Alexandria University

Faculty of Engineering

Division of Communications & Electronics

CC322 Computer Architecture
Sheet 1: MIPS Instruction Set Architecture

Exercises 1-16 from the reference book: "Digital design and computer Architecture by David Harris, 2nd Edition"

Exercises 17-22 from the exercises of: "Architecture des Ordinateurs Course, 2014, EPFL"

1. Consider memory storage of a 32-bit word stored at memory word 42 in a byte addressable memory.

(a) What is the byte address of memory word 42?
(b) What are the byte addresses that memory word 42 spans?
(c) Draw the number 0xFF223344 stored at word 42 in both big-endian and little-endian machines. Clearly label the byte address corresponding to each data byte value.

2. Repeat Exercise 1 for memory storage of a 32-bit word stored at memory word 15 in a byte-addressable memory.

3. Explain how the following program can be used to determine whether a computer is big-endian or little-endian:

```
li $t0, 0xABCD9876
sw $t0, 100($0)
lb $s5, 101($0)
```

4. The *nori* instruction is not part of the MIPS instruction set, because the same functionality can be implemented using existing instructions. Write a short assembly code snippet that has the following functionality: *$t0 = $t1 NOR 0xF234*. Use as few instructions as possible.

5. Implement the following high-level code segments using the *slt* instruction. Assume the integer variables *g* and *h* are in registers *$s0* and *$s1*, respectively.

(a)
```
if (g > h)
    g = g + h;
else
    g = g - h;
```

(b)
```
if (g >= h)
    g = g + 1;
else
    h = h - 1;
```

(c)
```
if (g <= h)
    g = 0;
else
    h = 0;
```

6. Write a function in a high-level language for *int find42(int array[], int size)*. *size* specifies the number of elements in *array*, and *array* specifies the base address of the array. The function should return the index number of the first array entry that holds the value 42. If no array entry is 42, it should return the value –1.

7. The high-level function *strcpy* copies the character string *src* to the character string *dst* (see page 360).

```
// C code
void strcpy(char dst[], char src[]) {
  int i = 0;

  do {
    dst[i] = src[i];
  } while (src[i++]);
}
```

(a) Implement the *strcpy* function in MIPS assembly code. Use *$s0* for *i*.

(b) Draw a picture of the stack before, during, and after the *strcpy* function call. Assume *$sp = 0x7FFFFF00* just before *strcpy* is called.

8. Consider the MIPS assembly code below. *func1*, *func2*, and *func3* are non-leaf functions. *func4* is a leaf function. The code is not shown for each function, but the comments indicate which registers are used within each function.

```
0x00401000   func1 : . . .        # func1 uses $s0–$s1
0x00401020           jal func2
 . . .
0x00401100   func2:  . . .        # func2 uses $s2–$s7
0x0040117C           jal func3
 . . .
0x00401400   func3:  . . .        # func3 uses $s1–$s3
0x00401704           jal func4
 . . .
0x00403008   func4:  . . .        # func4 uses no preserved
0x00403118           jr $ra       # registers
```

9. Each number in the Fibonacci series is the sum of the previous two numbers. The following table lists the first few numbers in the series, *fib(n)*.

n	1	2	3	4	5	6	7	8	9	10	11	...
fib(*n*)	1	1	2	3	5	8	13	21	34	55	89	...

(a) What is *fib(n)* for *n = 0* and *n = –1?*

(b) Write a function called *fib* in a high-level language that returns the Fibonacci number for any nonnegative value of *n*. Hint: You probably will want to use a loop. Clearly comment your code.

(c) Convert the high-level function of part (b) into MIPS assembly code. Add comments after every line of code that explain clearly what it does. Use the SPIM simulator to test your code on *fib(9)*.

10. Consider C Code **Example 6.27**. For this exercise, assume factorial is called with input argument *n = 5.*

(a) What value is in *$v0* when factorial returns to the calling function?

(b) Suppose you delete the instructions at addresses *0x98* and 0xBC that save and restore *$ra*. Will the program (1) enter an infinite loop but not crash; (2) crash (cause the stack to grow beyond the dynamic data segment or the PC to jump to a location outside the program); (3) produce an incorrect value in *$v0* when the program returns to loop (if so, what value?), or (4) run correctly despite the deleted lines?

