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 4×8 pixel black and white image
   (j) A PDF file containing the Fall 2019 CS241E midterm with solutions

   Solution: All of these can be represented by single word, or even a single bit! Why is that? Because bits don't inherently have meaning. Humans assign meaning to bit sequences when designing computers, so that the computers can do useful operations on useful data. We define various conventions that allow us to easily work with things like numbers and strings, but these are just conventions – there is no law of the universe that tells us what bit sequences have to mean.

   This is why on assignments, you often see phrases like “interpret the value in register one as a two's complement integer” or “interpret the value in register one as the memory address of an array”. The values stored in registers are just bit sequences, and we have to assign meaning to them ourselves.

   For example, how could we represent the CS241E midterm with a single bit? Well, we could just decide arbitrarily that 1 means “the CS241E midterm” and 0 means “the regular CS241 midterm”. Then a 32-bit word could represent a sequence of midterms, where each is either a CS241E midterm or a CS241 midterm. We can then imagine a machine that has physical copies of each midterm, and given an input bit sequence, produces a sequence of photocopies of the midterms according to the bit sequence. This is pointless, but it shows that we could design a machine that uses a single bit to represent an entire midterm, if we really wanted to.

   We can apply the same principle to anything else on the list – a single bit can represent anything we want. In practice though, we choose conventions that are sensible and usable instead of making a single bit represent a midterm. If we represent a midterm as a large sequence of bits which encodes the data in an understandable way, it's easier to do things like writing programs that display the midterm or let us modify the midterm.

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.)

   Solution: The bit sequence is 1111. . . 1 (32 1s). As a base 10 number, this is $2^{32} - 1$.

   (b) What is the largest 32-bit signed (two’s complement) integer? Write it as a bit sequence and as a base 10 number.

   Solution: The bit sequence is 0111. . . 1 (0 followed by 31 1s). As a base 10 number, this is $2^{31} - 1$.

   (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.

   Solution: The bit sequence is 1000. . . 0 (1 followed by 31 0s). As a 32-bit signed integer, this is -2^{31} in base 10. As a 32-bit unsigned integer, this is 2^{31} 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.

   Solution: The bit sequence (with spaces for readability) is 1000 0000 0000 0000 0000 0000 0010 1010. As a 32-bit signed integer, this is $-2^{31} + 42$. The relationship is that they are both congruent modulo 2^{32}: we have $2^{31} + 42 \equiv 2^{31} + 42 - 2^{32} \equiv -2^{31} + 42 \pmod{2^{32}}$.

   (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.

   Solution: Consider dividing the following two bit sequences:
   0000 0000 0000 0000 0000 0000 0000 0001
   1111 1111 1111 1111 1111 1111 1111 1111
   If we interpret these as unsigned integers, we get $1/(2^{32} - 1) = 0$. The resulting bit sequence is
   0000 0000 0000 0000 0000 0000 0000 0000
   However, with signed integers we get $1/(-1) = -1$. The resulting bit sequence is
   1111 1111 1111 1111 1111 1111 1111 1111
   Therefore, we cannot design a single circuit that handles the signed and unsigned cases simultaneously.

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?

Solution: Here is what happens when the program counter reaches the BEQ instruction. First, the instruction is fetched. Then, the program counter is incremented. Now the program counter is pointing to the next instruction – the JR. Then the BEQ is executed. This increments the program counter by `X * 4`. Here is what happens in each case.

- If `X = -2`, we move back two instructions from the JR, and reach the second LIS. This LIS instruction, when executed, skips over the BEQ (after loading its binary representation into register 3) and the program counter ends up back at the JR. The program terminates normally.
- If `X = -1`, we move back one instruction from the JR, and execute the BEQ again on the next step. The program loops infinitely.
- If `X = 0`, the BEQ does not change the program counter at all. On the next step, we execute the JR and the program terminates normally.
- If `X = 1`, we move forward one instruction from the JR. The next instruction fetched is the invalid instruction -1, and the program crashes.
- If `X = 2`, we move forward two instructions from the JR. The next instruction fetched in the invalid instruction -1, and the program crashes.

