SYNERGIES OF TEACHING ASSEMBLY AND C IN A JUNIOR MICROPROCESSORS COURSE

Arlen Planting and Sin Ming Loo
Electrical & Computer Engineering Department
Boise State University

Abstract

As part of an effort to update its core computer engineering courses, Boise State University has developed course material to effectively transition students in a Microprocessors course through both the assembly and C languages. It was found that teaching both languages in the same course provides benefits not found in teaching them separately. The material has been developed to promote both a thorough understanding of microprocessors, and greater productivity that allows students to do more intriguing and relevant projects. The course presents just enough C, at a very low level and in a specific topic order, to enable the students to better comprehend microprocessors and how they can control a broad range of devices. The updated Microprocessors course is currently in its fourth iteration.

Introduction

The C programming language is increasingly being utilized in development of embedded systems and ultra-small microcontrollers that were previously the domain of assembly language-only programming. Teaching assembly only in a Microprocessors course does not provide students the skills they are likely to need in the workplace [1], and the time required to produce code for each new device in assembly results in the course becoming more software-oriented rather than focusing on the hardware and devices. However, using the C language only is not considered practical for teaching microprocessors since assembly is the language of the processor and thus is necessary for understanding how the microprocessor works. Simply rewriting device code in C without applying software engineering principles [2] yields poor quality code that is difficult to maintain and cannot be readily targeted to other platforms. However, by selectively applying some of the object oriented principles [3] that can be found in the Linux kernel and device drivers, the C programming language can provide an effective solution for programming many of these small devices. Principles of layering, data encapsulation and abstraction can help make the code more readable, maintainable and portable.

It should be noted that the intent is not to write C code as translated assembly, which is hard to read, hard to maintain and offers little benefit over assembly code. By effectively utilizing the facilities of the C language, many assembly language routines can be reduced to very small and elegant solutions. For students who lack the insight into how to write assembly programs but are proficient at C or Java, compiling a program in C is a method of seeing how a program can be written in C. Writing code at the lowest level to access devices is generally very tedious in either language, but by providing appropriate abstractions this code can be isolated in layers to allow the higher level more freedom to solve problems with less consideration for hardware details. This also provides for easier retargeting to other platforms.

Consequently, Boise State University (BSU) decided to update its junior level Microprocessors course by incorporating the C programming language in addition to assembly. It was not practical - or necessary - to teach the entire C programming language in order to significantly enhance software skills beyond what is achieved with the assembly language alone. Following a heuristic approach, BSU developed course material to transition students in a Microprocessors course through both assembly and C, and found that overlapping the

teaching of both languages in the same course is more beneficial teaching them separately. The course presents the C language at a very low level, with selected topics presented in a specific order to enhance understanding of microprocessors and their ability to control a wide range of devices.

Boise State University has an ABET-accredited electrical engineering program with computer engineering as an option. Both electrical engineering and computer science students take the Microprocessors course after they have taken Introduction to Computer Science (basic software skills and object oriented programming with Java) and Digital Systems (digital logic). The ECE 332/332L Microprocessors course at BSU covers microprocessor architecture, software development tools, and hardware interfacing with emphasis on 16- and 32-bit microprocessor systems. Machine and assembly language programming, instruction set, addressing modes, programming techniques, memory systems, I/O interfacing, and interrupt handling are among the topics studied with practical applications in data acquisition, control, and interfacing.

An experimental course addressing the usage of the C programming language for embedded applications was undertaken in Spring 2007 to investigate methods of incorporating the C language in the electrical engineering curriculum. The experimental course approach included an accelerated presentation of the C language directed to specific course objectives, and the use of object oriented principles with low level languages. The teaching philosophy demonstrated by this model was subsequently used to update the Microprocessors course in Fall 2007. Further refinements have been made in two subsequent offerings of the course in 2008, and in the current semester.

