a.

```
int compare(int a, int b) {
    if (sub(a, b) >= 0)
        return 1;
    else
        return 0;
}
int sub (int a, int b) {
    return a-b;
}
```

b.

```
int fib_iter(int a, int b, int n){
    if(n == 0)
        return b;
    else
        return fib_iter(a+b, a, n-1);
}
```

2.19.1 [15] <2.8> Implement the C code in the table in MIPS assembly. What is the total number of MIPS instructions needed to execute the function?

2.19.2 [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 in the table 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.

2.19.3 [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 addresss 0x7fffffc, and follow the register conventions as specified in Figure 2.11.

The following three problems in this exercise refer to a function `f` that calls another function `func`. The code for C function `func` is already compiled in another module using the MIPS calling convention from Figure 2.14. The function declaration for `func` is "`int func(int a, int b);`". The code for function `f` is as follows:

a.

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

b.

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

2.19.4 [10] <2.8> Translate function `f` into MIPS assembler, also using the MIPS calling convention from Figure 2.14. If you need to use registers `$t0` through `$t7`, use the lower-numbered registers first.

2.19.5 [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?

2.19.6 [5] <2.8> Right before your function `f` from Problem 2.19.4 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.

Exercise 2.20

This exercise deals with recursive procedure calls. For the following problems, the table has an assembly code fragment that computes the factorial of a number. However, the entries in the table have errors, and you will be asked to fix these errors.

a.

```
FACT:  addi  $sp, $sp, -8
       sw    $ra, 4($sp)
       sw    $a0, 0($sp)
       slti  $t0, $a0, 1
       beq   $t0, $0, L1
       addi  $v0, $0, 1
       addi  $sp, $sp, 8
       jr    $ra

L1:    addi  $a0, $a0, -1
       jal   FACT
       lw    $a0, 4($sp)
       lw    $ra, 0($sp)
       addi  $sp, $sp, 8
       mul   $v0, $a0, $v0
       jr    $ra
```

b.

```
FACT:  addi  $sp, $sp, -8
       sw    $ra, 4($sp)
       sw    $a0, 0($sp)
       slti  $t0, $a0, 1
       beq   $t0, $0, L1
       addi  $v0, $0, 1
       addi  $sp, $sp, 8
       jr    $ra

L1:    addi  $t0, $t0, -1
       jal   FACT
       lw    $a0, 4($sp)
       lw    $ra, 0($sp)
       addi  $sp, $sp, 8
       mul   $v0, $a0, $v0
       jr    $ra
```

2.20.1 [5] <2.8> The MIPS assembly program above computes the factorial of a given input. The integer input is passed through register `$a0`, and the result is returned in register `$v0`. In the assembly code, there are a few errors. Correct the MIPS errors.

2.20.2 [10] <2.8> For the recursive factorial MIPS program above, assume that the input is 4. Rewrite the factorial program to operate in a nonrecursive manner.

Restrict your register usage to registers $s0-$s7. What is the total number of instructions used to execute your solution from 2.20.2 versus the recursive version of the factorial program?

2.20.3 [5] <2.8> Show the contents of the stack after each function call, assuming that the input is 4.

For the following problems, the table has an assembly code fragment that computes a Fibonacci number. However, the entries in the table have errors, and you will be asked to fix these errors.

a.

```
FIB:  addi  $sp,$sp, -12
      sw    $ra, 0($sp)
      sw    $s1, 4($sp)
      sw    $a0, 8($sp)
      slti  $t0, $a0, 1
      beq   $t0, $0, L1
      addi  $v0,$a0, $0
      j     EXIT

L1:   addi  $a0,$a0, -1
      jal   FIB
      addi  $s1,$v0, $0
      addi  $a0,$a0, -1
      jal   FIB
      add   $v0, $v0, $s1

EXIT: lw    $ra, 0($sp)
      lw    $a0, 8($sp)
      lw    $s1, 4($sp)
      addi  $sp, $sp, 12
      jr    $ra
```

b.

```
FIB:  addi  $sp,$sp, -12
      sw    $ra, 0($sp)
      sw    $s1, 4($sp)
      sw    $a0, 8($sp)
      slti  $t0, $a0, 1
      beq   $t0, $0, L1
      addi  $v0,$a0, $0
      j     EXIT

L1:   addi  $a0,$a0, -1
      jal   FIB
      addi  $s1,$v0, $0
      addi  $a0,$a0, -1
      jal   FIB
      add   $v0, $v0, $s1

EXIT: lw    $ra, 0($sp)
      lw    $a0, 8($sp)
      lw    $s1, 4($sp)
      addi  $sp, $sp, 12
      jr    $ra
```

2.20.4 [5] <2.8> The MIPS assembly program above computes the Fibonacci of a given input. The integer input is passed through register `$a0`, and the result is returned in register `$v0`. In the assembly code, there are a few errors. Correct the MIPS errors.