The key to questions like this is to remember the order in which the CPU performs actions: fetch the instruction, increment the program counter, and then execute the fetched instruction. The program counter is incremented before the instruction gets 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.

   Solution: Here is such a program.

```
LIS(Reg(3))
Use(label)
Define(label)
```

   How does this work? When the assembler eliminates the labels, it produces metadata about the definitions and uses of labels. The metadata is computed under the assumption that the program will be loaded at address zero. The metadata looks something like this:

```
Use(label) at 4
Define(label) at 8
```

   After labels are eliminated and the code is relocated to some address Y, it will look like this:

```
Y+0 LIS(Reg(3))
Y+4 Word(encodeUnsigned(Y+8))
```

The offsets in the metadata are adjusted by the address Y. Then the use of the label, at address Y+4, is replaced by its value, which was adjusted from 8 to Y+8. Now let's trace through this program and make sure that register 3 really contains the same value as the program counter when it's over.

- The program counter starts at $Y + 0$. The LIS instruction is fetched.
- The program counter is incremented to $Y + 4$.
- The LIS instruction is executed. Register 3 gets the value in memory stored at address $Y+4$. This value is $Y + 8$. Then, the program counter is incremented again – as part of the LIS instruction itself, not the fetch-increment-execute cycle! The program counter is now $Y + 8$.

We see that at the end of this code, both register 3 and the program counter contain $Y+8$, as required.

(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.

Solution: The basic idea is to use JALR to store the program counter in register 31, then add this to some constant value in register 3 so that register 3 ends up being equal to the program counter.

The problem is that when we use JALR, we will jump to another memory address. If we jump into some random part of memory we don't have control over, we don't know what will happen. There are two areas of memory we have control over – our program itself, and the "reserved word".

Jumping to another part of our program is out of the question, since we don't have labels to identify specific lines of our program that we might want to jump to. So we use the reserved word.

Our strategy will be to write a JR(Reg(31)) instruction into the reserved word. Then when we jump to it, we'll immediately jump back to the next line of our program, but now register 31 will have a previous value of the program counter in it. So we just need to increment that program counter by the right constant and store it in register 3.

How do we write an instruction into memory?? Well, code is just data. An instruction is just a specific bit sequence. So we can load the instruction into a register with LIS, and write it into memory with SW, just like we would do with any other piece of data.

Here's the skeleton of our program. We assume it is loaded at some address Y.

```
Y+00 LIS(Reg(3))
Y+04 Word(encodeUnsigned(X))
Y+08 LIS(Reg(4))
Y+12 JR(Reg(31))
Y+16 SW(Reg(4),0,Reg(1))
Y+20 JALR(Reg(1))
Y+24 ADD(Reg(3),Reg(3),Reg(31))
```

The first five lines are just setup: we load a constant X (we'll figure out what X needs to be soon) into register 3, and we store a JR(Reg(31)) instruction in the reserved word. Let's trace through the program, starting from the point where the program counter is at $Y + 20$.

- The JALR(Reg(1)) instruction is fetched.
- PC is incremented to $Y + 24$.
- JALR(Reg(1)) is executed. Now Reg(31) $= Y + 24$, and the program counter is set to the address in Reg(1).
- The JR(Reg(31)) instruction, which we stored at the address in Reg(1), is fetched.
- PC is incremented (but it doesn't matter since JR is about to change it).
- JR(Reg(31)) is executed. Now the program counter is at $Y + 24$.
- The ADD instruction is fetched.
- PC is incremented to $Y + 28$.
- The ADD instruction is executed, and Reg(3) now contains $X + Y + 24$.