Microprocessors Course Approach

In the ECE 332/332L Microprocessors course at BSU, basic microprocessor concepts are first explored with assembly language then revisited and expanded upon using C. A modern development platform consisting of an FPGA and a soft core processor with a MIPS-like design were selected to implement the teaching of the C programming language in addition to assembly in the updated Microprocessors course. The use of FPGAs in place of traditional instructional platforms has been an important part of the process of updating the computer engineering curriculum at BSU [4,5]. For the Microprocessors course, the FPGA is used to instantiate a soft-core processor. The reconfigurability of the FPGA with soft-core processor allows the instructor to quickly create different configurations for various labs and projects.

The Altera DE2 was selected as the FPGA development board for updating the Microprocessors course, with the Nios II processor used for software development on the DE2. The Nios II microprocessor system contains a processor (with a control unit and general purpose registers) and attached external memory. The Nios II processor has thirty-two 32-bit general purpose registers, twenty-two of which are available for general use (the remaining ten registers are reserved for a specific purpose).

One of the desirable features of the Altera DE2 with Nios II processor is that it has a RISC architecture closely approximating MIPS. A RISC microprocessor provides several educational advantages. The fixed length instructions for RISC platforms are simpler to learn than the variable-length instructions for CISC platforms. Since many of the instructions available with the CISC are not applicable to the basic microprocessors course, the significantly smaller set of instructions provided for a RISC platform was considered more appropriate for teaching basic microprocessor concepts. The students will also use RISC in the senior level Computer Architecture course.

The instruction set for the Nios II platform used for the Microprocessors course is comprised of just 84 instructions. To further

simplify the learning process these instructions were categorized by task, which effectively reduces the number of core instructions that the students need to learn by about 75%. The remaining instructions represent variations on these core instructions. The instructions were divided into four basic groups that address the majority of all applications: instructions that move data from memory to registers (MR), operate on register values and place results back into a register (RR), move data from registers to memory (RM), and change the flow (FC) of the instruction sequence.

Once the students have been familiarized with registers, cache, memory, and instructions to move and manipulate data in assembly language, the course is transitioned to the C language. Concepts from C such as data structures, unions and bit fields provide capabilities beyond what is available in assembly. The classic text for C programmers "The C Programming Language" (K&R) [6] was adapted for use as a reference for this portion of the course. Supplementary material was necessary since much of the focus of K&R is on algorithms, and instruction on algorithms in this course is minimal because the focus is on devices. Thus the concepts presented in K&R were approached in the course from a data viewpoint, e.g. pointers were treated as another data type. Data types in C were compared to equivalent data types in assembly.

The synergies from teaching the C language in conjunction with assembly proceed from the use of C at a low level. In the continuum of programming languages, the C language can span the gap between a high level language such as Java and the lowest level (assembly) language. This versatility can confound the students if they do not grasp that the Microprocessors course utilizes the C language at a low level, just a layer above assembly. Since all students taking Microprocessors have previously had Java, many of them initially believe there is nothing new to be learned with C. Those who have also learned the C language often have difficulty learning C concepts at the lowest level. The challenge for these students is to realize that knowledge of the C language as a high level language does not necessarily translate to a working knowledge of C at a low level.

Bit manipulation is one concept that benefits from the introduction of the C language. The manipulation of bits is generally the realm of hardware devices. The process of bit twiddling using techniques such as bit shifting and masking has traditionally been done in assembly language, and moving that code to C does not yield any benefits. However, this process is reduced to fairly straightforward code in C with the combined usage of bit structures and unions. The introduction of the constructs of pointers, structures and unions thus can reduce the tedium of dealing with the signals of connected hardware devices. Since bit structures can be platform-dependent, their usage is best restricted to lower platform-dependent layers.

Teaching C *in addition to* assembly provides advantages that would not be provided by simply replacing assembly language with C. In either language, working at the device level requires becoming familiar with the processor and the address space. The concept of pointers must also be learned in either case (though pointers in assembly languages may not be recognized as such in the same context as C). Pointers are the most difficult concept to learn in C. Teaching the concepts of pointers in assembly first, observing the instructions involved, and then translating that knowledge to implementation in C simplifies understanding the concept of pointers in C. Once pointers have been learned in assembly, the only differences that need to be learned in C are syntactic. Pointers are the primary reason that C can replace assembly language for device level code.

