include accumulator architectures, general-purpose register architectures, stack architectures, and a brief history of ARM and the x86. We also review the controversial subjects of high-level-language computer architectures and reduced instruction set computer architectures. The history of programming languages includes Fortran, Lisp, Algol, C, Cobol, Pascal, Simula, Smalltalk, C++, and Java, and the history of compilers includes the key milestones and the pioneers who achieved them. The rest of Section 2.21 is found online.

2.22 Exercises

Appendix A describes the MIPS simulator, which is helpful for these exercises. Although the simulator accepts pseudoinstructions, try not to use pseudoinstructions for any exercises that ask you to produce MIPS code. Your goal should be to learn the real MIPS instruction set, and if you are asked to count instructions, your count should reflect the actual instructions that will be executed and not the pseudoinstructions.

There are some cases where pseudoinstructions must be used (for example, the `la` instruction when an actual value is not known at assembly time). In many cases, they are quite convenient and result in more readable code (for example, the `li` and `move` instructions). If you choose to use pseudoinstructions for these reasons, please add a sentence or two to your solution stating which pseudoinstructions you have used and why.

2.1 [5] <§2.2> For the following C statement, what is the corresponding MIPS assembly code? Assume that the variables `f`, `g`, `h`, and `i` are given and could be considered 32-bit integers as declared in a C program. Use a minimal number of MIPS assembly instructions.

```
f = g + (h − 5);
```

2.2 [5] <§2.2> For the following MIPS assembly instructions above, what is a corresponding C statement?

```
add  f, g, h
add  f, i, f
```

2.3 [5] <§§2.2, 2.3> For the following C statement, what is the corresponding MIPS assembly code? Assume that the variables f, g, h, i, and j are assigned to registers $s0, $s1, $s2, $s3, and $s4, respectively. Assume that the base address of the arrays A and B are in registers $s6 and $s7, respectively.

```
B[8] = A[i-j];
```

2.4 [5] <§§2.2, 2.3> For the MIPS assembly instructions below, what is the corresponding C statement? Assume that the variables f, g, h, i, and j are assigned to registers $s0, $s1, $s2, $s3, and $s4, respectively. Assume that the base address of the arrays A and B are in registers $s6 and $s7, respectively.

```
sll  $t0, $s0, 2     # $t0 = f * 4
add  $t0, $s6, $t0   # $t0 = &A[f]
sll  $t1, $s1, 2     # $t1 = g * 4
add  $t1, $s7, $t1   # $t1 = &B[g]
lw   $s0, 0($t0)     # f = A[f]
addi $t2, $t0, 4
lw   $t0, 0($t2)
add  $t0, $t0, $s0
sw   $t0, 0($t1)
```

2.5 [5] <§§2.2, 2.3> For the MIPS assembly instructions in Exercise 2.4, rewrite the assembly code to minimize the number if MIPS instructions (if possible) needed to carry out the same function.

2.6 The table below shows 32-bit values of an array stored in memory.

Address	Data
24	2
~~38~~ 2	4
32	3
36	6
40	1

2.6.1 [5] <§§2.2, 2.3> For the memory locations in the table above, write C code to sort the data from lowest to highest, placing the lowest value in the smallest memory location shown in the figure. Assume that the data shown represents the C variable called `Array`, which is an array of type `int`, and that the first number in the array shown is the first element in the array. Assume that this particular machine is a byte-addressable machine and a word consists of four bytes.

2.6.2 [5] <§§2.2, 2.3> For the memory locations in the table above, write MIPS code to sort the data from lowest to highest, placing the lowest value in the smallest memory location. Use a minimum number of MIPS instructions. Assume the base address of `Array` is stored in register `$s6`.

2.7 [5] <§2.3> Show how the value `0xabcdef12` would be arranged in memory of a little-endian and a big-endian machine. Assume the data is stored starting at address 0.

2.8 [5] <§2.4> Translate `0xabcdef12` into decimal.

2.9 [5] <§§2.2, 2.3> Translate the following C code to MIPS. Assume that the variables `f`, `g`, `h`, `i`, and `j` are assigned to registers `$s0`, `$s1`, `$s2`, `$s3`, and `$s4`, respectively. Assume that the base address of the arrays `A` and `B` are in registers `$s6` and `$s7`, respectively. Assume that the elements of the arrays `A` and `B` are 4-byte words:

```
B[8] = A[i] + A[j];
```

2.10 [5] <§§2.2, 2.3> Translate the following MIPS code to C. Assume that the variables `f`, `g`, `h`, `i`, and `j` are assigned to registers `$s0`, `$s1`, `$s2`, `$s3`, and `$s4`, respectively. Assume that the base address of the arrays `A` and `B` are in registers `$s6` and `$s7`, respectively.

```
addi $t0, $s6, 4
add  $t1, $s6, $0
sw   $t1, 0($t0)
lw   $t0, 0($t0)
add  $s0, $t1, $t0
```

2.11 [5] <§§2.2, 2.5> For each MIPS instruction, show the value of the opcode (OP), source register (RS), and target register (RT) fields. For the I-type instructions, show the value of the immediate field, and for the R-type instructions, show the value of the destination register (RD) field.

