# CS 241E Midterm Review

Fall 2019

## 1 Bits, Bytes and Words

1. Which of the following pieces of data can be represented by a single word on a 32-bit computer? What about a single bit, or a byte (8 bits)?
   (a) The number 0
   (b) The number -42
   (c) The number $2^{32}$
   (d) The number $10^{100}$
   (e) The boolean value "true"
   (f) The capital letter "C"
   (g) The character "猫"
   (h) The string "I am a kitty"
   (i) An $4 \times 8$ pixel black and white image
   (j) A PDF file containing the Fall 2019 CS241E midterm with solutions
2. You may know about the concept of "integer overflow" in programming. After taking this course, you should understand precisely why integer overflow occurs and how it works.
   (a) When you subtract one from zero, what is the result as a 32-bit unsigned integer? Write the bit sequence, as well as the value when interpreted as a base 10 number. (You can use powers of two when expressing the number in base 10.)
   (b) What is the largest 32-bit signed (two's complement) integer? Write it as a bit sequence and as a base 10 number.
   (c) When you add one to the largest 32-bit signed integer, which integer do you get? Write it as a bit sequence. Interpret this bit sequence as a 32-bit signed integer, and as a 32-bit unsigned integer, and write both in base 10.
   (d) Interpret the base 10 number $2^{31} + 42$ as a 32-bit unsigned integer and write its bit sequence. Then interpret this bit sequence as a 32-bit signed integer, and write it as a base 10 number. Explain the relationship between these two numbers in terms of modular arithmetic.
   (e) By allowing integer overflow to occur (i.e., doing arithmetic modulo $2^{32}$), and by choosing our conventions for representing integers carefully, we can implement addition for both unsigned and signed integers using the same hardware. This also works for subtraction and multiplication.

   Give an example to show that for a 32-bit machine, different hardware circuits must be used to implement integer division on unsigned and signed integers.

## 2 MIPS Architecture and Assembly Language

1. Consider the following MIPS program. Assume it is loaded with the "termination constant" stored in register 31.

```
LIS(Reg(3))
LIS(Reg(3))
BEQ(Reg(0),Reg(0),X)
JR(Reg(31))
Word(encodeSigned(-1))
Word(encodeSigned(-1))
```

   For each value of X in the set $\{-2, -1, 0, 1, 2\}$, determine whether this program will terminate normally, loop infinitely, or crash with an invalid instruction error. (The two's complement encoding of $-1$ does not correspond to a valid instruction.)

   Hint: the BEQ instruction will increment the program counter by `X * 4`. What value does the program counter have when the BEQ instruction is executed?

2. In the following questions, the goal is to write a MIPS program that stores the value of the program counter in register 3. That is, after the last line of the program is executed, the program counter and register 3 should have the same value. You do not need to return with JR(Reg(31)) afterwards – just write a snippet of code that puts the correct value in register 3.

   You are only allowed to use assembly language instructions and labels – further abstractions like variables and expressions are not allowed.

   (a) Write a program that does this. The program should work even when relocated to another address.

   Hint: use a label. The program should be very short.

   (b) Write a program that does this without using labels. Assume that register 1 contains the address of a "reserved word" somewhere in memory that your program can freely access and change. The program should work even when relocated to another address, and you can assume the address of the "reserved word" never overlaps your program code.

   Hint 1: JALR will store the current value of the program counter in register 31. But it also jumps you to another memory address – how will you get back?

   Hint 2: Code is data.

3. An *object file* consists of two parts:

   - Machine code (that is, no opcodes or labels, just a sequence of words).
   - Metadata specifying on which lines labels were defined, and on which lines they were used.

   This question concerns two object files: MAIN and PROCEDURE. Below is the definition of MAIN. For convenience, the machine code has been converted from binary to decimal.

| Address | Machine Code | Label Definitions | Label Uses |
|---|---|---|---|
| 0 | 2068 | main is 0 | |
| 4 | 65011720 | | |
| 8 | 4116 | | |
| 12 | 28 | | location at 12 |
| 16 | 16404 | | |
| 20 | 0 | | procedure at 20 |
| 24 | 16777225 | | |
| 28 | 0 | location is 28 | end at 28 |

Below is the definition of PROCEDURE.

| Address | Machine Code | Label Definitions | Label Uses |
|---|---|---|---|
| 0 | 2353201152 | procedure is 0 | |
| 4 | 2889940992 | | |
| 8 | 6291464 | | |
| 12 | 0 | | main at 12 |
| 16 | 65011720 | end is 16 | |

(a) Fill in the blanks in the table below, so that it shows the result of linking these two object files and loading them at address 100 (with MAIN coming first, immediately followed by PROCEDURE).