Since we want the program to end with Reg(3) and the program counter equal, it is clear that we must set X to 4. So our final program looks like this:

```
LIS(Reg(3))
Word(encodeUnsigned(4))
LIS(Reg(4))
JR(Reg(31))
SW(Reg(4),0,Reg(1))
JALR(Reg(1))
ADD(Reg(3),Reg(3),Reg(31))
```

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	

Solution: Let's go through line by line.

- The value at address 100 should be the first line of MAIN. We see that a label is defined at this first line, but no label is used. Since this value is not a use of a label, it is probably an important value (such as an instruction or constant) that should not be adjusted. So we copy the value from line 0 of MAIN into line 100 of the linked and relocated file.
- The next blank line is address 112. Since the program has been relocated to address 100, this corresponds to address 12 in MAIN. We see this line of MAIN is the use of the label "location". We also see that the label "location" is defined in MAIN, at address 28. We should relocate this label to address 128. So in the linked and relocated file, we put the value 128 at address 112.
- Next is address 120. This line is the use of a label "procedure". The label "procedure" is defined at address 0 in the PROCEDURE file. Because MAIN has been relocated to address 100, and MAIN is 28 words long, and PROCEDURE comes directly after MAIN, it follows that PROCEDURE has been relocated to address 132 ($128 + 4$). So the label definition of "procedure" is at address 132, which means we should put the value 132 at address 120 in the linked and relocated file.
- Next is address 128, which contains the use of a label "end". The label "end" is defined at address 16 in PROCEDURE. Since procedure has been relocated to address 132, the new value of end is 148 ($132 + 16$). So address 128 gets the value 148.
- Next is address 132. Here there is again a label definition, but no label use. So we should preserve the original machine code value of 2353201152.
- Next is address 144. We have a use of label "main", which is defined in MAIN at address 0. After relocation, the new address is 100.
- Finally, address 148 contains a definition of a label but not a use, so again we preserve the original machine code value of 65011720.

Here is the result.

Address	Machine Code
100	2068
104	65011720
108	4116
112	128
116	16404
120	132
124	16777225
128	148
132	2353201152
136	2889940992
140	6291464
144	100
148	65011720

(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?

Solution: First, let's focus on what the procedure does.

- Take the memory address in register 2, and load the value at that address into register 3.
- Take the value in register 1, and store it at the memory address in register 2.
- Jump to the address in register 3.

After this jump, in general, we don't know what will happen. But what about for this particular call of the procedure? In register 1 we have a JR(Reg(17)) instruction. In register 2 we have the value of the label "location", which is the address of a use of the label "end". So this procedure will:

- Load the value of label "end" into register 3.
- Overwrite the place containing the value of the label "end" with JR(Reg(17)).
- Jump to the location of the label "end", which happens to be a JR(Reg(31)) line that returns us from the procedure.

When we return to the procedure, we are on the line after JALR(Reg(8)). But what is in this line? During the procedure, we overwrote this line with JR(Reg(17)). So we execute JR(Reg(17)).

What does Reg(17) contain? Well, from the first line, we see it contains the original value of Reg(31), which starts off as the "termination constant". So JR(Reg(17)) terminates the program.

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 $\mathtt{n}_i$ for the i-th instance of variable `n`.

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

Solution:

Variable Kind	Instance Extent	Storage Location
Global variable	Entire execution of program	Fixed memory address
Procedure-local	Duration of procedure	Stack
Object field	From time of allocation to time of last use	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.