2.20.5 [10] <2.8> For the recursive Fibonacci MIPS program above, assume that the input is 4. Rewrite the Fibonacci program to operate in a nonrecursive manner. Restrict your register usage to registers `$s0-$s7`. What is the total number of instructions used to execute your solution from 2.20.2 versus the recursive version of the factorial program?

2.20.6 [5] <2.8> Show the contents of the stack after each function call, assuming that the input is 4.

Exercise 2.21

Assume that the stack and the static data segments are empty and that the stack and global pointers start at address 0x7fff fffc and 0x1000 8000, respectively. Assume the calling conventions as specified in Figure 2.11 and that function inputs are passed using registers `$a0` and returned in register `$v0`. Assume that leaf functions may only use saved registers.

a.

```
main()
{
    leaf_function(1);
}
int leaf_function (int f)
{
    int result;
    result = f + 1;
    if (f > 5)
        return result;
    leaf_function(result);
}
```

b.

```
int my_global = 100;
main()
{
    int x = 10;
    int y = 20;
    int z;
    z = my_function(x, my_global)
}
int my_function(int x, int y)
{
    return x - y;
}
```

2.21.1 [5] <2.8> Show the contents of the stack and the static data segments after each function call.

2.21.2 [5] <2.8> Write MIPS code for the code in the table above.

2.21.3 [5] <2.8> If the leaf function could use temporary registers (`$t0`, `$t1`, etc.), write the MIPS code for the code in the table above.

The following three problems in this exercise refer to this function, written in MIPS assembler following the calling conventions from Figure 2.14:

a.
```
f: sub   $s0,$a0,$a3
   sll   $v0,$s0,0x1
   add   $v0,$a2,$v0
   sub   $v0,$v0,$a1
   jr    $ra
```

b.
```
f: addi  $sp,$sp,8
   sw    $ra,4($sp)
   sw    $s0,0($sp)
   move  $s0,$a2
   jal   g
   add   $v0,$v0,$s0
   lw    $ra,4($sp)
   lw    $s0,0($sp)
   addi  $sp,$sp,-8
   jr    $ra
```

2.21.4 [10] <2.8> This code contains a mistake that violates the MIPS calling convention. What is this mistake and how should it be fixed?

2.21.5 [10] <2.8> What is the C equivalent of this code? Assume that the function's arguments are named `a`, `b`, `c`, etc. in the C version of the function.

2.21.6 [10] <2.8> At the point where this function is called register `$a0`, `$a1`, `$a2`, and `$a3` have values 1, 100, 1000, and 30, respectively. What is the value returned by this function? If another function `g` is called from `f`, assume that the value returned from `g` is always 500.

Exercise 2.22

This exercise explores ASCII and Unicode conversion. The following table shows strings of characters.

a.	A byte
b.	computer

2.22.1 [5] <2.9> Translate the strings into decimal ASCII byte values.

2.22.2 [5] <2.9> Translate the strings into 16-bit Unicode (using hex notation and the Basic Latin character set).

The following table shows hexadecimal ASCII character values.

a.	61 64 64
b.	73 68 69 66 74

2.22.3 [5] <2.5, 2.9> Translate the hexadecimal ASCII values to text.

Exercise 2.23

In this exercise, you will be asked to write a MIPS assembly program that converts strings into the number format as specified in the table.

a.	positive integer decimal strings
b.	two's complement hexadecimal integers

2.23.1 [10] <2.9> Write a program in MIPS assembly language to convert an ASCII number string with the conditions listed in the table above, 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 nondigit 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 50_{ten}, 52_{ten}, 0_{ten} (the null-terminated string "24"), then when the program stops, register `$v0` should contain the value 24_{ten}.

Exercise 2.24

Assume that the register `$t1` contains the address 0x1000 0000 and the register `$t2` contains the address 0x1000 0010.

a.	`lb $t0, 0($t1)` `sw $t0, 0($t2)`
b.	`lb $t0, 0($t1)` `sb $t0, 0($t2)`

2.24.1 [5] <2.9> Assume that the data (in hexadecimal) at address 0x1000 0000 is:

1000 0000	12	34	56	78

What value is stored at the address pointed to by register `$t2`? Assume that the memory location pointed to `$t2` is initialized to 0xFFFF FFFF.

2.24.2 [5] <2.9> Assume that the data (in hexadecimal) at address 0x1000 0000 is:

1000 0000	80	80	80	80

What value is stored at the address pointed to by register `$t2`? Assume that the memory location pointed to `$t2` is initialized to 0x0000 0000.

2.24.3 [5] <2.9> Assume that the data (in hexadecimal) at address 0x1000 0000 is:

1000 0000	11	00	00	FF

What value is stored at the address pointed to by register `$t2`? Assume that the memory location pointed to `$t2` is initialized to 0x5555 5555.