Other synergies between the assembly and C languages are observed in relation to understanding registers, processor architecture, and processor address space. In all processors, data manipulation is accomplished at the

register level. That fact is completely apparent in assembly, whereas the C language abstracts away the concept of registers and makes it appear that everything is done in memory. Therefore, the introduction of the *register* keyword in C is difficult to understand until one becomes familiar with the concept of registers in assembly. Doing low (device) level microprocessor development in C is difficult to do without a good understanding of the processor architecture and the processor address space (including the program, data, stack and devices). It can be argued that understanding the assembler for a processor before trying to do work with C is a definite advantage, which is why overlapping the instruction of both assembly and C languages provides synergism.

Integration of Assembly and C

Teaching both assembly and C in the same course can be effectively accomplished only by integrating selected topics into a unified whole directed toward achieving the course goals. Choices must be made as to which topics to present and in what context and order, and the presentation needs to be coordinated to provide a seamless transition between the two languages. In order to accomplish this, assembly is presented from a different perspective than is traditionally used, with emphasis on how to interface assembly and C. Assembly is taught using an object oriented approach focused more on utilizing the instructions than on the details of the instructions. The concept of abstraction is introduced in assembly, and the C language is subsequently presented as a means to further abstract assembly. The subset of the C language used in the course was selected for manipulating bits in order to control devices found in small microprocessor systems.

In addition to basic microprocessor concepts typically covered in assembly (e.g. memory usage, addressing, strings, etc.), several topics more traditionally addressed in C are included in the assembly portion of the course. Modularization, usage of functions, and the abstraction process are foundational concepts that are introduced early in the course. Though one may question the need for these advanced concepts in assembly, learning them at an early stage provides the framework for development of well-designed code that is appropriately layered with meaningful abstractions and appropriate usage of data encapsulation.

When the Microprocessors course was first updated, pointers were introduced after basics of the C programming language had been presented. As the course has evolved, teaching of the concept of pointers has been moved progressively earlier in the course until now it is introduced early in the assembly portion. The word ‘pointers’ is purposely used when discussing addresses to familiarize the students with the underlying mechanism for how a pointer is utilized by addressing. Early and repeated exposure to pointers reinforces understanding of the concept so the students are more comfortable with pointers when they appear in the C language portion of the course.

On the other hand, introduction of several topics was considered more suitable for the C language. Though structures can be taught using assembly, they are much easier to understand and utilize in C. For that reason, structures, unions and bit fields are not introduced until the C portion of the course. The C compiler can be considered as the ultimate macro processor, providing abstractions beyond what can be easily done by macros and functions in assembly. The compiler will generate the code for bit fields in assembly, eliminating the need for the students to hand write the code. The culmination of these topics involves combining bit fields and unions to easily manipulate the signals of externally attached devices.

Supplementary Examples

Some of the primary course materials developed for the Microprocessors course involve examples to help students understand the workings of the processor in the transition

from assembly to C. The Nios II incorporates a compiler that internally translates C code to assembly, in either an un-optimized or optimized format. The un-optimized assembler code generated is most useful for debugging purposes, while the optimized code provides an example of efficient coding. During the learning process, the optimization feature should be turned off since the effects on the code when working with optimizing compilers can easily confuse novices. If a programmer needs to produce highly optimized code, starting from scratch in assembly can be daunting; it is better to start with C, look at the optimized assembly, and proceed from there.

Several examples of classic cases provided to the students are included in this section, including 1) sum of integer array, 2) call by value methodology, 3) bit manipulation, and 4) pointers. The examples illustrate that compiled C, if optimized, can be virtually the same as efficient code written in assembly. Understanding assembly and seeing the results of the compiler optimization of C code can ultimately help the students develop better solutions that result in a significant reduction in code.