2.12 Assume that registers `$s0` and `$s1` hold the values `0x80000000` and `0xD0000000`, respectively.

2.12.1 [5] <§2.4> What is the value of `$t0` for the following assembly code?

```
add $t0, $s0, $s1
```

2.12.2 [5] <§2.4> Is the result in `$t0` the desired result, or has there been overflow?

2.12.3 [5] <§2.4> For the contents of registers `$s0` and `$s1` as specified above, what is the value of `$t0` for the following assembly code?

```
sub $t0, $s0, $s1
```

2.12.4 [5] <§2.4> Is the result in `$t0` the desired result, or has there been overflow?

2.12.5 [5] <§2.4> For the contents of registers `$s0` and `$s1` as specified above, what is the value of `$t0` for the following assembly code?

```
add $t0, $s0, $s1
add $t0, $t0, $s0
```

2.12.6 [5] <§2.4> Is the result in `$t0` the desired result, or has there been overflow?

2.13 Assume that `$s0` holds the value 128_{ten}.

2.13.1 [5] <§2.4> For the instruction `add $t0`, `$s0`, `$s1`, what is the range(s) of values for `$s1` that would result in overflow?

2.13.2 [5] <§2.4> For the instruction `sub $t0`, `$s0`, `$s1`, what is the range(s) of values for `$s1` that would result in overflow?

2.13.3 [5] <§2.4> For the instruction `sub $t0`, `$s1`, `$s0`, what is the range(s) of values for `$s1` that would result in overflow?

2.14 [5] <§§2.2, 2.5> Provide the type and assembly language instruction for the following binary value: `0000 0010 0001 0000 1000 0000 0010 0000`$_{two}$

2.15 [5] <§§2.2, 2.5> Provide the type and hexadecimal representation of following instruction: `sw $t1, 32($t2)`

2.16 [5] <§2.5> Provide the type, assembly language instruction, and binary representation of instruction described by the following MIPS fields:

```
op=0, rs=3, rt=2, rd=3, shamt=0, funct=34
```

2.17 [5] <§2.5> Provide the type, assembly language instruction, and binary representation of instruction described by the following MIPS fields:

```
op=0x23, rs=1, rt=2, const=0x4
```

2.18 Assume that we would like to expand the MIPS register file to 128 registers and expand the instruction set to contain four times as many instructions.

2.18.1 [5] <§2.5> How this would this affect the size of each of the bit fields in the R-type instructions?

2.18.2 [5] <§2.5> How this would this affect the size of each of the bit fields in the I-type instructions?

2.18.3 [5] <§§2.5, 2.10> How could each of the two proposed changes decrease the size of an MIPS assembly program? On the other hand, how could the proposed change increase the size of an MIPS assembly program?

2.19 Assume the following register contents:

```
$t0 = 0xAAAAAAAA, $t1 = 0x12345678
```

2.19.1 [5] <§2.6> For the register values shown above, what is the value of `$t2` for the following sequence of instructions?

```
sll $t2, $t0, 44
or  $t2, $t2, $t1
```

2.19.2 [5] <§2.6> For the register values shown above, what is the value of `$t2` for the following sequence of instructions?

```
sll  $t2, $t0, 4
andi $t2, $t2, −1
```

2.19.3 [5] <§2.6> For the register values shown above, what is the value of `$t2` for the following sequence of instructions?

```
srl  $t2, $t0, 3
andi $t2, $t2, 0xFFEF
```

2.20 [5] <§2.6> Find the shortest sequence of MIPS instructions that extracts bits 16 down to 11 from register `$t0` and uses the value of this field to replace bits 31 down to 26 in register `$t1` without changing the other 26 bits of register `$t1`.

2.21 [5] <§2.6> Provide a minimal set of MIPS instructions that may be used to implement the following pseudoinstruction:

```
not $t1, $t2      // bit-wise invert
```

2.22 [5] <§2.6> For the following C statement, write a minimal sequence of MIPS assembly instructions that does the identical operation. Assume `$t1 = A`, `$t2 = B`, and `$s1` is the base address of C.

```
A = C[0] << 4;
```

2.23 [5] <§2.7> Assume `$t0` holds the value `0x00101000`. What is the value of `$t2` after the following instructions?

```
         slt  $t2, $0,  $t0
         bne  $t2, $0,  ELSE
         j    DONE
ELSE:    addi $t2, $t2, 2
DONE:
```

2.24 [5] <§2.7> Suppose the program counter (PC) is set to `0x2000 0000`. Is it possible to use the jump (j) MIPS assembly instruction to set the PC to the address as `0x4000 0000`? Is it possible to use the branch-on-equal (beq) MIPS assembly instruction to set the PC to this same address?

