 10	Quantum	3	6.4 gigabytes	$549.99	9.88¢
Note 10	Maxtor	3	5.2 gigabytes	$438.99	9.71¢
Note 10	Maxtor	3	7.0 gigabytes	$579.99	9.53¢
Note 10	Maxtor	3	8.4 gigabytes	$679.99	9.31¢

	1998 January 16				
Note 12	Western Digital	3	6.4 gigabytes	$529.99	9.52¢
Note 12	Quantum	3	4.3 gigabytes	$349.99	9.36¢
Note 12	Quantum	3	6.4 gigabytes	$479.99	8.63¢
Note 12	Maxtor	3	8.4 gigabytes	note 12	8.39¢
	1998 February 3				
Note 13	not known	3	5.2 gigabytes	$355.00	7.85¢
Note 13	not known	3	6.4 gigabytes	$435.00	7.82¢
	1998 April 2				
Note 14	Maxtor	3	5.1 gigabytes	$379.99	8.57¢
Note 14	Maxtor	3	4.3 gigabytes	$319.99	8.56¢
Note 14	Western Digital	3	6.4 gigabytes	note 14	7.43¢
Note 14	Quantum	3	6.4 gigabytes	$339.99	6.11¢
	1998 April 4				
Note 17	not known		5.2 gigabytes	$349.00	7.72¢
Note 15	Maxtor	3	4.3 gigabytes	note 15	7.63¢
Note 16	not known	3	6.4 gigabytes	$370.00	6.65¢
Note 16	not known	3	5.2 gigabytes	$300.00	6.63¢
Note 17	not known		9.0 gigabytes	$499.00	6.38¢
	1998 April 17				
Note 18	Fujitsu		4.3 gigabytes	$282.00	7.54¢
Note 18	Fujitsu		5.2 gigabytes	$331.00	7.32¢
Note 18	Fujitsu		6.4 gigabytes	$368.00	6.61¢
	1998 May 2				
Note 19	Seagate	1	6.4 gigabytes	$349.99	6.29¢
	1998 May 9				
Note 20	Seagate		6.4 gigabytes	$329.99	5.93¢
	1998 May 11				
Note 21	Fujitsu		3.2 gigabytes	$227.00	8.16¢
Note 21	Fujitsu		4.3 gigabytes	$257.00	6.87¢
Note 21	Fujitsu		5.2 gigabytes	$299.00	6.61¢
Note 21	Fujitsu		6.4 gigabytes	$328.00	5.89¢
	1998 June 6				
Note 22	Maxtor	3	5.7 gigabytes	$299.99	6.05¢
	1998 June 12				
Note 23	Quantum		4.3 gigabytes	$228.00	6.10¢
Note 23	Quantum		6.4 gigabytes	$298.00	5.35¢
	1998 July 15				
Note 25			5.2 gigabytes	$249.00	5.51¢
	1998 July 31				
Note 26	Western Digital IDE		5.1 gigabytes	$262.00	5.91¢
Note 26	Fujitsu IDE		5.2 gigabytes	$252.00	5.57¢
Note 26	Western Digital IDE		6.4 gigabytes	$294.00	5.28¢
Note 26	Western Digital IDE		8.4 gigabytes	$382.00	5.23¢
Note 26	Fujitsu IDE		6.4 gigabytes	$291.00	5.23¢
	1998 August 1				
Note 27	Western Digital EIDE		4.0 gigabytes	note 27	5.46¢
	1998 August 6				
Note 28	Western Digital EIDE	3	5.1 gigabytes	note 28	4.64¢
	1998 August 14				
Note 29	Fujitsu		6.4 gigabytes	$289.00	5.19¢