Exercise 2.25

In this exercise, you will explore 32-bit constants in MIPS. For the following problems, you will be using the binary data in the table below.

a.	$1010\ 1101\ 0001\ 0000\ 0000\ 0000\ 0000\ 0010_{TWO}$
b.	$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{TWO}$

2.25.1 [10] <2.10> Write the MIPS code that creates the 32-bit constants listed above and stores that value to register `$t1`

2.25.2 [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 the table above?

2.25.3 [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 the table above?

2.25.4 [5] <2.6, 2.10> If the current value of the PC is 0x00400600, can you use a single branch instruction to get to the PC address as shown in the table above?

2.25.5 [10] <2.10> If the immediate field of a MIPS instruction was only 8 bits wide, write the MIPS code that creates the 32-bit constants listed above and stores that value to register `$t1`. Do not use the `lui` instruction.

For the following problems, you will be using the MIPS assembly code as listed in the table.

a.	`lui $t0, 0x1234` `ori $t0, $t0, 0x5678`
b.	`ori $t0, $t0, 0x5678` `lui $t0, 0x1234`

2.25.6 [5] <2.6, 2.10> What is the value of register `$t0` after the sequence of code in the table above?

2.25.7 [5] <2.6, 2.10> Write C code that is equivalent to the assembly code in the table. Assume that the largest constant that you can load into a 32-bit integer is 16 bits.

Exercise 2.26

For this exercise, you will explore the range of branch and jump instructions in MIPS. For the following problems, use the hexadecimal data in the table below.

a.	0x00001000
b.	0xFFFC0000

2.26.1 [10] <2.6, 2.10> If the PC is at address 0x00000000, how many branch (no jump instructions) do you need to get to the address in the table above?

2.26.2 [10] <2.6, 2.10> If the PC is at address 0x00000000, how many jump instructions (no jump register instructions or branch instructions) are required to get to the target address in the table above?

2.26.3 [10] <2.6, 2.10> In order to reduce the size of MIPS programs, MIPS designers have decided to cut the immediate field of I-type instructions from 16 bits to 8 bits. If the PC is at address 0x0000000, how many branch instructions are needed to set the PC to the address in the table above?

For the following problems, you will be using making modifications to the MIPS instruction set architecture.

a.	8 registers
b.	10 bit immediate/address field

2.26.4 [10] <2.6, 2.10> If the instruction set of the MIPS processor is modified, the instruction format must also be changed. For each of the suggested changes above, what is the impact on the range of addresses in a beq instruction? Assume that all instructions remain 32 bits long and any changes made to the instruction

format of I-type instructions only increase/decrease the immediate field of the beq instruction.

2.26.5 [10] <2.6, 2.10> If the instruction set of the MIPS processor is modified, the instruction format must also be changed. For each of the suggested changes above, what is the impact on the range of addresses a jump instruction? Assume that instructions remain 32 bits long and any changes made to the instruction format of J-type instructions only impact the address field of the jump instruction.

2.26.6 [10] <2.6, 2.10> If the instruction set of the MIPS processor is modified, the instruction format must also be changed. For each of the suggested changes above, what is the impact on the range of addresses a jump register instruction, assuming that each instruction must be 32 bits.

Exercise 2.27

In the following problems, you will be using exploring different addressing modes in the MIPS instruction set architecture. These different addressing modes are listed in the table below.

a.	Register Addressing
b.	PC-relative Addressing

2.27.1 [5] <2.10> In the table above are different addressing modes of the MIPS instruction set. Give an example MIPS instructions that shows the MIPS addressing mode.

2.27.2 [5] <2.10> For the instructions in 2.27.1, what is the instruction format type used for the given instruction?

2.27.3 [5] <2.10> List benefits and drawbacks of a particular MIPS addressing mode. Write MIPS code that shows these benefits and drawbacks.

In the following problems, you will be using the MIPS assembly code as listed below to explore the tradeoffs of the immediate field in the MIPS I-type instructions.

a.	0x00000000 0x00000004	lui $s0, 100 ori $s0, $s0, 40
b.	0x00000100 0x00000104	addi $t0, $0, 0x0000 lw $t1, 0x4000($t0)

2.27.4 [15] <2.10> For the MIPS statements above, show the bit-level instruction representation of each of the instructions in hexadecimal.

2.27.5 [10] <2.10> By reducing the size of the immediate fields of the I-type and J-type instructions, we can save on the number of bits needed to represent instructions. If the immediate field of I-type instructions were 8 bits and the immediate field of J-type instructions were 18 bits, rewrite the MIPS code above to reflect this change. Avoid using the `lui` instruction.

2.27.6 [5] <2.10> How many extra instructions are needed to execute your code in 2.27.5 MIPS statements in the table versus the code shown in the table above?

Exercise 2.28

The following table contains MIPS assembly code for a lock.

```
try:  MOV     R3,R4
      MOV     R6,R7
      LL      R2,0(R2)
      LL      R5,0(R1)
      SC      R3,0(R1)
      SC      R6,0(R1)
      BEQZ    R3,try
      MOV     R4,R2
      MOV     R7,R5
```

2.28.1 [5] <2.11> For each test and fail of the store conditional, how many instructions need to be executed?