2.25 The following instruction is not included in the MIPS instruction set:

```
rpt $t2, loop # if(R[rs]>0) R[rs]=R[rs]−1, PC=PC+4+BranchAddr
```

2.25.1 [5] <§2.7> If this instruction were to be implemented in the MIPS instruction set, what is the most appropriate instruction format?

2.25.2 [5] <§2.7> What is the shortest sequence of MIPS instructions that performs the same operation?

2.26 Consider the following MIPS loop:

```
LOOP: slt  $t2, $0,  $t1
      beq  $t2, $0,  DONE
      subi $t1, $t1, 1
      addi $s2, $s2, 2
      j    LOOP
DONE:
```

2.26.1 [5] <§2.7> Assume that the register `$t1` is initialized to the value 10. What is the value in register `$s2` assuming `$s2` is initially zero?

2.26.2 [5] <§2.7> For each of the loops above, write the equivalent C code routine. Assume that the registers `$s1`, `$s2`, `$t1`, and `$t2` are integers `A`, `B`, `i`, and temp, respectively.

2.26.3 [5] <§2.7> For the loops written in MIPS assembly above, assume that the register `$t1` is initialized to the value N. How many MIPS instructions are executed?

2.27 [5] <§2.7> Translate the following C code to MIPS assembly code. Use a minimum number of instructions. Assume that the values of `a`, `b`, `i`, and `j` are in registers `$s0`, `$s1`, `$t0`, and `$t1`, respectively. Also, assume that register `$s2` holds the base address of the array D.

```
for(i=0; i<a; i++)
   for(j=0; j<b; j++)
      D[4*j] = i + j;
```

2.28 [5] <§2.7> How many MIPS instructions does it take to implement the C code from Exercise 2.27? If the variables a and b are initialized to 10 and 1 and all elements of D are initially 0, what is the total number of MIPS instructions that is executed to complete the loop?

2.29 [5] <§2.7> Translate the following loop into C. Assume that the C-level integer `i` is held in register `$t1`, `$s2` holds the C-level integer called `result`, and `$s0` holds the base address of the integer `MemArray`.

```
      addi $t1, $0, $0
LOOP: lw   $s1, 0($s0)
      add  $s2, $s2, $s1
      addi $s0, $s0, 4
```

```
        addi $t1, $t1, 1
        slti $t2, $t1, 100
        bne  $t2, $s0, LOOP
```

2.30 [5] <§2.7> Rewrite the loop from Exercise 2.29 to reduce the number of MIPS instructions executed.

2.31 [5] <§2.8> Implement the following C code in MIPS assembly. What is the total number of MIPS instructions needed to execute the function?

```
int fib(int n){
    if (n==0)
       return 0;
    else if (n == 1)
       return 1;
    else
       return fib(n-1) + fib(n-2);
```

2.32 [5] <§2.8> Functions can often be implemented by compilers "in-line." An in-line function is when the body of the function is copied into the program space, allowing the overhead of the function call to be eliminated. Implement an "in-line" version of the C code above in MIPS assembly. What is the reduction in the total number of MIPS assembly instructions needed to complete the function? Assume that the C variable n is initialized to 5.

2.33 [5] <§2.8> For each function call, show the contents of the stack after the function call is made. Assume the stack pointer is originally at address 0x7ffffffc, and follow the register conventions as specified in Figure 2.11.

2.34 Translate function `f` into MIPS assembly language. If you need to use registers `$t0` through `$t7`, use the lower-numbered registers first. Assume the function declaration for `func` is "`int f(int a, int b);`". The code for function `f` is as follows:

```
int f(int a, int b, int c, int d){
  return func(func(a,b),c+d);
}
```

2.35 [5] <§2.8> Can we use the tail-call optimization in this function? If no, explain why not. If yes, what is the difference in the number of executed instructions in f with and without the optimization?

2.36 [5] <§2.8> Right before your function `f` from Exercise 2.34 returns, what do we know about contents of registers `$t5`, `$s3`, `$ra`, and `$sp`? Keep in mind that we know what the entire function `f` looks like, but for function `func` we only know its declaration.