	1998 August 26				
Note 40	Seagate	1	6.4 gigabytes	$279.99	5.03¢
	1998 September 1				
Note 41	Maxtor UDMA	3	8.4 gigabytes	$379.99	5.20¢
Note 41	Maxtor UDMA		6.8 gigabytes	$279.99	4.74¢
	1998 September 10				
Note 42	Western Digital EIDE	3	5.1 gigabytes	note 42	4.79¢
	1998 October 1				
Note 43	Quantum		6.4 gigabytes	note 43	4.26¢
	1999 February 12				
Note 44	Quantum		8.0 gigabytes	$299.99	4.31¢
	1999 February 26				
Note 45	Maxtor		8.4 gigabytes	see note 45	3.77¢
Note 46	Quantum		8.0 gigabytes	see note 46	3.65¢
	1999 February 27				
Note 47	Quantum		19.2 gigabytes	$512.46	3.07¢
	1999 March 1				
Note 48	Fujitsu Ultra DMA	3	8.4 gigabytes	$253.00	3.46¢
Note 48	Fujitsu Ultra DMA	3	10.2 gigabytes	$299.00	3.37¢
	1999 March 3				
Note 49	Fujitsu Ultra DMA	3	8.4 gigabytes	$235.00	3.22¢
Note 49	Fujitsu Ultra DMA	3	10.2 gigabytes	$285.00	3.21¢
	1999 April 1				
Note 50	Fujitsu UDMA		10.2 gigabytes	$279.00	3.15¢
Note 50	Fujitsu UDMA		8.4 gigabytes	$229.00	3.14¢
	1999 May 21				
Note 51	Fujitsu UDMA		6.4 gigabytes	$179.99	3.23¢
	1999 May 27				
Note 52	Fujitsu UDMA		10.2 gigabytes	$245.00	2.76¢
Note 52	Fujitsu UDMA		8.4 gigabytes	$198.00	2.71¢
Note 52	Fujitsu UDMA		17.3 gigabytes	$369.00	2.45¢
	1999 May 28				
Note 53	Maxtor UDMA	3	10.0 gigabytes	$249.99	2.88¢
	1999 July 21				
Note 54	Maxtor Ultra DMA	3	8.4 gigabytes	$199.99	2.74¢
	1999 July 30				
Note 55	Fujitsu UDMA		6.4 gigabytes	$139.99	2.63¢
	1999 September 25				
Note 57	Not known		10.2 gigabytes	note 57	1.85¢
	1999 October 1				
Note 58	Quantum CX UTA 66		10.2 gigabytes	$199.00	2.24¢
Note 58	Quantum KA 7200 rpm		13.6 gigabytes	$249.00	2.11¢
Note 58	Western Digital		20.0 gigabytes	$359.00	2.06¢
Note 58	Western Digital 7200 rpm		27.3 gigabytes	$489.00	2.06¢
Note 58	Quantum CX UTA 66		13.6 gigabytes	$219.00	1.85¢
	1999 December 1				
Note 59	Western Digital IDE		20.5 gigabytes	$398.00	2.23¢

Note 59	Quantum IDE	18.2 gigabytes	$348.00	2.20¢
Note 60	Mfgr? UDMA	10.2 gigabytes	$189.00	2.13¢
Note 59	Fujitsu IDE	10.2 gigabytes	$189.00	2.13¢
Note 59	Fujitsu IDE	13.0 gigabytes	$208.00	1.84¢
Note 60	Mfgr? UDMA	13.0 gigabytes	$195.00	1.73¢
Note 59	Fujitsu IDE	20.4 gigabytes	$299.00	1.69¢
Note 59	Fujitsu IDE	17.3 gigabytes	$248.00	1.65¢
Note 59	Fujitsu IDE	27.3 gigabytes	$388.00	1.63¢
Note 60	Mfgr? UDMA	17.3 gigabytes	$225.00	1.50¢

From here on, the cost of hard drives will be stated per gigabyte (below) instead of per megabyte (above)