| Address | Machine Code |
|---|---|
| 100 | |
| 104 | 65011720 |
| 108 | 4116 |
| 112 | |
| 116 | 16404 |
| 120 | |
| 124 | 16777225 |
| 128 | |
| 132 | |
| 136 | 2889940992 |
| 140 | 6291464 |
| 144 | |
| 148 | |

(b) Here are two assembly language programs that are similar to the ones used to produce the object files above, but slightly different, because the original programs contained an error.

```
Define(main)
ADD(Reg(17),Reg(31),Reg(0))
LIS(Reg(1))
JR(Reg(17))
LIS(Reg(2))
Use(location)
LIS(Reg(8))
Use(procedure)
JALR(Reg(8))
Define(location)
Use(end)
```

```
Define(procedure)
LW(Reg(3),0,Reg(2))
SW(Reg(1),0,Reg(2))
JR(Reg(3))
Use(main)
Define(end)
JR(Reg(31))
```

Explain why the main program terminates.

Hint: What does the procedure do with the values in registers 1 and 2?

# 3 Intermediate Language

1. There is a distinction between *variables* and *variable instances*. For example, consider the following recursive procedure:

   ```
   def factorial(n: Int): Int = if(n <= 1) 1 else n * factorial(n-1)
   ```

   This procedure has one variable as a parameter. However, if we execute `factorial(3)`, this procedure will recurse three times, and three different *instances* of the variable will be created.

The *extent* of a variable instance is the time interval in which it can be accessed. Let us trace the execution of `factorial(3)` and describe the extent of each variable instance. We will write $\texttt{n}_i$ for the $i$-th instance of variable `n`.

- `factorial(3)` is called. The extent of $\texttt{n}_1$ begins.
- `factorial(2)` is called. The extent of $\texttt{n}_2$ begins.
- `factorial(1)` is called. The extent of $\texttt{n}_3$ begins.
- `factorial(1)` returns. The extent of $\texttt{n}_3$ ends.
- `factorial(2)` returns. The extent of $\texttt{n}_2$ ends.
- `factorial(3)` returns. The extent of $\texttt{n}_1$ ends.

(a) Fill in the blanks in the following table.

| Variable Kind | Instance Extent | Storage Location |
|---|---|---|
| Global variable | | Fixed memory address |
| Procedure-local | Duration of procedure | |
| Object field | | Heap |

(b) Consider the following Scala program:

```
object Kitty {
    val meow = "Meow!"
    def makeNoise(m: Int, noise:String): Int = {
        def printNoiseRepeatedly(n: Int): Unit = {
            if(n > 1) printNoiseRepeatedly(n-1)
            println(noise)
        }
        printNoiseRepeatedly(m)
        m
    }
    def main(args: Array[String]): Unit = {
        var totalNoises = 0
        totalNoises += makeNoise(1,meow)
        totalNoises += makeNoise(2,"Woof!")
        println("You made "+totalNoises+" noises. ")
    }
}
```

Trace the execution of this program, and for each variable instance, state when its extent begins and ends. Treat the variable `meow` as if it is a global variable rather than an object field.

2. Sylvie is doing Assignment 3, and she is not sure how to access variables within a chunk. She remembers that when you push a chunk onto the stack, the stack pointer is updated to point to the start of the chunk. She decides that it probably makes sense to access variables using offsets from the stack pointer.

(a) Write a test case that will cause Sylvie's compiler to produce incorrect code. You can use variables and assembly language features, but none of the abstractions from after Assignment 3 (so no expressions, conditionals, loops, or procedures).

(b) Generally speaking, which programs involving variable accesses will behave correctly under Sylvie's implementation, and which ones will fail?

(c) How can Sylvie fix her implementation?

3. Sylvie realizes while doing Assignment 4 that evaluating expressions will be more efficient if she uses registers instead of variables. She implements the expression evaluation function `binOp` as follows:

```
def binOp(leftExpr:Code, operation:Code, rightExpr:Code): Code = {
    block(
        leftExpr,  // evaluate the left expression into Reg.result
        ADD(Reg.scratch,Reg.result,Reg.zero),
        Comment("Scratch contains left expression value"),
        rightExpr, // evaluate the right expression into Reg.result
        Comment("Result contains right expression value"),
        operation  // perform the operation on Reg.scratch and Reg.result
    )
}
```

   (a) Give an example of an arithmetic expression that this strategy will evaluate incorrectly. You can use ordinary arithmetic notation instead of writing the expression in terms of `binOp`.

   (b) Using only the number 1, the addition operator, and brackets, write down three arithmetic expressions that this strategy will evaluate correctly.

4. In this question, we will add a "for-each loop" to the intermediate language. This control structure loops over the elements of an array and allows the user to operate on each element.

   This feature makes a bit more sense if we have a way to store arrays inside variables. However, implementing array variables is not easy. So for simplicity, we will pass an array into our for-each loop as two values: the starting address and the size.