2.28.2 [5] <2.11> For the load locked/store conditional code above, explain why this code may fail.

2.28.3 [15] <2.11> Rewrite the code above so that the code may operate correct. Be sure to avoid any race conditions.

Each entry in the following table has code and also shows the contents of various registers. The notation, "`($s1)`" shows the contents of a memory location pointed to by register `$s1`. The assembly code in each table is executed in the cycle shown on parallel processors with a shared memory space.

a.

Processor 1	Processor 2	Cycle	Processor 1		MEM	Processor 2	
			$t1	$t0	($s1)	$t1	$t0
		0	1	2	99	30	40
ll $t1, 0($s1)	ll $t1, 0($s1)	1					
sc $t0, 0($s1)		2					
	sc $t0, 0($s1)	3					

b.

Processor 1	Processor 2	Cycle	Processor 1			MEM	Processor 2		
			$s4	$t1	$t0	($s1)	$s4	$t1	$t0
		0	2	3	4	99	10	20	30
	try: add $t0, $0, $s4	1							
try: add $t0, $0, $s4	ll $t1, 0($s1)	2							
ll $t1, 0($s1)		3							
sc $t0, 0($s1)		4							
beqz $t0, try	sc $t0, 0($s1)	5							
add $s4, $0, $t1	beqz $t0, try	6							

2.28.4 [5] <2.11> Fill out the table with the value of the registers for each given cycle.

Exercise 2.29

The first three problems in this exercise refer to a critical section of the form

```
lock(lk);
operation
unlock(lk);
```

where the "operation" updates the shared variable `shvar` using the local (nonshared) variable `x` as follows:

	Operation
a.	`shvar=shvar+x;`
b.	`shvar=min(shvar,x);`

2.29.1 [10] <2.11> Write the MIPS assembler code for this critical section, assuming 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, i.e., you should include the MIPS instructions for `lock()`, `unlock()`, `max()`, and `min()` operations. Use `ll/sc` instructions to implement the `lock()` operation, and the `unlock()` operation is simply an ordinary store instruction.

2.29.2 [10] <2.11> Repeat problem 2.29.1, 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`.

2.29.3 [10] <2.11> Compare the best-case performance of your code from 2.29.1 and 2.29.2, assuming that each instruction takes one cycle to execute. Note: best-case means that `ll/sc` always succeeds, the lock is always free when we want to `lock()`, and if there is a branch we take the path that completes the operation with fewer executed instructions.

2.29.4 [10] <2.11> Using your code from 2.29.2 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.

2.29.5 [10] <2.11> Explain why in your code from 2.29.2 register `$a1` contains the address of variable `shvar` and not the value of that variable, and why register `$a2` contains the value of variable `x` and not its address.

2.29.6 [10] <2.11> If we want to atomically perform the same operation on two shared variables (e.g., `shvar1` and `shvar2`) in the same critical section, we can do this easily using the approach from 2.29.1 (simply put both updates between the lock operation and the corresponding unlock operation). Explain why we cannot do this using the approach from 2.29.2., i.e., why we cannot use `ll/sc` to access both shared variables in a way that guarantees that both updates are executed together as a single atomic operation.

Exercise 2.30

Assembler pseudoinstructions are not a part of the MIPS instruction set, but often appear in MIPS programs. The table below contains some MIPS pseudoinstructions that ,when assembled, are translated to other MIPS assembly instructions.

a.	`move $t1, $t2`
b.	`beq $t1, small, LOOP`

2.30.1 [5] <2.12> For each pseudo instruction in the table above, produce a minimal sequence of actual MIPS instructions to accomplish the same thing. You may need to use temporary registers in some cases. In the table `large` refers to a number that requires 32 bits to represent and `small` to a number that can fit into 16 bits.

The table below contains some MIPS pseudoinstructions, that when assembled, are translated to other MIPS assembly instructions.

a.	`la $s0, v`
b.	`blt $a0, $v0, loop`

2.30.2 [5] <2.12> Does the instruction in the table above need to be edited during the link phase? Why?

Exercise 2.31

The table below contains the link-level details of two different procedures. In this exercise, you will be taking the place of the linker.

a.

	Procedure A				Procedure B			
Text Segment	Address	Instruction			Text Segment	Address	Instruction	
	0	lw $a0, 0($gp)				0	sw $a1, 0($gp)	
	4	jal 0				4	jal 0	
	...	...				...	...	
Data Segment	0	(X)			Data Segment	0	(Y)	
	...	...				...	...	
Relocation Info	Address	Instruction Type	Dependency		Relocation Info	Address	Instruction Type	Dependency
	0	lw	X			0	sw	Y
	4	jal	B			4	jal	A
Symbol Table	Address	Symbol			Symbol Table	Address	Symbol	
	–	X				–	Y	
	–	B				–	A	

b.