(c)Repeat part (b) when the instructions at the following instruction addresses are deleted:

(i) 0x94 and 0xC0 (instructions that save and restore *$a0*)

(ii) 0x90 and 0xC4 (instructions that save and restore *$sp*). Note: the *factorial* label is not deleted

(iii) 0xAC (an instruction that restores *$sp*)

11. Ben Bitdiddle is trying to compute the function *f(a, b) = 2a + 3b* for nonnegative *b*. He goes overboard in the use of function calls and recursion and produces the following high-level code for functions *f* and *f2*.

```
// high-level code for functions f and f2
int f(int a, int b) {
  int j;
  j = a;
  return j + a + f2(b);
}

int f2(int x)
{
  int k;
  k = 3;
  if (x == 0) return 0;
  else return k + f2(x – 1);
}
```

Ben then translates the two functions into assembly language as follows. He also writes a function, test, that calls the function *f(5, 3)*.

```
# MIPS assembly code
# f: $a0 = a, $a1 = b, $s0 = j; f2: $a0 = x, $s0 = k

0x00400000  test: addi $a0, $0, 5     # $a0 = 5 (a = 5)
0x00400004        addi $a1, $0, 3     # $a1 = 3 (b = 3)
0x00400008        jal  f              # call f(5, 3)
0x0040000C  loop: j    loop           # and loop forever

0x00400010  f:    addi $sp, $sp, -16  # make room on the stack
                                      # for $s0, $a0, $a1, and $ra
0x00400014        sw   $a1, 12($sp)   # save $a1 (b)
0x00400018        sw   $a0, 8($sp)    # save $a0 (a)
0x0040001C        sw   $ra, 4($sp)    # save $ra
0x00400020        sw   $s0, 0($sp)    # save $s0
0x00400024        add  $s0, $a0, $0   # $s0 = $a0 (j = a)
0x00400028        add  $a0, $a1, $0   # place b as argument for f2
0x0040002C        jal  f2             # call f2(b)
0x00400030        lw   $a0, 8($sp)    # restore $a0 (a) after call
0x00400034        lw   $a1, 12($sp)   # restore $a1 (b) after call
0x00400038        add  $v0, $v0, $s0  # $v0 = f2(b) + j
0x0040003C        add  $v0, $v0, $a0  # $v0 = (f2(b) + j) + a
0x00400040        lw   $s0, 0($sp)    # restore $s0
0x00400044        lw   $ra, 4($sp)    # restore $ra
0x00400048        addi $sp, $sp, 16   # restore $sp (stack pointer)
0x0040004C        jr   $ra            # return to point of call

0x00400050  f2:   addi $sp, $sp, -12  # make room on the stack for
                                      # $s0, $a0, and $ra
0x00400054        sw   $a0, 8($sp)    # save $a0 (x)
0x00400058        sw   $ra, 4($sp)    # save return address
0x0040005C        sw   $s0, 0($sp)    # save $s0
0x00400060        addi $s0, $0, 3     # k = 3
0x00400064        bne  $a0, $0, else  # x = 0?
0x00400068        addi $v0, $0, 0     # yes: return value should be 0
0x0040006C        j    done           # and clean up
0x00400070  else: addi $a0, $a0, -1   # no: $a0 = $a0 - 1 (x = x - 1)
0x00400074        jal  f2             # call f2(x - 1)
0x00400078        lw   $a0, 8($sp)    #   restore $a0 (x)
0x0040007C        add  $v0, $v0, $s0  #   $v0 = f2(x - 1) + k
0x00400080  done: lw   $s0, 0($sp)    #   restore $s0
0x00400084        lw   $ra, 4($sp)    #   restore $ra
0x00400088        addi $sp, $sp, 12   #   restore $sp
0x0040008C        jr   $ra            #   return to point of call
```

You will probably find it useful to make drawings of the stack similar to the one in Figure 6.26 of the reference book to help you answer the following questions.

(a)If the code runs starting at test, what value is in *$v0* when the program gets to loop ? Does his program correctly compute *2a + 3b?*

(b)Suppose Ben deletes the instructions at addresses 0x0040001C and 0x00400044 that save and restore $ra. Will the program (1) enter an infinite loop but not crash; (2) crash (cause the stack to grow beyond the dynamic data segment or the PC to jump to a location outside the program); (3) produce an incorrect value in $v0 when the program returns to loop (if so, what value?), or (4) run correctly despite the deleted lines?

(c)Repeat part (b) when the instructions at the following instruction addresses are deleted. Note that labels aren't deleted, only instructions.

(i) 0x00400018 and 0x00400030 (instructions that save and restore $a0)

(ii) 0x00400014 and 0x00400034 (instructions that save and restore $a1)

(iii) 0x00400020 and 0x00400040 (instructions that save and restore $s0)

(iv) 0x00400050 and 0x00400088 (instructions that save and restore $sp)

(v) 0x0040005C and 0x00400080 (instructions that save and restore $s0)

(vi) 0x00400058 and 0x00400084 (instructions that save and restore $ra)

(vii) 0x00400054 and 0x00400078 (instructions that save and restore $a0)

12. Consider the following C code snippet.

```
// C code
void setArray(int num) {
  int i;
  int array[10];

  for (i = 0; i < 10; i = i + 1) {
    array[i] = compare(num, i);
  }
}

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;
}
```

(a)Implement the C code snippet in MIPS assembly language. Use $s0 to hold the variable i. Be sure to handle the stack pointer appropriately. The array is stored on the stack of the *setArray* function (see Section 6.4.6 of the reference).