Solution:

- Before `main` is called, the extent of $\texttt{meow}_1$ begins.
- `main` is called. The extent of $\texttt{args}_1$ begins.
- Local variable `totalNoises` is defined. The extent of $\texttt{totalNoises}_1$ begins.
- `makeNoise(1,meow)` is called. The extents of $\texttt{m}_1$ and $\texttt{noise}_1$ begin.
- `printNoiseRepeatedly(1)` is called. The extent of $\texttt{n}_1$ begins.
- `printNoiseRepeatedly(1)` returns. The extent of $\texttt{n}_1$ ends.
- `makeNoise(1,meow)` returns. The extents of $\texttt{m}_1$ and $\texttt{noise}_1$ end.
- `makeNoise(2,"Woof")` is called. The extents of $\texttt{m}_2$ and $\texttt{noise}_2$ begin.
- `printNoiseRepeatedly(2)` is called. The extent of $\texttt{n}_2$ begins.
- `printNoiseRepeatedly(1)` is called. The extent of $\texttt{n}_3$ begins.
- `printNoiseRepeatedly(1)` returns. The extent of $\texttt{n}_3$ ends.
- `printNoiseRepeatedly(2)` returns. The extent of $\texttt{n}_2$ ends.
- `makeNoise(2,"Woof")` returns. The extents of $\texttt{m}_2$ and $\texttt{noise}_2$ end.
- `main` returns. The extents of $\texttt{args}_1$ and $\texttt{totalNoises}_1$ end. The program is finished, so the extent of $\texttt{meow}_1$ also ends.

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).

Solution: There are many possible solutions. The key is that your test case should modify the stack pointer so that it can no longer be used to access variables reliably. In the following test case, the compiled program will very likely crash when trying to read the value of the "oldStackPointer" variable, since it temporarily points to an invalid address.

```
val oldStackPointer = new Variable("old stack pointer")
val code = block(
    write(oldStackPointer,Reg.stackPointer),
    LIS(Reg.stackPointer),
    Word(encodeSigned(-1)),
    read(Reg.stackPointer,oldStackPointer)

)
```

You could argue this test case is mean-spirited because it temporarily messes up the stack pointer for no reason, instead of using the stack in a conventional way. However, a test case that pushed data onto the stack "normally" would still be very likely to produce incorrect results if a read or write is performed after changing the stack pointer.

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

Solution: Any program which changes the stack pointer and then accesses variables before changing it back will have incorrect behaviour. Even if the program happens to produce the correct result, variable values will be stored in a memory location that they were not intended to be stored in, and variable reads will obtain information from the wrong location. However, programs that do not modify the stack pointer will work perfectly!

(c) How can Sylvie fix her implementation?

Solution: Instead of accessing variables as offsets from the stack pointer, access them as offsets from the frame pointer. After allocating a chunk, save the current stack pointer in the frame pointer register. Use the frame pointer register as the basis for all accesses of variables within that chunk. Then the stack can be freely used, and we will always still be able to access variables with the frame pointer.

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`.

Solution: One example is $1 + (2 + 1)$, but there are many examples. The key is to find an expression such that computing rightExpr will overwrite the stored value of leftExpr with something different.

When evaluating this expression, Sylvie's `binOp` will evaluate leftExpr, which is 0, and store it in Reg.scratch. Then it will evaluate rightExpr, which is $2+1$. To evaluate this, we store 2 in Reg.scratch and 1 in Reg.result and add them to get 3. So now Reg.scratch contains 2, and Reg.result contains 3. The final result is then 5, which is incorrect.

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

Solution: It is easy to see that $1 + 1$ will be evaluated correctly.

The expression $(1 + 1) + 1$ will also be evaluated correctly. Why? Well, in this case leftExpr is $1 + 1$, which we know will be evaluated correctly. So, the correct value of leftExpr will be stored in Reg.scratch. Now, since rightExpr is just 1, it is stored in Reg.result without modifying Reg.scratch. So Reg.scratch and Reg.result will both have the correct values

Based on this idea, one can prove that in general, if an expression E is evaluated correctly, then $(E)+1$ will be evaluated correctly. So a third example is $((1 + 1) + 1) + 1$.

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.

Solution: The idea is to loop over the array with a while loop. Before the main body, load the current element of the array into the element variable. After the main body, store the value of the element variable back into the array, so that changes to the element are reflected in the array. Then increment the "current element" memory address to point to the next element of the array.