Source	Manufacturer	Way	Capacity	Price of Drive	Cost per GB	Cost per MB
	2000 February 1					
Note 61	Mfgr? UDMA		10.2 gigabytes	$175.00	$19.73	1.97¢
Note 62	Fujitsu		20.4 gigabytes	$299.00	$16.86	1.69¢
Note 62	Fujitsu		13.6 gigabytes	$199.00	$16.83	1.68¢
Note 64	Mfgr?		12.9 gigabytes	$187.99	$16.76	1.68¢
Note 63	Fujitsu		13.6 gigabytes	$197.80	$16.73	1.67¢
Note 61	Mfgr? UDMA		13.0 gigabytes	$186.00	$16.45	1.65¢
Note 62	Fujitsu		17.3 gigabytes	$238.00	$15.82	1.58¢
Note 62	Fujitsu		27.3 gigabytes	$375.00	$15.80	1.58¢
Note 63	Fujitsu		17.3 gigabytes	$232.30	$15.44	1.54¢
Note 61	Mfgr? UDMA		17.3 gigabytes	$215.00	$14.29	1.43¢
Note 64	Mfgr?		20.4 gigabytes	$211.99	$11.95	1.20¢
	2000 April 1					
Note 69	IBM		20.5 gigabytes	$279.00	$15.65	1.57¢
Note 69	Maxtor		15.2 gigabytes	$199.00	$15.06	1.51¢
Note 70	Maxtor 7200 rpm		20.0 gigabytes	$259.00	$14.89	1.49¢
Note 70	Maxtor UDMA		15.0 gigabytes	$192.00	$14.72	1.47¢
Note 70	Seagate UDMA		17.2 gigabytes	$218.00	$14.58	1.46¢
Note 70	Seagate UDMA		28.0 gigabytes	$349.00	$14.33	1.43¢
Note 68			17.3 gigabytes	$215.00	$14.29	1.43¢
Note 70	IBM UDMA 5400 rpm		20.3 gigabytes	$245.00	$13.88	1.39¢
Note 70	Maxtor UDMA		17.0 gigabytes	$204.00	$13.80	1.38¢
Note 70	Maxtor 7200 rpm		27.0 gigabytes	$320.00	$13.63	1.36¢
Note 68			20.4 gigabytes	$239.00	$13.47	1.35¢
Note 70	Maxtor UDMA		36.5 gigabytes	$411.00	$12.95	1.30¢
Note 70	Maxtor UDMA		27.0 gigabytes	$299.00	$12.74	1.27¢
Note 70	Western Digital UDMA		20.0 gigabytes	$218.00	$12.54	1.25¢
Note 70	Maxtor UDMA		20.0 gigabytes	$217.00	$12.48	1.25¢
Note 70	Maxtor UDMA		30.0 gigabytes	$308.00	$11.81	1.18¢