(b)Assume *setArray* is the first function called. Draw the status of the stack before calling *setArray* and during each function call. Indicate the names of registers and variables stored on the stack, mark the location of *$sp*, and clearly mark each stack frame.

(c) How would your code function if you failed to store *$ra* on the stack?

13. Consider the following high-level function.

```
// C code
int f(int n, int k) {
  int b;

  b = k + 2;
  if (n == 0) b = 10;
  else b = b + (n * n) + f(n - 1, k + 1);
  return b * k;
}
```

(a)Translate the high-level function f into MIPS assembly language. Pay particular attention to properly saving and restoring registers across function calls and using the MIPS preserved register conventions. Clearly comment your code. You can use the MIPS *mul* instruction. The function starts at instruction address 0x00400100. Keep local variable b in *$s0*.

(b) Step through your function from part (a) by hand for the case of *f(2, 4)*. Draw a picture of the stack similar to the one in Figure 6.26(c) of the reference book. Write the register name and data value stored at each location in the stack and keep track of the stack pointer value (*$sp*). Clearly mark each stack frame. You might also find it useful to keep track of the values in *$a0*, *$a1*, *$v0*, and *$s0* throughout execution. Assume that when *f* is called, *$s0* = 0xABCD and *$ra* = 0x400004. What is the final value of *$v0?*

14. The following questions examine the limitations of the jump instruction, *j*. Give your answer in number of instructions relative to the jump instruction.

(a) In the worst case, how far can the jump instruction (j) jump forward (i.e., to higher addresses)? (The worst case is when the jump instruction cannot jump far.) Explain using words and examples, as needed.

(b) In the best case, how far can the jump instruction (j) jump forward? (The best case is when the jump instruction can jump the farthest.) Explain.

(c) In the worst case, how far can the jump instruction (j) jump backward (to lower addresses)? Explain.

(d) In the best case, how far can the jump instruction (j) jump backward? Explain.

15. Write a function in high-level code that takes a 10-entry array of 32-bit integers stored in little-endian format and converts it to big-endian format. After writing the high-level code, convert it to MIPS assembly code. Comment all your code and use a minimum number of instructions.

16. Consider two strings: *string1* and *string2*.

(a)Write high-level code for a function called *concat* that concatenates (joins together) the two strings: *void concat(char string1[], char string2[], char stringconcat[])*. The function does not return a value. It concatenates *string1* and *string2* and places the resulting string in *stringconcat*. You may assume that the character array *stringconcat* is large enough to accommodate the concatenated string.

(b) Convert the function from part (a) into MIPS assembly language

17. Consider the following MIPS program.

```
start:
    add  $v0, $zero, $zero
    add  $t0, $zero, $zero
outer:
    sltu $t2, $t0, $a1
    beq  $t2, $zero, fin
    lw   $t3, 0($a0)
    addi $t4, $zero, 32
inner:
    beq  $t4, $zero, next
    andi $t1, $t3, 1
    add  $v0, $v0, $t1
    srl  $t3, $t3, 1
    subi $t4, $t4, 1
    j    inner
next:
    addi $t0, $t0, 1
    addi $a0, $a0, 4
    j    outer
fin:
    jr   $ra
```

a) Describe the function (purpose) of the program in one sentence.

b) Is it necessary to use the two instructions sltu/beq for the loop test, and is it possible to use one instruction instead? If so, simplify the program.

c) Mark the instruction(s) that can cause an overflow (ignoring the instructions that compute addresses and indices).

d) Correct the program in order to take into account a possible overflow and return ’-1’ instead of the result if an overflow occurs (again, ignore the instructions that compute addresses and indexes).

e) Is it possible to modify the program and minimize the number of times the internal loop is being executed? Show the eventual modifications of the program (an idea: the loop counter is not necessary).

18. Consider the following MIPS program:

```
    add  $t0, $zero, $zero
    add  $v0, $zero, $zero
    add  $v1, $zero, $zero

Loop:
    sltu $t2, $t0, $a1
    beq  $t2, $zero, fin
    lw   $t1, 0($a0)
    sltu $t2, $t1, $v0
    bne  $t2, $zero, skip
    add  $v0, $t1, $zero
    add  $v1, $t0, $zero

Skip:
    addi $t0, $t0, 1
    addi $a0, $a0, 4
    j  loop

fin:
```

a) The program inputs are given in registers $a0 and $a1. The program outputs are returned via registers $v0 and $v1. What does this program do (in a sentence)? What are the values returned in $v0 and $v1?

b) What type of quantities are stored in the array (explain the answer)?