	Procedure A				Procedure B			
Text Segment	Address	Instruction			Text Segment	Address	Instruction	
	0	lui $at, 0				0	sw $a0, 0($gp)	
	4	ori $a0, $at, 0				4	jmp 0	
	8	jal 0				...	...	
	...	...				0x180	jr $ra	
						...	...	
Data Segment	0	(X)			Data Segment	0	(Y)	
	...	...				...	...	
Relocation Info	Address	Instruction Type	Dependency		Relocation Info	Address	Instruction Type	Dependency
	0	lui	X			0	sw	Y
	4	ori	X			4	jmp	F00
	8	jal	B					
Symbol Table	Address	Symbol			Symbol Table	Address	Symbol	
	–	X				–	Y	
	–	B				0x180	F00	

2.31.1 [5] <2.12> Link the object files above to form the executable file header. Assume that Procedure A has a text size of 0x140, data size of 0x40 and Procedure B has a text size of 0x300 and data size of 0x50. Also assume the memory allocation strategy as shown in Figure 2.13.

2.31.2 [5] <2.12> What limitations, if any, are there on the size of an executable?

2.31.3 [5] <2.12> Given your understanding of the limitations of branch and jump instructions, why might an assembler have problems directly implementing branch and jump instructions in an object file?

Exercise 2.32

The first three problems in this exercise assume that function `swap`, instead of the code in Figure 2.24, is defined in C as follows:

a.
```
void swap(int v[], int k, int j)
  int temp;
  temp=v[k];
  v[k]=v[j];
  v[j]=temp;
}
```

b.
```
void swap(int *p)
  int temp;
  temp=*p;
  *p=*(p+1);
  *(p+1)=*p;
}
```

2.32.1 [10] <2.13> Translate this function into MIPS assembler code.

2.32.2 [5] <2.13> What needs to change in the `sort` function?

2.32.3 [5] <2.13> If we were sorting 8-bit bytes, not 32-bit words, how would your MIPS code for `swap` in 2.32.1 change?

For the remaining three problems in this exercise, we assume that the `sort` function from Figure 2.27 is changed in the following way:

a.	Use s-registers instead of t-registers.
b.	Use the bltz (branch on less than zero) instruction instead of slt and bne at the `for2tst` label.

2.32.4 [5] <2.13> Does this change affect the code for saving and restoring registers in Figure 2.27?

2.32.5 [10] <2.13> When sorting a 10-element array that was already sorted, how many more (or fewer) instructions are executed as a result of this change?

2.32.6 [10] <2.13> When sorting a 10-element array that was sorted in descending order (opposite of the order that `sort()` creates), how many more (or fewer) instructions are executed as a result of this change?

Exercise 2.33

The problems in this exercise refer to the following function, given as array code:

a.

```
int find(int a[], int n, int x){
  int i;
  for(i=0;i!=n;i++)
    if(a[i]==x)
      return i;
  return -1;
}
```

b.

```
int count(int a[], int n, int x){
  int res=0;
  int i;
  for(i=0;i!=n;i++)
    if(a[i]==x)
      res=res+1;
  return res;
}
```

2.33.1 [10] <2.14> Translate this function into MIPS assembly.

2.33.2 [10] <2.14> Convert this function into pointer-based code (in C).

2.33.3 [10] <2.14> Translate your pointer-based C code from 2.33.2 into MIPS assembly.

2.33.4 [5] <2.14> Compare the worst-case number of executed instructions per nonlast loop iteration in your array-based code from 2.33.1 and your pointer-based code from 2.33.3. Note: the worst-case occurs when branch conditions are such that the longest path through the code is taken, i.e., if there is an if statement, the result of the condition check is such that the path with more instructions is taken. However, if the result of the condition check would cause the loop to exit, then we assume that the path that keeps us in the loop is taken.

2.33.5 [5] <2.14> Compare the number of temporary registers (t-registers) needed for your array-based code from 2.33.1 and for your pointer-based code from 2.33.3.

2.33.6 [5] <2.14> What would change in your answer from 2.33.4 if registers $t0-$t7 and $a0-$a3 in the MIPS calling convention were all callee-saved, just like $s0-$s7?

Exercise 2.34

The table below contains ARM assembly code. In the following problems, you will translate ARM assembly code to MIPS.

a.		MOV	r0, #10	;init loop counter to 10
	LOOP:	ADD	r0, r1	;add r1 to r0
		SUBS	r0, 1	;decrement counter
		BNE	LOOP	;if Z=0 repeat loop
b.		ROR	r1, r2, #4	;r1 = $r2_{3:0}$ concatenated with $r2_{31:4}$

2.34.1 [5] <2.16> For the table above, translate this ARM assembly code to MIPS assembly code. Assume that ARM registers r0, r1, and r2 hold the same values as MIPS registers $s0, $s1, and $s2, respectively. Use MIPS temporary registers ($t0, etc.) where necessary.

2.34.2 [5] <2.16> For the ARM assembly instructions in the table above, show the bit fields that represent the ARM instructions.