2000 May 12

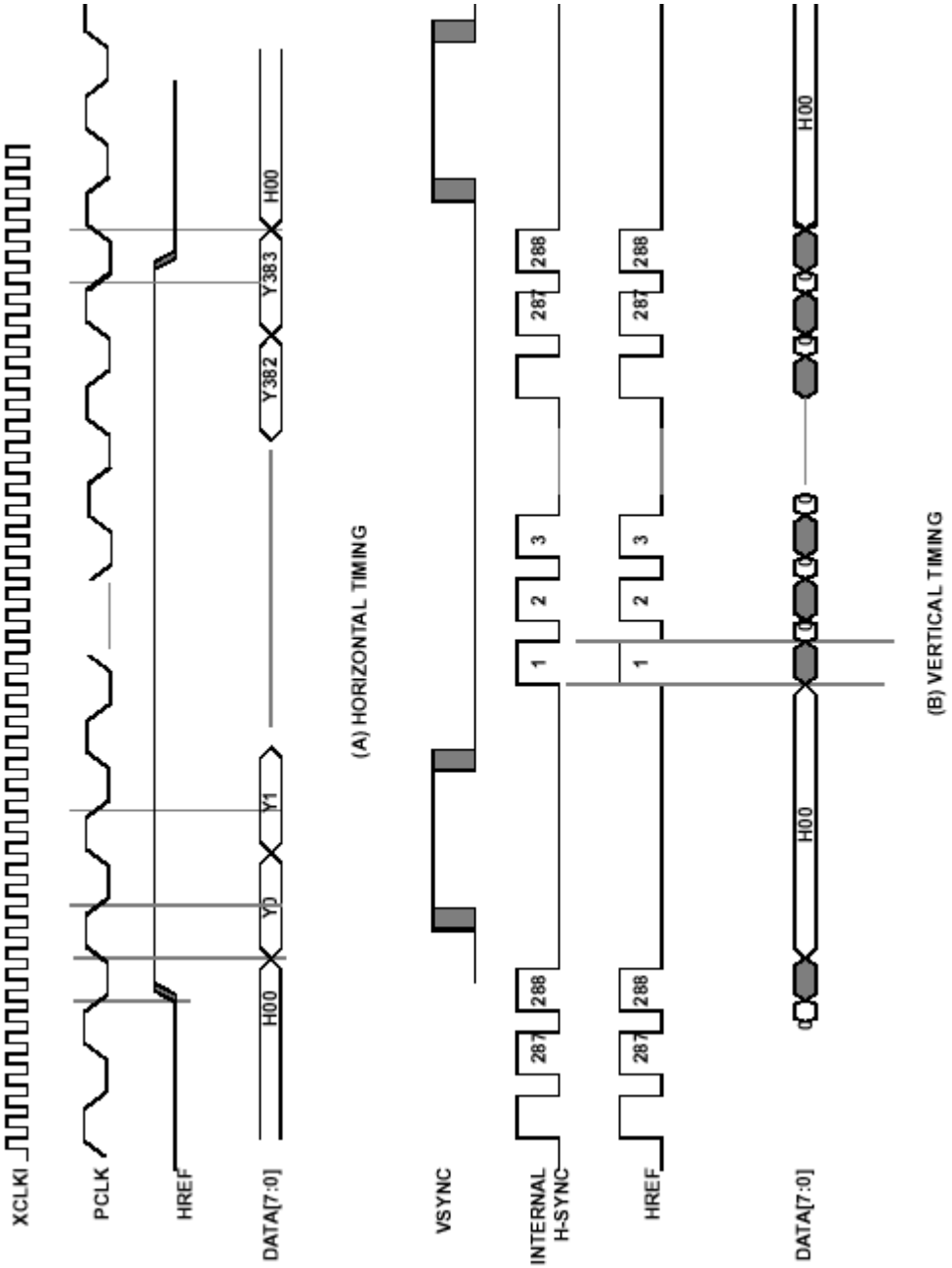
Note 65	Western Digital Ultra ATA/66 5400 rpm		13.6 gigabytes	$179.99	$15.22	1.52¢
Note 65	Maxtor UDMA/66 7200 rpm		30.0 gigabytes	$319.99	$12.27	1.23¢
Note 66	Maxtor UDMA/66 7200 rpm	3	40.0 gigabytes	$399.99	$11.50	1.15¢
	2000 June 2					
Note 67	Maxtor UDMA/66 5400 rpm	3	15.0 gigabytes	$189.99	$14.57	1.46¢
	2000 August 1					
Note 73	Samsung		15.0 gigabytes	$162.00	$12.42	1.24¢
Note 74	Maxtor IDE 7200rpm		30.5 gigabytes	$298.00	$11.24	1.12¢
Note 73	Samsung		20.0 gigabytes	$175.00	$10.06	1.01¢
	2000 August 19-20					
Note 72	Maxtor 7200rpm 9ms		40.9 gigabytes	$388.00	$10.91	1.09¢
Note 71	Maxtor 5400rpm		15.3 gigabytes	$144.00	$10.82	1.08¢
Note 72	Maxtor 7200rpm 9ms		30.7 gigabytes	$278.00	$10.41	1.04¢
Note 71	Maxtor 5400rpm		20.4 gigabytes	$164.00	$9.25	0.925¢
Note 71	Maxtor 5400rpm		30.7 gigabytes	$214.00	$8.02	0.802¢
	2000 August 25					
Note 75	Maxtor 5400rpm		15.0 gigabytes	$149.99	$11.50	1.15¢
Note 75	Maxtor 7200rpm		40.0 gigabytes	$349.99	$10.06	1.01¢
Note 75	Maxtor 7200rpm UDMA/66		30.0 gigabytes	$249.99	$9.58	0.958¢

The right-hand column (below), states the storage capacity, in megabytes, available at a retail cost of one cent.

"Price of drive" is the store price, excluding sales tax. "Cost per gigabyte" and "Megabytes for one cent" are stated with 15% sales tax included (purchaser's cost).

Source	Manufacturer	Wy	Capacity	Price of Drive	Cost per GB	MB for 1¢
	2000 October 27					
Note 76	Maxtor 7200rpm		30.7 gigabytes	$244.00	$9.14	1.09
Note 76	Maxtor 7200rpm		40.9 gigabytes	$318.00	$8.94	1.12
Note 76	Maxtor 5400rpm		61.4 gigabytes	$398.00	$7.45	1.34
Note 76	Maxtor 5400rpm		81.9 gigabytes	$518.00	$7.27	1.37
Note 76	Maxtor 5400rpm		30.7 gigabytes	$194.00	$7.27	1.38
Note 76	Maxtor 5400rpm		40.9 gigabytes	$254.00	$7.14	1.40
	2000 November 1					
Note 77	Maxtor 7200rpm UDMA/66		30.0 gigabytes	note 77	$7.88	1.27

APPENDIX B
TIMING DIAGRAM FOR OV5017 CMOS CAMERA
SOURCE: CONFIDENTIAL PRELIMINARY PRODUCT SPECIFICATION OCTOBER 1997 V1.6



APPENDIX C
SERIAL CODE CONVERSION FROM BINARY TO BCD

SOURCE: PETER ALFKE AND BERNIE NEW, XILINX APPLICATION NOTE XAPP 029, OCTOBER 29, 1997 (VERSION 1.1)

Binary-to-BCD conversion is performed in a modified shift register that successively doubles its BCD contents. The binary data is shifted into the converter serially, MSB first. Subsequent bits are entered into the shift register to fill the LSB vacated by the doubling. The conversion is complete when all bits of the binary input have been entered, at which time the BCD result is available in parallel form. Each input bit will have been doubled and redoubled to regain its original binary weight, but in BCD format.