Sum of Integer Array

To facilitate the comparison between C and assembly, we start with a relatively simple algorithm that can be easily coded in both languages. The problem chosen is to write a function that is passed an array of integers and a count of numbers passed, and returns the sum of those integers. Figure 1 shows a high-level routine written in C that will call the *sum* function. We then write the code to solve this problem separately in C and assembly, and compare the code produced by the C compiler to the code written in assembly. The results illustrate how the C compiler deals with registers and memory. To get a better feel for how the C compiler abstracts the concept of registers, the generated machine code is first done without optimization followed by observing the code generated when optimizations are enabled.

```
#include <stdio.h>
#include "sum.h"

int main()
{
  int Values[] = {3,2,7,9,4};
  int nbr;

  nbr = sum(sizeof(Values)/sizeof
        (Values[0]), Values);

  printf("Sum: %d\n", nbr);

  return 0;
}
```

Figure 1. Calling *sum* routine in C.

Figure 2 shows efficient assembly language code to solve the problem. Note that this solution allocates no memory; the only memory it accesses is the array of integers passed. Since the memory access is minimal, the assembly code is highly efficient. Figure 3 displays the resulting code from the view of the debugger that is disassembling the machine code.

Code is then written in C (Figure 4) to solve the same problem. (Note the usage of register hints to the compiler.) Figure 5 displays the resulting un-optimized assembly code produced by the debugger's disassembler. Because the un-optimized compilation is an abstraction of variables, the variable values are associated with memory rather than registers. This results in a large number of data movements between memory and registers. When optimization is enabled (Figure 6), virtually all extraneous movement of data between registers and memory is eliminated. Optimization reduces the code by approximately 65% in this case.

Register hints and optimizations provide students first-hand experience in how coding techniques in C affect the underlying generation of assembly/machine code. By comparing optimized and un-optimized code, the various abstractions of variables become apparent. In the un-optimized case, variables are always

```
# sum.s

.text

# *********************************************
# Register usage:
#
#   r2: sum (return value)
#   r3: temp value
#   r4: passed count
#   r5: passed pointer to values
#
# *********************************************

.global sum
sum:

   mov       r2,r0         # initialize sum

   for:

      beq     r4,r0,for_end
      ldw     r3,0(r5)      # get next value
      add     r2,r2,r3      # add to sum
      addi    r5,r5,4       # position to next value
      subi    r4,r4,1       # decrement count
      br      for

   for_end:

   ret                      # return with sum in r2

.data

.end
```

Figure 2. Implementation of *sum* routine in assembly.

```
0x00020270 <sum>:      mov r2,zero

0x00020274 <for>:      beq  r4,zero,0x2028c <for_end>
0x00020278 <for+4>:    ldw  r3,0(r5)
0x0002027c <for+8>:    add  r2,r2,r3
0x00020280 <for+12>:   addi r5,r5,4
0x00020284 <for+16>:   addi r4,r4,-1
0x00020288 <for+20>:   br   0x20274 <for>

0x0002028c <for_end>:  ret
```

Figure 3. Disassembled memory snapshot for *sum.s*.

```
#include "sum.h"

int sum(int count, int *values)
{
    register int i;
    register int sum = 0;

    for (i=0; i<count; i++)
        sum+=values[i];

    return sum;
}
```

Figure 4. Implementation of *sum* routine in C.

```
{
0x00020270 <sum>:     addi sp,sp,-20
0x00020274 <sum+4>:   stw  fp,16(sp)
0x00020278 <sum+8>:   mov  fp,sp
0x0002027c <sum+12>:  stw  r4,0(fp)
0x00020280 <sum+16>:  stw  r5,4(fp)
    register int i;
    register int sum = 0;
0x00020284 <sum+20>:  stw  zero,12(fp)

    for (i=0; i<count; i++)
0x00020288 <sum+24>:  stw  zero,8(fp)
0x0002028c <sum+28>:  ldw  r2,0(fp)
0x00020290 <sum+32>:  ldw  r3,8(fp)
0x00020294 <sum+36>:  bge  r3,r2,0x202c8 <sum+88>
0x00020298 <sum+40>:  ldw  r2,8(fp)

0x0002029c <sum+44>:  muli r3,r2,4
0x000202a0 <sum+48>:  ldw  r2,4(fp)
0x000202a4 <sum+52>:  add  r2,r3,r2
0x000202a8 <sum+56>:  ldw  r2,0(r2)
0x000202ac <sum+60>:  ldw  r3,12(fp)
0x000202b0 <sum+64>:  add  r3,r3,r2
0x000202b4 <sum+68>:  stw  r3,12(fp)

0x000202b8 <sum+72>:  ldw  r2,8(fp)
0x000202bc <sum+76>:  addi r2,r2,1
0x000202c0 <sum+80>:  stw  r2,8(fp)
0x000202c4 <sum+84>:  br   0x2028c <sum+28>
        sum+=values[i];

    return sum;
0x000202c8 <sum+88>:  ldw  r2,12(fp)
}
0x000202cc <sum+92>:  ldw  fp,16(sp)
0x000202d0 <sum+96>:  addi sp,sp,20
0x000202d4 <sum+100>: ret
```