There are a few ways to check the termination condition for the for-each loop (i.e., to check when you've reached the end of the array). We will do it by computing the address that is one word past the end of the array, and we will stop the loop once the "current element" memory address reaches this end address.

```
def varia(v: Variable): Code = read(Reg.result, v)
def const(i: Int): Code = block(LIS(Reg.result),Word(encodeUnsigned(i)))
def forEachLoop(address: Code, size: Code, element: Variable, body: Code): Code = {
    val currentAddress = new Variable("currentAddress")
    val endAddress = new Variable("lastAddress")
    Scope(Seq(currentAddress, endAddress), block(
        assign(currentAddress, address),
        assign(endAddress, binOp(varia(currentAddress),plus,binOp(size,times,const(4)))),
        whileLoop(varia(currentAddress),neCmp,varia(endAddress), block(
            assign(element,deref(varia(currentAddress))),
            body,
            assignToAddr(varia(currentAddress),varia(element)),
            assign(currentAddress,binOp(varia(currentAddress),plus,const(4)))
        ))
    ))
}
```

Note that we do not put `element` in the Scope of the for-each loop. This is so an element variable can be used across multiple for-each loops without raising a duplicate variable error. If we included `element` in the Scope of the for-each loop, we would need to create a separate variable for each for-each loop, which is annoying. However, your solution is not incorrect if you put `element` in the for-each loop Scope, it's just an implementation choice.

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)	

Solution: The filled-out table is on the next page.

Line	Call	Reg.result value
0	assign(tempVars(0), arguments(0))	value of the argument
1	Stack.allocate(paramChunks(target))	parameter chunk address
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)	procedure frame address
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)	n = 41
13	write(exprTempVar, Reg.result)	
14	LIS(Reg.result); Word(encodeUnsigned(1))	1
15	read(Reg.scratch, exprTempVar)	
16	ADD(Reg.result, Reg.scratch, Reg.result)	n+1 = 42
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.

Solution:

- Line 3: Here Reg.result contains the address of the parameter chunk. If Reg.result was incorrect, the parameter values would be stored in the wrong memory location, possibly overwriting important data; or the program could even crash if Reg.result contained an invalid memory address.
- Line 6: Here Reg.result still contains the address of the parameter chunk. It is copied into Reg.savedParamPtr, which is used to store the parameter chunk pointer on the procedure's frame. If Reg.result was incorrect, the procedure would have an incorrect parameter chunk pointer, and would potentially obtain the wrong values or even crash when trying to access its parameters.
- Lines 8, 9, 10: Here Reg.result contains the address of the procedure's frame. This address is being used to store the dynamic link (address of caller's frame), the saved program counter (return address) and the parameter chunk pointer. If Reg.result was incorrect, these values would be stored in the wrong location, possibly overwriting important data; or the program could crash if Reg.result contained an invalid memory address.
- Line 11: Here Reg.result contains the address of the procedure's frame. This address is being copied into the frame pointer register. The frame pointer register is used as a base register for all variable accesses in the procedure. If Reg.result was incorrect, variable accesses would return incorrect data or cause crashes. Additionally, the frame pointer is used to restore important data at the end of the procedure: the dynamic link (address of caller's frame) and the saved program counter (return address). If the frame pointer is incorrect, the caller might not be able to access its frame properly after the callee returns, and the callee might return to the wrong location.
- Lines 13 and 16: On these lines, Reg.result contains an intermediate value for an expression. If Reg.result was incorrect, the value of the expression might be incorrect.

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

Solution: By our conventions, Reg.result stores the return value of the procedure. If it was modified in the epilogue, then the caller may receive an incorrect return value.

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?

Solution: Let us trace the program.