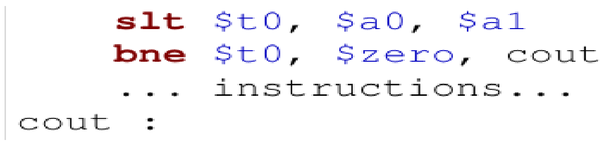
c) Adapt the program (1): make it a procedure (the values of $a0 and $a1 should be preserved).

d) Adapt the program (2): keep its functionality but let it operate on an array of unsigned bytes (avoid using the lb instruction!). Assume a little-endian machine.

e) Adapt the program resulting from the previous point, considering that the words in memory have been previously stored in big-endian.

19.

a) The following instructions, in MIPS assembly, represent a control structure very common in high-level programming languages (Java, Ada, C,. . .). Which structure is it?



```
        slt $t0, $a0, $a1
        bne $t0, $zero, cout
        ... instructions...
cout  :
```

b) Write a MIPS assembly program equivalent to the following pseudo-instructions. If necessary, you can use register $t0 to memorize intermediary values. No other register can be used.

i) add ($s0),$s1,($s2) #mem[$s0]=$s1+mem[$s2]

This MIPS instruction does not exist, because it uses an addressing mode not supported by RISC processors.

ii) SWAP $s0 # bits 31-16 <-> bits 15-0

This instruction allows us to swap the 16 most significant bits with the 16 least significant ones of a 32-bit word.

iii) PUSH $s0

This instruction is not a MIPS instruction either. It decrements the stack pointer (SP), then saves $s0 at this address.

c) Decode the following two MIPS instructions:

Adresse	Code
0x10000000	0x20080020
0x10000004	0x8D090004

20. Analyse the following program, supposing that initially (at the beginning of the execution) the registers of the processor have the following values:

$a0 contains the address of a matrix of unsigned 8-bit numbers.

$a1 contains the number of rows of this matrix.

$a2 contains the number of columns of this matrix.

$a3 contains an unsigned 8-bit value.

```
start:
    add $t0, $zero, $zero
    add $t1, $zero, $zero
    add $t3, $zero, $zero

loop:
    lbu $t2, 0($a0)
    add $t2, $t2, $a3
    slt $t4, $t2, $t3
    bne $t4, $zero, skip
    add $v0, $t0, $zero
    add $v1, $t1, $zero
    add $t3, $t2, $zero

skip:
    sb   $t2, 0($a0)
    addi $a0, $a0, 1
    addi $t0, $t0, 1
    bne  $t0, $a1, loop
    add  $t0, $zero, $zero
    addi $t1, $t1, 1
    bne  $t1, $a2, loop
end:
```

a) Describe in one sentence what this program does(suppose there is no overflow). In particular, give the values of registers $v0 and $v1 at the end of the execution if $a3=0 and the matrix is:

$$\begin{pmatrix} 12 & 34 & 56 \\ 78 & 113 & 24 \\ 35 & 46 & 57 \\ 11 & 122 & 33 \end{pmatrix}$$

b) If the previous matrix is stored starting from address 1000, give the memory contents of addresses 1000 to 1008.

c) i) Can the instruction add $t2, $t2, $a3 generate a result that cannot be represented using 32 bits?

ii) Can it give a result that is not representable using 8 bits?

iii) Take into account the possible overflows and modify the program to saturate the result in case of an overflow (if the result is not representable, replace it by the greatest possible value).

21. Consider the following MIPS program:

```
    add  $t0, $a0, $zero
    add  $t1, $a1, $zero
    add  $t2, $a2, $a2
    add  $t2, $t2, $t2
    add  $t3, $t0, $t2
loop:
    lw   $t4, 0($t0)
    sw   $t4, 0($t1)
    addi $t0, $t0, 4
    addi $t1, $t1, 4
    sltu $t5, $t0, $t3
    bne  $t5, $zero, loop
```

Assume that initially registers $a0 and $a1 store addresses in memory and register $a2 stores an integer N. Registers $t0 to $t5 are used to store temporary values and $zero is a register that always has the value zero.

a) Briefly comment each line of the code.

b) Describe in one sentence what this program does (its purpose).

c) Why did the instruction addi add 4 to registers $t0 and $t1?

d) Why is the instruction sltu (set less than unsigned) used instead of the instruction slt?

22. Analyze the following MIPS function:

When the function is called, $a0 contains the memory address of a vector of 32-bit numbers and $a1 contains an integer.

a) Describe in a sentence what the program does.

b) Must the numbers contained in the vector be either signed or unsigned? Or is it possible to have both signed and unsigned numbers in the vector ? Briefly explain your answer.

c) We would like to change this program so that it can process (handle) bytes. To this effect we need a function that receives four bytes in $a0 and returns the same four bytes in the reverse order in $v0: byte B3 (bits 32-24) is swapped with byte B0 and B2 with B1. Write such a function respecting ordinary MIPS conventions.