Figure 5. Compiled *sum.c* (un-optimized).

```
{
    register int i;
    register int sum = 0;
0x00020278 <sum>:    mov  r3,zero

    for (i=0; i<count; i++)

0x0002027c <sum+4>:  bge  zero,r4,0x20294 <sum+28>
0x00020280 <sum+8>:  ldw  r2,0(r5)
0x00020284 <sum+12>: addi r4,r4,-1
0x00020288 <sum+16>: addi r5,r5,4

        sum+=values[i];

0x0002028c <sum+20>: add  r3,r3,r2
0x00020290 <sum+24>: bne  r4,zero,0x20280 <sum+8>

    return sum;
}

0x00020294 <sum+28>: mov  r2,r3
0x00020298 <sum+32>: ret
```

Figure 6. Compiled *sum.c* (optimized).

backed by memory whereas optimization typically removes the backing of memory and leaves much of the solution to be accomplished in registers. For students accustomed to a high level language such as Java, observing the assembly code generated by an efficient compiler can be an effective method of transitioning from a highly abstracted environment and refocusing on handling details at a low level where few abstractions are provided.

Call By Value Methodology

Another issue that is difficult for students to understand is how parameters are passed to functions. Seeing and understanding the resulting assembly code underlying C can shed

some light on how C sets up parameters to be sent to a function. Since C is a call by value language, the question might arise as to how to pass literal values vs. variables to the same function. In the case where literals are passed, the literal value is moved directly into the calling register. In the case of a call that references a variable, the content of the variable is copied into the calling register. The called routine (Figure 7) does not see the two calls differently. All of the work to accommodate the different call types is done by the compiler at compile time.

```
int add_c(int x, char y)
{
    return x + y;
}
```

Figure 7. C language *add_c.c* routine.

Figure 8 shows code produced for the *add_c* function (un-optimized). Note the different handling of the variables x (type int) and y (type char) when processed by machine instructions (<add_c+24> and <add_c+20>, respectively). The optimized code is shown in Figure 9.

```
{
0x00020214 <add_c>:    addi sp,sp,-12
0x00020218 <add_c+4>:  stw  fp,8(sp)
0x0002021c <add_c+8>:  mov  fp,sp
0x00020220 <add_c+12>: stw  r4,0(fp)
0x00020224 <add_c+16>: stb  r5,4(fp)
    return x + y;
0x00020228 <add_c+20>: ldbu r2,4(fp)
0x0002022c <add_c+24>: ldw  r3,0(fp)
0x00020230 <add_c+28>: add  r2,r2,r3
}
0x00020234 <add_c+32>: ldw  fp,8(sp)
0x00020238 <add_c+36>: addi sp,sp,12
0x0002023c <add_c+40>: ret
```

Figure 8. *add_c* function (un-optimized).