- `f(5)` calls `h()` and returns its value. We have `t = 5`.
- `h()` returns `t + j() = 5 + j()`. So we need the return value of `j()`.
- `j()` calls `i()`. The value of `d` is the return value of `i()`.
- `i()` calls `g(b)`. The variable `b` is a local variable of `h` equal to `4`. The value of `c` is the value of `g(4)`.
- `g(4)` adds `4` to `x` and `y`, which are local variables of `f`, initialized to `1` and `2` respectively. We have `x = 5` and `y = 6`. Then `g(4)` returns `11`.
- Back to `i`, we return `x + a + c`. We know `x = 5` and `c = g(4) = 11`. The variable `a` is a local variable of `h` initialized to `3`. We return `5 + 3 + 11 = 19`.
- Back to `j`, we return `y + b + d`. We have `y = 6`, `b = 4`, and `d = 19`. We return `29`.
- Back to `h`, we return `t + j() = 5 + 29 = 34`.
- Back to `f`, we return `34`.

So the return value of `f(5)` is `34`.

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

Solution: The number of times we need to follow a static link to access a variable is equal to the number of times we need to ascend to an outer procedure to find the variable. So the only way we will ever need to follow a static link more than once is if we are in a procedure of nesting level 2 or deeper. This means `i` and `j` are the only relevant procedures.

In `i` we access `x`, `a` and `c`. Since `c` is a local variable, we follow no static links. Since `a` is one nesting level up, we follow a static link only once. But for `x`, we need to follow a static link twice to reach a frame of `f`.

Similarly, in `j`, we need to follow a static link twice to access `y`.

There are no other cases in this set of procedures where we need to follow a static link more than once to access a variable.

(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.

Solution: The frame pointer that is passed in is always the frame pointer of the direct outer procedure of the callee. We look at each call in the chain of procedure calls.

- `f` calls `h`: here `f` passes its own frame pointer.
- `h` calls `j`: here `h` passes its own frame pointer.
- `j` calls `i`: here `j` passes in the frame pointer of `h`, which it obtains by following one static link.
- `i` calls `g`: here `i` passes in the frame pointer of `f`, which is obtains by following two static links (one to reach `h`, and another to reach `f`).

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.

Solution: We proceed by working backwards. The idea is that whenever we read a variable, it must have been live before the line executes. Whenever we write to a variable without reading it on the same line, this means it is not live before the line executes, because whatever value it had before this line is never read.

First, note that at the very end of the program, no variables are live because nothing will ever be read after the program ends.

Now, here is the last line:

```
v_2 = v_4 - v_2
```

We read v_4 and v_2 on this line, so they must be live before this line executes. Next:

```
v_4 = v_1 + v_1 + v_1 + v_1
Live after this line: {v_2, v_4}
```

We read v_1, so it must be live before this line executes. But also, we write to v_4, and we do not read from it, so v_4 is not live before this line. Since v_2 was not read or written to, its live status does not change.

```
v_3 = v_2 + v_2 - v_1
Live after this line: {v_1, v_2}
```

In this line, we read variables that are known to be live after this line, so there are no changes here.

```
v_2 = v_3 - v_1
Live after this line: {v_1, v_2}
```

Here we read v_1 and v_3, so they are live before this line. We write to v_2 without reading it, so it is not live before this line.

```
v_1 = v_4 - v_3
Live after this line: {v_1, v_3}
```

We read v_3 and v_4, so they are live before this line. We write to v_1 without reading it, so it is not live before this line.

```
v_4 = v_3 + v_1
Live after this line: {v_3, v_4}
```

We read v_1 and v_3, so they are live before this line. We write to v_4 without reading it, so it is not live before this line.

```
v_3 = v_2 + v_1
Live after this line: {v_1, v_3}
```

We read v_1 and v_2, so they are live before this line. We write to v_3 without reading it, so it is not live before this line.

```
v_2 = v_1 + v_1
Live after this line: {v_1, v_2}
```

We read v_1, so it is live before this line. We write to v_2 without reading it, so it is not live before this line.

```
v_1 = 1
Live after this line: {v_1}
```