   Complete the implementation of the following function:

```
def forEachLoop(address: Code, size: Code, element: Variable, body: Code): Code = {
    ???
}
```

   The `address` and `size` Codes are assumed to place the starting address and size of the array, respectively, into Reg.result. The `body` Code is the main body of the loop. The `element` Variable is used to store the current element of the array on each iteration. The `element` variable can be used in the body, and any changes to the `element` variable in the body should be reflected in the array.

   Your implementation can use all of the functions that you implemented in Assignment 4. In particular, you can use the `deref` and `assignToAddr` functions for accessing memory addresses, and you can use `whileLoop` to implement the loop. You can use `binOp` expressions and the comparisons `eqCmp`, `neCmp`, etc. You may also define shorthand for expressions which consist of a single variable access or a single numeric constant.

5. Reg.result is used for many things in our compiler. In this question, we will trace what happens to Reg.result during a call of the following procedure:

```
val n = new Variable("n")
val addOne = new Procedure("addOne",Seq(n))
addOne.code =
    binOp(read(Reg.result,n),plus,block(LIS(Reg.result),Word(encodeUnsigned(1))))
call(addOne,block(LIS(Reg.result,Word(encodeUnsigned(41)))))
```

   (a) In the following table, whenever a line of code changes the value of Reg.result, describe the new value of Reg.result. You can use phrases like "frame pointer of current procedure" or "value of variable `n`".

| Line | Call | Reg.result value |
|---|---|---|
| 0 | assign(tempVars(0), arguments(0)) | |
| 1 | Stack.allocate(paramChunks(target)) | |
| 2 | read(Reg.scratch, tempVars(0)) | |
| 3 | paramChunks(target).store(Reg.result, target.parameters(0), Reg.scratch) | |
| 4 | LIS(Reg.targetPC); Use(target.label) | |
| 5 | JALR(Reg.targetPC) | |
| **Line** | **Prologue** | **Reg.result value** |
| 6 | ADD(Reg.savedParamPtr, Reg.result, Reg.zero) | |
| 7 | Stack.allocate(frame) | |
| 8 | frame.store(Reg.result, procedure.dynamicLink, Reg.framePointer) | |
| 9 | frame.store(Reg.result, procedure.savedPC, Reg.link) | |
| 10 | frame.store(Reg.result, procedure.paramPtr, Reg.savedParamPtr) | |
| 11 | ADD(Reg.framePointer, Reg.result, Reg.zero) | |
| **Line** | **Body** | **Reg.result value** |
| 12 | read(Reg.result, n) | |
| 13 | write(exprTempVar, Reg.result) | |
| 14 | LIS(Reg.result); Word(encodeUnsigned(1)) | |
| 15 | read(Reg.scratch, exprTempVar) | |
| 16 | ADD(Reg.result, Reg.scratch, Reg.result) | |
| **Line** | **Epilogue** | **Reg.result value** |
| 17 | frame.load(Reg.framePointer, Reg.link, procedure.savedPC) | |
| 18 | frame.load(Reg.framePointer, Reg.framePointer, procedure.dynamicLink) | |
| 19 | Stack.pop | |
| 20 | Stack.pop | |
| 21 | JR(Reg.link) | |

(b) On lines 3, 6, 8, 9, 10, 11, 13, and 16, the value of Reg.result is used. For each line, explain what could go wrong if the value of Reg.result was overwritten before the line was executed.

(c) What would go wrong if Reg.result was modified in the epilogue?

6. Consider the following set of nested procedures.

```
def f(t) {
    var x = 1
    var y = 2
    def g(z) {
        x += z
        y += z
        x + y
    }
    def h() {
        var a = 3
        var b = 4
        def i() {
            var c = g(b)
            x + a + c
        }
        def j() {
            var d = i()
            y + b + d
        }
        t + j()
    }
    h()
}
```

(a) What value does `f(5)` return?

(b) Identify which variable accesses in the program require following a static link more than once.

(c) Whenever a procedure is called, the caller computes the static link of the callee and passes it as a parameter to the caller. In some cases, the caller simply passes in its own frame pointer. In other cases, the caller follows static links and passes in the frame pointer of a different procedure. For each static link computation that occurs when `f` is called, identify which frame pointer is passed in, and how many static links the caller needs to follow to reach this frame pointer.

# 4 Register Allocation

1. Here is a program.

```
v_1 = 1

v_2 = v_1 + v_1

v_3 = v_2 + v_1

v_4 = v_3 + v_1

v_1 = v_4 - v_3

v_2 = v_3 - v_1

v_3 = v_2 + v_2 - v_1

v_4 = v_1 + v_1 + v_1 + v_1

v_2 = v_4 - v_2
```

(a) After each line, list which variables are live at that point in the program.

(b) Draw the interference graph for the variables. Using the graph, determine the minimal number of registers needed to allocate the variables.