Comparing the two different methods of calling this function (Figures 10 and 11) clearly illustrates the call by value feature of the C programming language. When the called function is called, it expects that the passed values are contained in the calling registers.

```
{
    return x + y;
0x00020214 <add_c>:   andi r2,r5,255
}
0x00020218 <add_c+4>: add  r2,r2,r4
0x0002021c <add_c+8>: ret
```

Figure 9. *add_c* function (optimized).

```
    c = add_c(12, 34);
0x0002025c <main+28>: movi r4,12
0x00020260 <main+32>: movi r5,34
0x00020264 <main+36>: call 0x20214 <add_c>
0x00020268 <main+40>: stw  r2,0(fp)
```

Figure 10. Calling *add_c* function with literal values.

```
c = add_c(a, b);
0x00020298 <test+24>: ldbu r5,4(fp)
0x0002029c <test+28>: ldw  r4,0(fp)
0x000202a0 <test+32>: call 0x20214 <add_c>
0x000202a4 <test+36>: stw  r2,8(fp)
```

Figure 11. Calling *add_c* function with variable arguments.

Bit Manipulation

The ability to manipulate data at the bit level (for controlling and pulling data off devices) for low level coding is very important when dealing with hardware devices. Setting a bit can turn an LED (or any other electronic device) on or off; getting a bit can determine whether a switch is on or off. Being able to manipulate individual bits within a hardware register (bit fields) is a useful concept. Assembly is used to understand the low level process of manipulating bits within a word.

To illustrate working with assembly and C for bit manipulations, we create functions that are passed a 32-bit word and a bit value that is to be set in bit 5 of the 32 bits. (Note that this code has been simplified by eliminating all movement of data to/from hardware devices; its only purpose is to illustrate bit manipulation techniques.) Figure 12 represents a C function that accomplishes this task utilizing traditional C function bit manipulation techniques; the

resulting generated assembly code is shown in Figure 13. Though the traditional C function bit manipulation takes just one line of code in C, learning how to develop this single line is not a straightforward process; it is more of an art that is acquired over time.

For this type of function, the students often find it easier to develop assembly code. Figures 14 and 15 represent the same solution written in assembly language.

Yet another approach is to use the facility in the C programming language known as bit fields. This technique is demonstrated in Figures 16 and 17. On the surface this solution appears to be more complex than traditional C bit manipulation techniques, but it is more easily reproducible and thus more usable. (It is interesting to note that all three solutions generate the same machine code.)

```
typedef unsigned int uint

uint set_bit5(uint word, uint bit)
{
  return (word & ~(1<<5)) | ((bit&0x1)<<5);
}
```

Figure 12. *set_bit5.c* (C language) source code.

```
{
 return (word & ~(1<<5)) | ((bit&0x1)<< 5);
0x00020288 <set_bit5>:    andi r6,r5,1
0x0002028c <set_bit5+4>:  slli r2,r6,5
0x00020290 <set_bit5+8>:  movi r3,-33
0x00020294 <set_bit5+12>: and  r4,r4,r3
}
0x00020298 <set_bit5+16>: or   r2,r4,r2
0x0002029c <set_bit5+20>: ret
```

Figure 13. *set_bit5.c* memory image.

```
.text

.global set_bit5
set_bit5:
      andi   r6,r5,1      # isolate passed bit
      slli   r2,r6,5      # move to ACTIVE position
      movi   r3,~(1<<5)   # movi  r5,~(0x20) ==> -33
      and    r4,r4,r3     # zero ACTIVE position
      or     r2,r4,r2     # merge new bit value

ret

.end
```

Figure 14. *set_bit5.s* (Assembly language) source code.

```
0x00020214 <set_bit5>:    andi r6,r5,1
0x00020218 <set_bit5+4>:  slli r2,r6,5
0x0002021c <set_bit5+8>:  movi r3,-33
0x00020220 <set_bit5+12>: and  r4,r4,r3
0x00020224 <set_bit5+16>: or   r2,r4,r2
0x00020228 <set_bit5+20>: ret
```

Figure 15. *set_bit5.s* memory image.

```
unsigned int set_bit5(unsigned int word, unsigned int bit)
{
      union {
            unsigned int word;
            struct {
                  unsigned int fill_1 : 5;
                  unsigned int bit5   : 1;
                  unsigned int fill_2 : 26;
            } bits;
      } data;

      data.word = word;

      data.bits.bit5 = bit;

      return data.word;
}
```