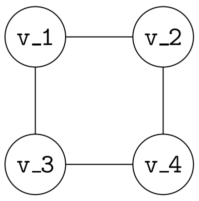
Now we've worked backwards to the beginning, so we can write things out forward to produce the solution:

```
v_1 = 1
{v_1}
v_2 = v_1 + v_1
{v_1, v_2}
v_3 = v_2 + v_1
{v_1, v_3}
v_4 = v_3 + v_1
{v_3, v_4}
v_1 = v_4 - v_3
{v_1, v_3}
v_2 = v_3 - v_1
{v_1, v_2}
v_3 = v_2 + v_2 - v_1
{v_1, v_2}
v_4 = v_1 + v_1 + v_1 + v_1
{v_2, v_4}
v_2 = v_4 - v_2
{}
```

Reading this program forwards, do the points at which various variables start being live and stop being live make sense to you?

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

Solution: Here is the interference graph.



The interference graph can be 2-coloured, with `v_1` and `v_4` receiving one colour, and `v_2` and `v_3` receiving the other. This means we can allocate these variables with two registers. It is not possible to 1-colour the graph, so we cannot allocate the variables with one register.

Side note (probably not required knowledge for the midterm): If you try to actually use this register allocation scheme, you will notice something strange. Here is a trace of the program with all variables replaced with their respective registers.

```
r_1 = 1                          // r_1 (v_1) = 1
r_2 = r_1 + r_1                  // r_2 (v_2) = 1 + 1 = 2
r_2 = r_2 + r_1                  // r_2 (v_3) = 2 + 1 = 3
r_1 = r_2 + r_1                  // r_1 (v_4) = 3 + 1 = 4
r_1 = r_1 - r_2                  // r_1 (v_1) = 4 - 3 = 1
r_2 = r_2 - r_1                  // r_2 (v_2) = 3 - 1 = 2
r_2 = r_2 + r_2 - r_1            // r_2 (v_3) = 2 + 2 - 1 = 3
r_1 = r_1 + r_1 + r_1 + r_1      // r_1 (v_4) = 1 + 1 + 1 + 1 = 4
r_2 = r_1 - r_2                  // r_2 (v_2) = 4 - 3 = 1
```

Notice that value on the final line is incorrect! The value of `v_2`, which is assigned to register `r_2`, got overwritten with the value of `v_3`, which is also assigned to `r_2`. This doesn't happen with the variable version of the program. Does this mean our register allocation algorithm is wrong?

It turns out the allocation algorithm is fine, but for some programs we need to do an extra optimization to ensure correctness. Notice that the final value of `v_3` that we overwrite `r_2` with is never used. We can tell it is never used because if it was used, then `v_3` would be live immediately after the line `v_3 = v_2 + v_2 - v_1`, but it isn't. When we encounter a case like this – where we write to a variable, but the variable does not become live on the next line – we should throw away the result of the right-hand-side expression, rather than storing it in a register. Otherwise, storing this unused value in a register could overwrite a value that will actually be used.

If we make this optimization, and throw away the unused value of `v_2`, the program will work correctly.

```
r_1 = 1                         // r_1 (v_1) = 1
r_2 = r_1 + r_1                 // r_2 (v_2) = 1 + 1 = 2
r_2 = r_2 + r_1                 // r_2 (v_3) = 2 + 1 = 3
r_1 = r_2 + r_1                 // r_1 (v_4) = 3 + 1 = 4
r_1 = r_1 - r_2                 // r_1 (v_1) = 4 - 3 = 1
r_2 = r_2 - r_1                 // r_2 (v_2) = 3 - 1 = 2
r_2 = r_2 + r_2 - r_1           // (result thrown away)
r_1 = r_1 + r_1 + r_1 + r_1     // r_1 (v_4) = 1 + 1 + 1 + 1 = 4
r_2 = r_1 - r_2                 // r_2 (v_2) = 4 - 2 = 2
```