The table below contains MIPS assembly code. In the following problems, you will translate MIPS assembly code to ARM.

a.	slt $t0, $s0, $s1 bit $t0, $0, FARAWAY
b.	add $s0, $s1, $s2

2.34.3 [5] <2.16> For the table above, find the ARM assembly code that corresponds to the sequence of MIPS assembly code.

2.34.4 [5] <2.16> Show the bit fields that represent the ARM assembly code.

Exercise 2.35

The ARM processor has a few different addressing modes that are not supported in MIPS. The following problems explore these new addressing modes.

a.	LDR r0, [r1]	; r0 = memory[r1]
b.	LDMIA r0, {r1, r2, r4}	; r1 = memory[r0], r2 = memory[r0+4] ; r4 = memory[r0+8]

2.35.1 [5] <2.16> Identify the type of addressing mode of the ARM assembly instructions in the table above.

2.35.2 [5] <2.16> For the ARM assembly instructions above, write a sequence of MIPS assembly instructions to accomplish the same data transfer.

In the following problems, you will compare code written using the ARM and MIPS instruction sets. The following table shows code written in the ARM instruction set.

a.

```
        LDR   r0, =Table1          ;load base address of table
        LDR   r1, #100             ;initialize loop counter
        EOR   r2, r2, r2           ;clear r2
ADDLP:  LDR   r4, [r0]             ;get first addition operand
        ADD   r2, r2, r4           ;add to r2
        ADD   r0, r0, #4           ;increment to next table element
        SUBS  r1, r1, #1           ;decrement loop counter
        BNE   ADDLP                ;if loop counter != 0, go to ADDLP
```

b.

```
        ROR    r1, r2, #4          ;r1 = r2[3:0] concatenated with r2[31:4]
```

2.35.3 [10] <2.16> For the ARM assembly code above, write an equivalent MIPS assembly code routine.

2.35.4 [5] <2.16> What is the total number of ARM assembly instructions required to execute the code? What is the total number of MIPS assembly instructions required to execute the code?

2.35.5 [5] <2.16> Assuming that the average CPI of the MIPS assembly routine is the same as the average CPI of the ARM assembly routine, and the MIPS processor has an operation frequency that is 1.5 times the ARM processor, how much faster is the ARM processor than the MIPS processor?

Exercise 2.36

The ARM processor has an interesting way of supporting immediate constants. This exercise investigates those differences. The following table contains ARM instructions.

a.

```
ADD, r3, r2, r1, LSL #3    ;r3 = r2 + (r1 << 3)
```

b.

```
ADD, r3, r2, r1, ROR #3    ;r3 = r2 + (r1, rotated_right 3 bits)
```

2.36.1 [5] <2.16> Write the equivalent MIPS code for the ARM assembly code above.

2.36.2 [5] <2.16> If the register R1 had the constant value of 8, rewrite your MIPS code to minimize the number of MIPS assembly instructions needed.

2.36.3 [5] <2.16> If the register R1 had the constant value of 0x06000000, rewrite your MIPS code to minimize the number of MIPS assembly instructions needed.

The following table contains MIPS instructions.

a.	`addi r3, r2, 0x1`
b.	`addi r3, r2, 0x8000`

2.36.4 [5] <2.16> For the MIPS assembly code above, write the equivalent ARM assembly code.

Exercise 2.37

This exercise explores the differences between the MIPS and x86 instruction sets. The following table contains x86 assembly code.

a.	`mov edx, [esi+4*ebx]`

b.	`START: mov ax, 00101100b`
	`mov cx, 00000011b`
	`mov bx, 11110000b`
	`and ax, bx`
	`or ax, cx`

2.37.1 [10] <2.17> Write pseudo code for the given routine.

2.37.2 [10] <2.17> What is the equivalent MIPS for the given routine?

The following table contains x86 assembly instructions.

a.	`mov edx, [esi+4*ebx]`
b.	`add eax, 0x12345678`

2.37.3 [5] <2.17> For each assembly instruction, show the size of each of the bit fields that represent the instruction. Treat the label `MY_FUNCTION` as a 32-bit constant.

2.37.4 [10] <2.17> Write equivalent MIPS assembly statements.

Exercise 2.38

The x86 instruction set includes the REP prefix that causes the instruction to be repeated a given number of times or until a condition is satisfied. The first three problems in this exercise refer to the following x86 instruction:

	Instruction	Interpretation
a.	REP MOVSB	Repeat until ECX is zero: Mem8[EDI]=Mem8[ESI], EDI=EDI+1, ESI=ESI+1, ECX=ECX-1
b.	REP MOVSD	Repeat until ECX is zero: Mem32[EDI]=Mem32[ESI], EDI=EDI+4, ESI=ESI+4, ECX=ECX-1

2.38.1 [5] <2.17> What would be a typical use for this instruction?