Figure 16. *set_bit5_fields.c* (C language) source code.

```
{
      union {
            unsigned int word;
            struct {
                  unsigned int fill_1 : 5;
                  unsigned int bit5   : 1;
                  unsigned int fill_2 : 26;
            } bits;
      } data;

      data.word = word;

      data.bits.bit5 = bit;
0x00020288 <set_bit5>:    andi r6,r5,1
0x0002028c <set_bit5+4>:  slli r2,r6,5
0x00020290 <set_bit5+8>:  movi r3,-33
0x00020294 <set_bit5+12>: and  r4,r4,r3

      return data.word;
}
0x00020298 <set_bit5+16>: or   r2,r4,r2
0x0002029c <set_bit5+20>: ret
```

Figure 17. *set_bit5_fields.c* memory image.

Pointers

Because pointers are difficult to learn in C, we start out by teaching the concept of pointers in assembly (register containing address of item to be accessed). By the time the students get to C, the only difference is syntax.

The classic strcpy routine presented in K&R (pg 105) to copy a string from one location to another is shown in Figure 18. By observing the optimized assembly code (Figure 19), it is clear what the function is doing at the machine level to accomplish the task. This clearly shows that registers r5 and r4 are pointers to the string source and target in their usage of the ldbu and stb assembly language instructions. Also the increment operations on both are performed after first accessing the data pointed to by those pointer variables. Sometimes students are mystified by precedence of the operators in this example. Does the increment operator increment the value pointed at or the pointer

itself? The resulting assembly code makes it perfectly clear what is happening.

```
/* strcpy:  copy t to s; pointer version 2 */
void strcpy(char *s, char *t)
{
    while ((*s++ = *t++) != '\0')
        ;
}
```

Figure 18. strcpy routine introduced in k&r.

```
{
    while ((*s++ = *t++) != '\0')
0x0002027c <strcpy>:    ldbu r2,0(r5)
0x00020280 <strcpy+4>:  addi r5,r5,1
0x00020284 <strcpy+8>:  stb  r2,0(r4)
0x00020288 <strcpy+12>: addi r4,r4,1
0x0002028c <strcpy+16>: bne  r2,zero,0x2027c
                                        <strcpy>
0x00020290 <strcpy+20>: ret
```

Figure 19. Resulting assembly language results of strcpy.

Summary

The updated core Microprocessors course at BSU is in the process of being taught for the fourth time, and continues to evolve. For example, the coverage of C programming language concepts has been abridged to target the most central microprocessor concepts. The order of presentation of topics has been revised to facilitate the transition from assembly language to C, by presenting pointers, structures, unions and bit structures at the beginning of the C language portion of the course rather than toward the end. The concept of addresses in assembly is tied to the concept of pointers in C. Supplementary examples have been prepared for both the assembly and C portions of the course to narrow the scope of and further clarify the concepts the students are expected to assimilate.

The assembly language is taught first in the course to provide a foundational understanding of processors and platforms that will accelerate the process of teaching C. Assembly language is the best way to understand and learn the foundations of microprocessors, since it is the language of the processor. The C language is added to provide a higher level view of the same processor concepts, further reinforcing the knowledge provided by learning assembly. Assembly helps to interpret what is going on at the processor level when the students are working with C, and C increases productivity for solutions to more complex problems. Rather than the students concentrating on learning the idiosyncrasies of the language specific to a particular platform, the focus of the course is on problem-solving.

The success of the course approach is gauged by student feedback, evaluation of student comprehension of concepts, and observations of student capabilities in ensuing courses. Apparent weaknesses are addressed by adjustments as the semester is progressing, and by further improvements in the next semester. Student feedback was especially helpful for refining the scope and methodology when the updated course was initially taught. During the course, the level of student comprehension of various microprocessor concepts is continually evaluated by means of homework, quizzes and exams.