To remain a valid BCD number when doubled, a BCD digit of 5 or greater must not just be shifted, but must be converted into the proper BCD representation of its doubled value; along with a 1 being shifted into the next higher digit, a 5 is converted into a 0, a 6 into a 2, a 7 into a 4, an 8 into a 6, and a 9 into an 8.

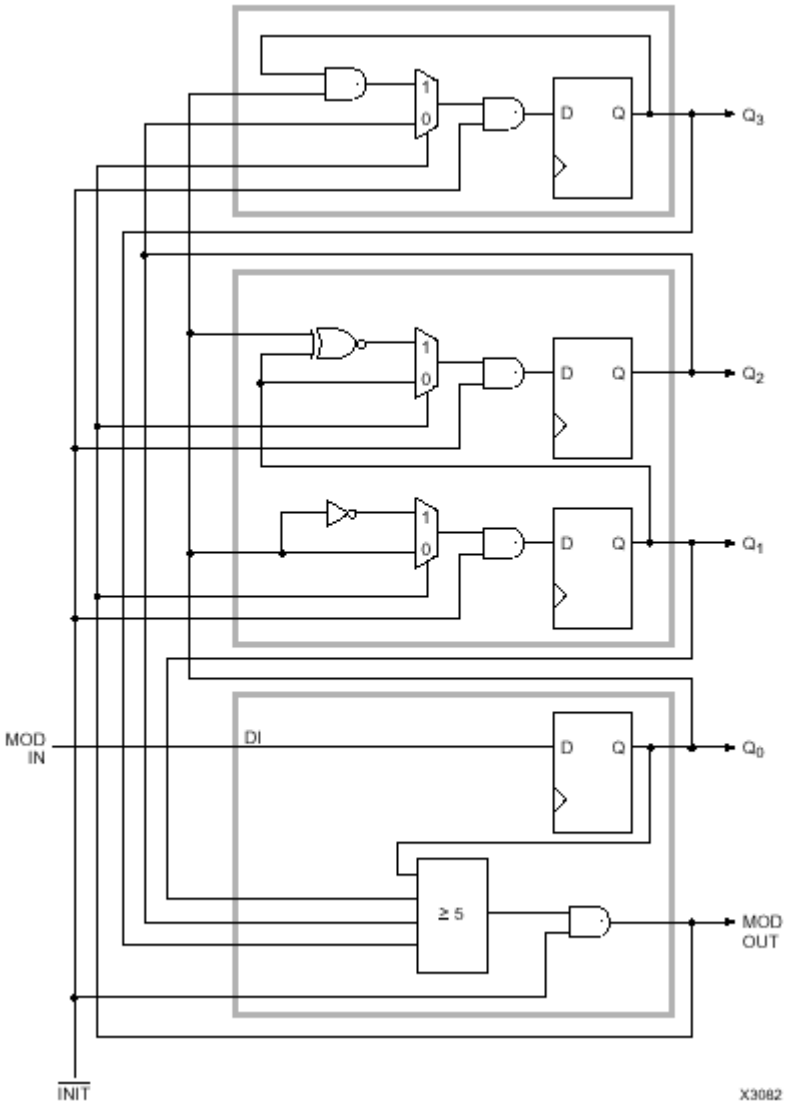
The binary-to-BCD converter requires three CLBs for each BCD digit in the output. To start a new conversion, all bits must be cleared.

An Example

1011 0110 (B 6 Hex, 182 Decimal)
Requires 3 Decimal Digits to Represent 0xB6

0000 0000 0000	0000 0000 0001	0000 0000 0010
0000 0000 0101	0000 0001 0001	0000 0010 0010
0000 0100 0101	0000 1001 0001	0001 1000 0010
		1 8 2

Component Diagram follows on the Next Page


1
0
D
Q
Q_3
Q_2
Q_1
Q_0
MOD
IN
DI
≥ 5
MOD
OUT
INIT
X3082