2.38.2 [5] <2.17> Write MIPS code that performs the same operation, assuming that `$a0` corresponds to ECX, `$a1` to EDI, `$a2` to ESI, and `$a3` to EAX.

2.38.3 [5] <2.17> If the x86 instruction takes one cycle to read memory, one cycle to write memory, and one cycle for each register update, and if MIPS takes one cycle per instruction, what is the speed-up of using this x86 instruction instead of the equivalent MIPS code when ECX is very large? Assume that the clock cycle time for x86 and MIPS is the same.

The remaining three problems in this exercise refer to the following function, given in both C and x86 assembly. For each x86 instruction, we also show its length in the x86 variable-length instruction format and the interpretation (what the instruction does). Note that the x86 architecture has very few registers compared to MIPS, and as a result the x86 calling convention is to push all arguments onto the stack. The return value of an x86 function is passed back to the caller in the EAX register.

	C code	x86 code

a.	`int f(int a, int b){`
	`  return a+b;`
	`}`
		`f: push %ebp  ; 1B, push %ebp to stack`
		`mov %esp,%ebp  ; 2B, move %esp to %ebp`
		`mov 0xc(%ebp),%eax  ; 3B, load 2nd arg to %eax`
		`add 0x8(%ebp),%eax  ; 3B, add 1st arg to %eax`
		`pop %ebp  ; 1B, restore %ebp`
		`ret  ; 1B, return`

b.	`void f(int *a, int *b){`
	`  *a=*a+*b;`
	`  *b=*a;`
	`}`
		`f: push %ebp  ; 1B, push %ebp to stack`
		`mov %esp,%ebp  ; 2B, move %esp to %ebp`
		`mov 8(%ebp),%eax  ; 3B, load 1st arg into %eax`
		`mov 12(%ebp),%ecx  ; 3B, load 2nd arg into %ecx`
		`mov (%eax),%edx  ; 2B, load *a into %edx`
		`add (%ecx),%edx  ; 2B, add *b to %edx`
		`mov %edx,(%eax)  ; 2B, store %edx to *a`
		`mov %edx,(%ecx)  ; 2B, store %edx to *b`
		`pop %ebp  ; 1B, restore %ebp`
		`ret  ; 1B, return`

2.38.4 [5] <2.17> Translate this function into MIPS assembly. Compare the size (how many bytes of instruction memory are needed) for this x86 code and for your MIPS code.

2.38.5 [5] <2.17> If the processor can execute two instructions per cycle, it must at least be able to read two consecutive instructions in each cycle. Explain how it would be done in MIPS and how it would be done in x86.

2.38.6 [5] <2.17> If each MIPS instruction takes one cycle, and if each x86 instruction takes one cycle plus a cycle for each memory read or write it has to perform, what is the speed-up of using x86 instead of MIPS? Assume that the clock cycle time is the same in both x86 and MIPS, and that the execution takes the shortest possible path through the function (i.e., every loop is exited immediately and every if statement takes the direction that leads toward the return from the function). Note that x86 ret instruction reads the return address from the stack.

Exercise 2.39

The CPI of the different instruction types is given in the following table.

	Arithmetic	Load/Store	Branch
a.	2	10	3
b.	1	10	4

2.39.1 [5] <2.18> Assume the following instruction breakdown given for executing a given program:

	Instructions (in millions)
Arithmetic	500
Load/Store	300
Branch	100

What is the execution time for the processor if the operation frequency is 5 GHz?

2.39.2 [5] <2.18> 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?

2.39.3 [5] <2.18> Suppose that we find a way to double the performance of arithmetic instructions? What is the overall speed-up of our machine? What if we find a way to improve the performance of arithmetic instructions by 10 times!?

The following table shows the proportions of instruction execution for the different instruction types.

	Arithmetic	Load/Store	Branch
a.	60%	20%	20%
b.	80%	15%	5%

2.39.4 [5] <2.18> Given the instruction mix above 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.

2.39.5 [5] <2.18> 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?

2.39.6 [5] <2.18> 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?

Exercise 2.40

The first three problems in this exercise refer to the following function, given in MIPS assembly. Unfortunately, the programmer of this function has fallen prey to the pitfall of assuming that MIPS is a word-addressed machine, but in fact MIPS is byte addressed.

a.
```
; int f(int a[], int n, int x):
f:  move $v0,$zero   ; ret=0
    move $t0,$zero   ; i=0
L:  add  $t1,$t0,$a0 ; &(a[i])
    lw   $t1,0($t1)  ; read a[i]
    bne  $t1,$a2,S   ; if(a[i]==x)
    addi $v0,$v0,1   ;   ret++;
S:  addi $t0,$t0,1   ; i++
    bne  $t0,$a1,L   ; repeat if i!=n
    jr   $ra         ; return ret
```

b.
```
; void f(int *a, int *b, int n);
f: move $t0,$a0       ; p=a
   move $t1,$a1       ; q=b
   add  $t2,$a2,$a0   ; &(a[n])
L: lw   $t3,0($t0)    ; read *p
   lw   $t4,0($t1)    ; read *q
   add  $t3,$t3,$t4   ; *p+*q
   sw   $t3,0($t0)    ; *p=*p+*q
   addi $t0,$t0,1     ; p=p+1
   addi $t1,$t1,1     ; q=q+1
   bne  $t0,$t2,L     ; repeat if p!=&(a[n])
   jr   $ra           ; return
```

Note that in MIPS assembly the ";" character denotes that the remainder of the line is a comment.

