5.4 Chapter Summary

- The CPU's core timer increments once every two ticks of the SYSCLK, or every 25 ns for an 80 MHz SYSCLK. The commands `_CP0_SET_COUNT(0)` and `unsigned int dt = _CP0_GET_COUNT()` can be used to measure the execution time of the code in between to within a few SYSCLK cycles.
- To generate a disassembly listing at the command line, use `xc32-objdump -S filename.elf > filename.dis`.
- With the prefetch cache module fully enabled, your PIC32 should be able to execute an assembly instruction nearly every cycle. The prefetch allows instructions to be fetched in advance for linear code, but the prefetch cannot run past conditional statements. For small loops, the entire loop can be stored in the cache.
- The linker assigns specific program flash VAs to all program instructions and data RAM VAs to all global variables. The rest of RAM is allocated to the heap, for dynamic memory allocation, and to the stack, for function parameters and temporary local variables. The heap is zero bytes by default.
- A map file provides a detailed summary of memory usage. To generate a map file at the command line, use the `-Map` option to the linker, e.g.,

```
xc32-gcc [details omitted] -Wl,-Map="out.map"
```

- Global variables can be initialized (assigned a value when they are defined) or uninitialized. Initialized global variables are stored in RAM memory sections `.data` and `.sdata` and uninitialized globals are stored in RAM memory sections `.bss` and `.sbss`. Sections beginning with `.s` mean that the variables are "small." When the program is executed, initialized global variables are assigned their values by C runtime startup code, and uninitialized global variables are set to zero.
- Global variables are packed tightly at the beginning of data RAM, 0xA0000000. The heap comes immediately after. The stack begins at the high end of RAM and grows "down" toward lower RAM addresses. Stack overflow occurs if the stack pointer attempts to move into an area reserved for the heap or global variables.

5.5 Exercises

Unless otherwise specified, compile with no optimizations for all problems.

1. Describe two examples of how you can write code differently to either make it execute faster or use less program memory.
2. Compile and run `timing.c`, Code Sample 5.2, with no optimizations (`make CFLAGS="-g -x c"`). With a stopwatch, verify the time taken by the delay loop. Do your results agree with Section 5.2.3?

3. To write time-efficient code, it is important to understand that some mathematical operations are faster than others. We will look at the disassembly of code that performs simple arithmetic operations on different data types. Create a program with the following local variables in `main`:

```
char c1=5, c2=6, c3;
int i1=5, i2=6, i3;
long long int j1=5, j2=6, j3;
float f1=1.01, f2=2.02, f3;
long double d1=1.01, d2=2.02, d3;
```

   Now write code that performs add, subtract, multiply, and divide for each of the five data types, i.e., for `char`s:

```
c3 = c1+c2;
c3 = c1-c2;
c3 = c1*c2;
c3 = c1/c2;
```

   Build the program with no optimization and look at the disassembly. For each of the statements, you will notice that some of the assembly code involves simply loading the variables from RAM into CPU registers and storing the result (also in a register) back to RAM. Also, while some of the statements are completed by a few assembly commands in sequence, others result in a jump to a software subroutine to complete the calculation. (These subroutines are provided with our C installation and included in the linking process.) Answer the following questions.

   a. Which combinations of data types and arithmetic functions result in a jump to a subroutine? From your disassembly file, copy the C statement and the assembly commands it expands to (including the jump) for one example.
   b. For those statements that do not result in a jump to a subroutine, which combination(s) of data types and arithmetic functions result in the fewest assembly commands? From your disassembly, copy the C statement and its assembly commands for one of these examples. Is the smallest data type, `char`, involved in it? If not, what is the purpose of extra assembly command(s) for the `char` data type vs. the `int` data type? (Hint: the assembly command `ANDI` takes the bitwise AND of the second argument with the third argument, a constant, and stores the result in the first argument. Or you may wish to look up a MIPS32 assembly instruction reference.)
   c. Fill in the following table. Each cell should have two numbers: the number of assembly commands for the specified operation and data type, and the ratio of this number (greater than or equal to 1.0) to the smallest number of assembly commands in the table. For example, if addition of two `int`s takes four assembly commands, and this is the fewest in the table, then the entry in that cell would be 1.0 (4). This has been filled in below, but you should change it if you get a different result. If a statement results in a jump to a subroutine, write J in that cell.

	char	int	long long	float	long double
+		1.0 (4)			
−					
*					
/					

d. From the disassembly, find out the name of any math subroutine that has been added to your assembly code. Now create a map file of the program. Where are the math subroutines installed in virtual memory? Approximately how much program memory is used by each of the subroutines? You can use evidence from the disassembly file and/or the map file. (Hint: You can search backward from the end of your map file for the name of any math subroutines.)