The final exam is considered one measure of overall student understanding. Most of the students in Fall 2008 appeared to understand basic assembly, with 94% of the students scoring 70% or more on a final exam question requiring assembly language encoding. The questions on the final exam addressing students' comprehension of assembly/C relationships (half of the problems) had mixed results. The percentage of students scoring 70% or more on each of those questions was as follows:

Question	Concept(s)	% Scoring ≥ 70%
Write ASM function to be called by C (provided)	Passing parameters between languages How C and assembly utilize memory	62.5%
Write C code to call ASM function (provided)	Passing parameters between languages How C and assembly utilize memory	75%
Correlate assembly instructions with resulting memory image of machine instructions	Address relocation	75%
Utilize assembly instructions and memory/register information to trace execution path of code	Interaction of code and data, where data resides in registers and memory	94%
Determine memory image after executing sequence of C instructions	How C utilizes memory with relation to basic data types and structures	50%

Based on these results, adjustments have been made to the current course offering. Additional homework and examples have been developed to facilitate student learning of the key concepts, and improvements have been noted in the current semester.

Observation of student capabilities in ensuing courses has provided the most encouraging measure of success. Several students who had previously learned the C language indicated that they finally understood pointers for the first time after taking this course. With each refinement of the course, students have been able to master the concepts with fewer reiterations. We also found that students who have been through the updated course can be productive more quickly than those who haven't taken the course or who have just recently transferred into our program. The improved skills of students who have taken the updated Microprocessors course are making a difference in subsequent courses such as Embedded Systems. In the Embedded Systems course, students are able to do more complex projects earlier in the semester than was previously possible due in part to the expanded language skills from the updated Microprocessors course.

An ultra-light menu system for embedded applications that was originally assigned in Week 7 of the experimental course previously mentioned is now a beginning project in the Embedded Systems and Portable Computing course. Students who have had the updated Microprocessors course are able to develop this small efficient menu without further instruction. Students are utilizing techniques learned in the updated Microprocessors course to produce well-designed code that is easier to maintain and is also portable to other platforms.

Conclusion

A combination of assembly and C language was used to teach the basics of microprocessor programming in the updated Microprocessors course at BSU, using a modern development environment (a soft processor instantiated on an FPGA with classic RISC architecture). Overlapping the teaching of both languages had a synergistic effect on educating the students about microprocessors. In addition to learning how microprocessors work and control a broad range of devices, the students learned problem-solving skills and practiced these skills with realistic laboratory assignments and projects. Materials developed to teach the updated Microprocessors course are continuing to be expanded and refined.

References

1. B.E. Dunne, A.J. Blausch, and A. Sterian, "The Case for Computer Programming Instructions for ALL Engineering Disciplines," *Proceedings of the 2005 ASEE Annual Conference,*Portland, OR June 12-15, 2005.

2. G. Skelton, "Introducing Software Engineering to Computer Engineering Students," *Proceedings of the 2006 Southeast Conference,* 0-4244-0169-0/062006 IEEE.

3. M. Curreri, "Object-Oriented C: Creating Foundation Classes Part 1," Available: http://www.embedded.com, *Embedded Systems Design, 9/10/03*.

4. S. M. Loo, "On the Use of a Soft Processor Core in Computer Engineering Education," *Proceedings of 2006 ASEE Annual Conference*, Chicago, IL, June 18-21, 2006.

5. S.M. Loo and C.A. Planting, "Use of Discrete and Soft Processors in Introductory Microprocessors and Embedded Systems Curriculum," *Proceedings of the 2008 Workshop on Embedded Systems Education (WESE),* Atlanta, GA, October 23-24, 2008.

6. B.W. Kernighan and D.M. Ritchie, 1988. *The C Programming Language,* 2nd ed. Upper Saddle River, NJ: Prentice Hall.

Biographical Information

C. Arlen Planting is with the Electrical and Computer Engineering Department, Boise State University, Boise, ID 83725, USA. Arlen Planting received his B.S. degree in Mathematics and his M.S. degree in Electrical Engineering from Boise State University.

Sin Ming Loo is with the Electrical and Computer Engineering Department, Boise State University, Boise, ID 83725, USA. Sin Ming Loo received his B.S. degree in electrical engineering and M.S. degree in computer engineering from the University of Alabama in Huntsville, and his Ph.D. in computer engineering from University of Alabama at Birmingham and the University of Alabama in Huntsville. He joined Boise State University in 2003.