2.40.1 [5] <2.18> The MIPS architecture requires word-sized accesses (lw and sw) to be word-aligned, i.e. the lowermost 2 bits of the address must both be zero. If an address is not word-aligned, the processor raises a "bus error" exception. Explain how this alignment requirement affects the execution of this function.

2.40.2 [5] <2.18> If "a" was a pointer to the beginning of an array of one-byte elements, and if we replaced lw and sw with lb (load byte) and sb (store byte), respectively, would this function be correct? Note: lb reads a byte from memory, sign-extends it, and places it into the destination register, while sb stores the least-significant byte of the register into memory.

2.40.3 [5] <2.18> Change this code to make it correct for 32-bit integers.

The remaining three problems in this exercise refer to a program that allocates memory for an array, fills the array with some numbers, calls the sort function from Figure 2.27, and then prints out the array. The main function of the program is as follows (given as both C and MIPS code):

main code in C	MIPS version of the main code

`main(){`
`  int *v;`
`  int n=5;`
`  v=my_alloc(5);`
`  my_init(v,n);`
`sort(v,n);`
`.`
`.`
`.`
	`main:`
	`li   $s0,5`
	`move $a0,$s0`
	`jal  my_alloc`
	`move $s1,$v0`
	`move $a0,$s1`
	`move $a1,$s0`
	`jal  my_init`
	`move $a0,$s1`
	`move $a1,$s0`
	`jal  sort`

The `my_alloc` function is defined as follows (given as both C and MIPS code). Note that the programmer of this function has fallen prey to the pitfall of using a pointer to an automatic variable `arr` outside the function in which it is defined.

my_alloc in C	MIPS code for my_alloc

`int *my_alloc(int n){`
`  int arr[n];`
`  return arr;`
`}`
	`my_alloc:`
	`  addu  $sp,$sp,-4  ; Push`
	`  sw  $fp,0($sp)  ; $fp to stack`
	`  move  $fp,$sp  ; Save $sp in $fp`
	`  sll  $t0,$a0,2  ; We need 4*n bytes`
	`  sub  $sp,$sp,$t0  ; Make room for arr`
	`  move  $v0,$sp  ; Return address of arr`
	`  move  $sp,$fp  ; Restore $sp from $fp`
	`  lw  $fp,0(sp)  ; Pop $fp`
	`  addiu  $sp,$sp,4  ; from stack`
	`  jr  ra`

The `my_init` function is defined as follows (MIPS code):

a.
```
my_init:
    move   $t0,$zero     ; i=0
    move   $t1,$a0
L:  sw     $zero,0($t1) ; v[i]=0
    addiu  $t1,$t1,4
    addiu  $t0,$t0,1     ; i=i+1
    bne    $t0,$a1,L     ; until i==n
    jr     $ra
```

b.
```
my_init:
    move   $t0,$zero     ; i=0
    move   $t1,$a0
L:  sub    $t2,$a1,$t0
    sw     $t2,0($t1)    ; a[i]=n-i
    addiu  $t1,$t1,4
    addiu  $t0,$t0,1     ; i=i+1;
    bne    $t0,$a1,L     ; until i==n
    jr     $ra
```

2.40.4 [5] <2.18> What are the contents (values of all five elements) of array `v` right before the "`jal sort`" instruction in the main code is executed?

2.40.5 [15] <2.18, 2.13> What are the contents of array `v` right before the `sort` function enters its outer loop for the first time? Assume that registers `$sp`, `$s0`, `$s1`, `$s2`, and `$s3` have values of 0x1000, 20, 40, 7, and 1, respectively, at the beginning of the main code (right before "`li $s0, 5`" is executed).

2.40.6 [10] <2.18, 2.13> What are the contents of the 5-element array pointed by `v` right after "`jal sort`" returns to the main code?

Answers to Check Yourself

§2.2, page 80: MIPS, C, Java
§2.3, page 87: 2) Very slow
§2.4, page 93: 3) -8_{ten}
§2.5, page 101: 4) `sub $s2, $s0, $s1`
§2.6, page 104: Both. AND with a mask pattern of 1s will leaves 0s everywhere but the desired field. Shifting left by the right 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 111: I. All are true. II. 1).
§2.8, page 122: Both are true.
§2.9, page 127: I. 2) II. 3)
§2.10, page 136: I. 4) +-128K. II. 6) a block of 256M. III. 4) `sll`
§2.11, page 139: Both are true.
§2.12, page 148: 4) Machine independence.