4. Let us look at the assembly code for bit manipulation. Create a program with the following local variables:

```
unsigned int u1=33, u2=17, u3;
```

and look at the assembly commands for the following statements:

```
u3 = u1 & u2;     // bitwise AND
u3 = u1 | u2;     // bitwise OR
u3 = u2 << 4;     // shift left 4 spaces, or multiply by 2^4 = 16
u3 = u1 >> 3;     // shift right 3 spaces, or divide by 2^3 = 8
```

How many commands does each use? For unsigned integers, bit-shifting left and right make for computationally efficient multiplies and divides, respectively, by powers of 2.

5. Use the core timer to calculate a table similar to that in Exercise 3, except with entries corresponding to the actual execution time in terms of SYSCLK cycles. So if a calculation takes 15 cycles, and the fastest calculation is 10 cycles, the entry would be 1.5 (15). This table should contain all 20 entries, even for those that jump to subroutines. (Note: subroutines often have conditional statements, meaning that the calculation could terminate faster for some operands than for others. You can report the results for the variable values given in Exercise 3.)
To minimize uncertainty due to the setup and reading time of the core timer, and the fact that the timer only increments once every two SYSCLK cycles, each math statement could be repeated ten or more times (no loops) between setting the timer to zero and reading the timer. The average number of cycles, rounded down, should be the number of cycles for each statement. Use the NU32 communication routines, or any other communication routines, to report the answers back to your computer.

6. Certain math library functions can take quite a bit longer to execute than simple arithmetic functions. Examples include trigonometric functions, logarithms, square roots, etc. Make a program with the following local variables:

```
float f1=2.07, f2;         // four bytes for each float
long double d1=2.07, d2;   // eight bytes for each long double
```

Also be sure to put `#include <math.h>` at the top of your program to make the math function prototypes available.

a. Using methods similar to those in Exercise 5, measure how long it takes to perform each of `f2 = cosf(f1)`, `f2 = sqrtf(f1)`, `d2 = cos(d1)`, and `d2 = sqrt(d1)`.
b. Copy and paste the disassembly from a `f2 = cosf(f1)` statement and a `d2 = cos(d1)` statement into your solution set and compare them. Based on the comparison of the assembly codes, comment on the advantages and disadvantages of using the eight-byte `long double` floating point representation compared to the four-byte `float` representation when you compute a cosine with the PIC32 compiler.
c. Make a map file for this program, and search for the references to the math library `libm.a` in the map file. There are several `libm.a` files in your C installation, but which one was used by the linker when you built your program? Give the directory.

7. Explain what stack overflow is, and give a short code snippet (not a full program) that would result in stack overflow on your PIC32.
8. In the map file of the original `timing.c` program, there are several `App's exec code`, one corresponding to `timing.o`. Explain briefly what each of the others are for. Provide evidence for your answer from the map file.
9. Create a map file for `simplePIC.c` from Chapter 3. (a) How many bytes does `simplePIC.o` use? (b) Where are the functions `main` and `delay` placed in virtual memory? Are instructions at these locations cacheable? (c) Search the map file for the `.reset` section. Where is it in virtual memory? Is it consistent with your `NU32bootloaded.ld` linker file? (d) Now augment the program by defining `short int`, `long int`, `long long int`, `float`, `double`, and `long double` global variables. Provide evidence from the map file indicating how much memory each data type uses.
10. Assume your program defines a global `int` array `int glob[5000]`. Now what is the maximum size of an array of `int`s that you can define as a local variable?
11. Provide global variable definitions (not an entire program) so that the map file has data sections `.sdata` of 16 bytes, `.sbss` of 24 bytes, `.data` of 0 bytes, and `.bss` of 200 bytes.
12. If you define a global variable and you want to set its initial value, is it "better" to initialize it when the variable is defined or to initialize it in a function? Explain any pros and cons.
13. The program `readVA.c` (Code Sample 5.2) prints out the contents of the 4-byte word at any virtual address, provided the address is word-aligned (i.e., evenly divisible by four). This allows you to inspect anything in the virtual memory map (Chapter 3), including the representations of variables in data RAM, program instructions in flash or boot flash, SFRs, and configuration bits. You can use code like this in conjunction with the map and disassembly files to better understand the code produced by a build. Here's a sample of the program's output to the terminal:

```
Enter the start VA (e.g., bd001970) and # of 32-bit words to print.
Listing 4 32-bit words starting from address bd001970.
 ADDRESS  CONTENTS
bd001970  0f40065e
bd001974  00000000
bd001978  401a6000
bd00197c  7f5a04c0
```