2.37 [5] <§2.9> Write a program in MIPS assembly language to convert an ASCII number string containing positive and negative integer decimal strings, to an integer. Your program should expect register `$a0` to hold the address of a null-terminated string containing some combination of the digits 0 through 9. Your program should compute the integer value equivalent to this string of digits, then place the number in register `$v0`. If a non-digit character appears anywhere in the string, your program should stop with the value −1 in register `$v0`. For example, if register `$a0` points to a sequence of three bytes 50ten, 52ten, 0ten (the null-terminated string "24"), then when the program stops, register `$v0` should contain the value 24_{ten}.

2.38 [5] <§2.9> Consider the following code:

```
lbu $t0, 0($t1)
sw  $t0, 0($t2)
```

Assume that the register `$t1` contains the address `0x1000 0000` and the register `$t2` contains the address `0x1000  0010`. Note the MIPS architecture utilizes big-endian addressing. Assume that the data (in hexadecimal) at address `0x1000 0000` is: `0x11223344`. What value is stored at the address pointed to by register `$t2`?

2.39 [5] <§2.10> Write the MIPS assembly code that creates the 32-bit constant `0010 0000 0000 0001 0100 1001 0010 0100`$_{two}$ and stores that value to register `$t1`.

2.40 [5] <§§2.6, 2.10> If the current value of the PC is `0x00000000`, can you use a single jump instruction to get to the PC address as shown in Exercise 2.39?

2.41 [5] <§§2.6, 2.10> If the current value of the PC is `0x00000600`, can you use a single branch instruction to get to the PC address as shown in Exercise 2.39?

2.42 [5] <§§2.6, 2.10> If the current value of the PC is `0x1FFFf000`, can you use a single branch instruction to get to the PC address as shown in Exercise 2.39?

2.43 [5] <§2.11> Write the MIPS assembly code to implement the following C code:

```
lock(lk);
shvar=max(shvar,x);
unlock(lk);
```

Assume that the address of the `lk` variable is in `$a0`, the address of the shvar variable is in `$a1`, and the value of variable `x` is in `$a2`. Your critical section should not contain any function calls. Use `ll/sc` instructions to implement the `lock()` operation, and the `unlock()` operation is simply an ordinary store instruction.

2.44 [5] <§2.11> Repeat Exercise 2.43, but this time use `ll/sc` to perform an atomic update of the `shvar` variable directly, without using `lock()` and `unlock()`. Note that in this problem there is no variable `lk`.

2.45 [5] <§2.11> Using your code from Exercise 2.43 as an example, explain what happens when two processors begin to execute this critical section at the same time, assuming that each processor executes exactly one instruction per cycle.

2.46 Assume for a given processor the CPI of arithmetic instructions is 1, the CPI of load/store instructions is 10, and the CPI of branch instructions is 3. Assume a program has the following instruction breakdowns: 500 million arithmetic instructions, 300 million load/store instructions, 100 million branch instructions.

2.46.1 [5] <§2.19> Suppose that new, more powerful arithmetic instructions are added to the instruction set. On average, through the use of these more powerful arithmetic instructions, we can reduce the number of arithmetic instructions needed to execute a program by 25%, and the cost of increasing the clock cycle time by only 10%. Is this a good design choice? Why?

2.46.2 [5] <§2.19> Suppose that we find a way to double the performance of arithmetic instructions. What is the overall speedup of our machine? What if we find a way to improve the performance of arithmetic instructions by 10 times?

2.47 Assume that for a given program 70% of the executed instructions are arithmetic, 10% are load/store, and 20% are branch.

2.47.1 [5] <§2.19> Given this instruction mix and the assumption that an arithmetic instruction requires 2 cycles, a load/store instruction takes 6 cycles, and a branch instruction takes 3 cycles, find the average CPI.

2.47.2 [5] <§2.19> For a 25% improvement in performance, how many cycles, on average, may an arithmetic instruction take if load/store and branch instructions are not improved at all?

2.47.3 [5] <§2.19> For a 50% improvement in performance, how many cycles, on average, may an arithmetic instruction take if load/store and branch instructions are not improved at all?

Answers to Check Yourself

§2.2, page 66: MIPS, C, Java
§2.3, page 72: 2) Very slow
§2.4, page 79: 2) -8_{ten}
§2.5, page 87: 4) sub `$t2, $t0, $t1`
§2.6, page 89: Both. AND with a mask pattern of 1s will leaves 0s everywhere but the desired field. Shifting left by the correct amount removes the bits from the left of the field. Shifting right by the appropriate amount puts the field into the rightmost bits of the word, with 0s in the rest of the word. Note that AND leaves the field where it was originally, and the shift pair moves the field into the rightmost part of the word.
§2.7, page 96: I. All are true. II. 1).
§2.8, page 106: Both are true.
§2.9, page 111: I. 1) and 2) II. 3)
§2.10, page 120: I. 4) +−128K. II. 6) a block of 256M. III. 4) `sll`
§2.11, page 123: Both are true.
§2.12, page 132: 4) Machine independence.