a. Build and run the program. Check its operation by consulting your disassembly file and confirming that 32-bit program instructions (in flash) listed there match the output of the program. Confirm that you get the same results whether you reference the same physical memory address using a cacheable or noncacheable virtual address. What happens if you specify a virtual address that is not divisible by four?
b. Examine the map file. At what virtual address in RAM is the variable `val` stored? Confirm its value using the program.
c. The values of the configuration bits (the four words DEVCFG0 to DEVCFG3, see Chapter 2.1.4) were set by the preinstalled bootloader. Use the program to print the values of these device configuration registers.
d. Consulting the map or disassembly file, what is the address of the last instruction in the program? Use the program to provide a listing of the instructions from a few addresses before to a few addresses after the last instruction. Addresses that do not have an instruction were erased when the program was loaded by the bootloader, but no instructions were written there. Knowing that, and by looking at your program's output, what value does an erased flash byte have?
e. Modify the program so the `unsigned int`s are defined as local to `main`, so that they are on the stack. Since `val` is no longer given a specific address by the linker, you do not find it in the map file. Use your program to find the address in RAM where `val` is stored.

Code Sample 5.2 `readVA.c`**. Code to Inspect the Virtual Memory Map.**

```
#include "NU32.h"
#define MSGLEN 100

// val is an initialized global; you can find it in the memory map with this program
char msg[MSGLEN];
unsigned int *addr;
unsigned int k = 0, nwords = 0, val = 0xf01dable;

int main(void) {

  NU32_Startup();
  while (1) {
    sprintf(msg, "Enter the start VA (e.g., bd001970) and # of 32-bit words to print: ");
    NU32_WriteUART3(msg);

    NU32_ReadUART3(msg,MSGLEN);
    sscanf(msg,"%x %d",&addr, &nwords);
    sprintf(msg,"\r\nListing %d 32-bit words starting from VA %08x.\r\n",nwords,addr);
    NU32_WriteUART3(msg);

    sprintf(msg," ADDRESS  CONTENTS\r\n");
    NU32_WriteUART3(msg);

    for (k = 0; k < nwords; k++) {
      sprintf(msg,"%08x  %08x\r\n", addr,*addr); // *addr is the 32 bits starting at addr
      NU32_WriteUART3(msg);
      ++addr;                  // addr is an unsigned int ptr so it increments by 4 bytes
    }
  }
  return 0;
}

// handle cpu exceptions, such as trying to read from a bad memory location
void _general_exception_handler(unsigned cause, unsigned status)
{
  unsigned int exccode = (cause & 0x3C) >> 2; // the exccode is reason for the exception
  // note: see PIC32 Family Reference Manual Section 03 CPU M4K Core for details
  // Look for the Cause register and the Status Register
  NU32_WriteUART3("Reset the PIC32 due to general exception.\r\n");
  sprintf(msg,"cause 0x%08x (EXCCODE = 0x%02x), status 0x%08x\r\n",cause,exccode,status);
  NU32_WriteUART3(msg);
  while(1) {
    ;
  }
}
```

Further Reading

MIPS32 M4K processor core software user's manual (2.03 ed.). (2008). MIPS Technologies.
PIC32 family reference manual. Section 04: Prefectch cache module. (2011). Microchip Technology Inc.