Names: Aaron Wang and Ethan Little
CSE 20221 Logic Design
HW03: More albaCore Programming

Preamble

1. Make sure you start from the Google Docs copy of this assignment, found in Google Drive.

2. Copy this assignment to your Google Drive folder and enter your answers in the boxes provided; **each empty box should be filled in with an answer**. You can either type your answers directly or scan handwritten work as long as it is legible. For solutions that require code, use a fixed-width font (Consolas preferred) no smaller than 10 points with proper indentation. Save your solutions as a single PDF file and upload them to Gradescope.

3. See the checklist at the end of the document for a list of what to turn in, and point totals.

4. Type your name(s) at the top of this document - you are encouraged to work in groups of 2.

5. Always double-check your final PDF prior to uploading to Gradescope. In past semesters, students occasionally encountered problems copying output from a terminal to a Google Doc.

6. When copying text or taking a screenshot, please use dark text on a light background, or light text on a dark background, otherwise it is very difficult to read.

Obtaining Skeleton Files

Several problems in this assignment refer to sample programs that can be accessed through the CSE student machines at:

```
~/esc-courses/fa24-cse-20221.01/public/hw_sample/hw03
```

This contains folders, with some reference and sample code for the problems:

```
p1  p2  p3  p4  p6
```

Here is the full directory structure:

```
hw03
├── p1
│   ├── p1_global_scalar.c
│   └── p1_global_scalar.s
├── p2
│   ├── p2_global_array.c
│   └── p2_global_array.s
├── p3
│   └── p3_branch.s
├── p4
│   └── p4_fun.s
└── p6
    ├── p6_mult.c
    ├── p6_mult.s
    └── p6_mult_while.c
```

Copy the relevant files to your own local directory and work on them there.

Problem 1

Compilers translate programs written in a high-level language such as C to the assembly language of a microprocessor. As an intermediate step to understanding this, we can rewrite high-level C statements as lower-level statements that more closely resemble assembly language (as we did in class). In this problem, we will explore how reading a global variable in a C program translates to low level pointer operations, that in turn correspond to the use of labeled addresses in the data segment and load instructions in albaCore assembly language.

The simple operation of reading a global scalar variable in “high-level” C is as follows:

```
int X = 0xFADE;
int main()
{
    int v = X;
}
```

This can be rewritten as the low-level C program `p1_global_scalar.c` using pointers that is closer to assembly language:

```
#include <stdio.h>
```

```
int X = 0xFADE;
int main()
{
    int* X_ptr = &X;
    int  v = *X_ptr;
}
```

Note: the C code in our course directory has 2 print statements for demonstration purposes; we will not translate this to albaCore.
Next, we can translate the low-level C to assembly language, with the C as comments, where global variables are labeled addresses with initial values in the data segment, and local variables are registers (with the variable names modified to indicate registers used). This may be found in `p1_global_scalar.s`

```
.data
// int X = 0xFADE;
X: 0xFADE

.text
main:               // int main()
                    // {
ldi r1, high(X)     //     int* r1_X_ptr = &X;
ldi r0, 8
shl r1, r1, r0
ldi r0, low(X)
or  r1, r1, r0
ld  r2, r1, 0       //     int  r2_v = *X_ptr;
quit                // }
```

Assemble the albaCore program `p1_global_scalar.s`, simulate it and answer the following questions.

Copy your assembled .mem file contents here

```
// .text
@0000 7100 // ldi r1, 0    label: main    target label: X
@0001 7008 // ldi r0, 8
@0002 5110 // shl r1, r1, r0
@0003 7007 // ldi r0, 7    target label: X
@0004 3110 // or r1, r1, r0
@0005 8201 // ld r2, r1, 0
@0006 f000 // sys 0

// .data
@0007 fade //     label: X
```

Copy your simulation output from albasimh here

```
0000: 7100  ldi r1, 0                    | r1 = 0x0 (0)
0001: 7008  ldi r0, 8                    | r0 = 0x8 (8)
0002: 5110  shl r1, r1, r0               | r1 = 0x0 (0)
0003: 7007  ldi r0, 7                    | r0 = 0x7 (7)
0004: 3110  or r1, r1, r0                | r1 = 0x7 (7)
```

```
0005: 8201  ld r2, r1, 0                | r2 = 0xfade (-1314)
0006: f000  sys, 0
```

At what memory address was X placed? Write your answer in hexadecimal, including any leading 0s. Recall addresses in albaCore are 16 bits.	0x0007
What are the values of high(X) and low(X) Write your answer in hexadecimal, including any leading 0s.	high(X) = 0x00 low(X) = 0x07

Now, modify your program so it looks like the following:

```
.data
// int X = 0xFADE;
X: 0xFADE

.text
main:               // int main()
                    // {
ldi r1, X           //     int* r1_X_ptr = &X;
ld  r2, r1, 0       //     int  r2_v = *X_ptr;
quit                // }
```

Assemble the new albaCore program, simulate it and answer the following questions.

Copy your assembled .mem file contents here

```
// .text
@0000 7103 // ldi r1, 3    label: main    target label: X
@0001 8201 // ld r2, r1, 0
@0002 f000 // sys 0

// .data
@0003 fade //     label: X
```

Copy your simulation output from albasimh here

```
0000: 7103  ldi r1, 3                   | r1 = 0x3 (3)
0001: 8201  ld r2, r1, 0                | r2 = 0xfade (-1314)
0002: f000  sys, 0
```

How did this program behave compared to the original version with high(X) and low(X)?	This program automatically takes the pointer of X and puts it into r1. It practically only loads in the low part. Luckily this works because there is not code for there to need a high part.
What is the address of X in this version of the	0x0003

program? Write your answer in hexadecimal, including any leading 0s.	
This version of the program, while concise, may not work in many circumstances. Why is that the case? Hint: think about the maximum immediate for an ldi instruction as compared to what range of memory addresses X could take. **Going forward, we will always use the high(X)/low(X) method.**	As indicated above, this program ignores the high part of the number. Although it works in this scenario, it would break if X is stored at an address that is greater than 0xff(8 bits).

Finally, modify your code to add a second variable, Z in the .data section with the value 0x55. You may use the shortcut method (i.e., a single ldi instruction) to get the address of Z. Add code to load Z from memory, increment it by 2, and place it into the memory location after Z. Do *not* create a third named variable in the .data section. Assemble the new albaCore program, simulate it and answer the following questions.

Copy your assembled .mem file contents here

```
// .text
@0000 7108 // ldi r1, 8    label: main    target label: X
@0001 8201 // ld r2, r1, 0
@0002 7309 // ldi r3, 9    target label: Z
@0003 8403 // ld r4, r3, 0
@0004 7502 // ldi r5, 2
@0005 0445 // add r4, r4, r5
@0006 9143 // st r4, r3, 1
@0007 f000 // sys 0

// .data
@0008 fade //     label: X
@0009 0055 //  label: Z
```

Copy your simulation output from albasimh here

```
0000: 7108  ldi r1, 8                    | r1 = 0x8 (8)
0001: 8201  ld r2, r1, 0                 | r2 = 0xfade (-1314)
0002: 7309  ldi r3, 9                    | r3 = 0x9 (9)
0003: 8403  ld r4, r3, 0                 | r4 = 0x55 (85)
0004: 7502  ldi r5, 2                    | r5 = 0x2 (2)
0005: 0445  add r4, r4, r5               | r4 = 0x57 (87)
0006: 9143  st r4, r3, 1                 | M[000a] = 0x0057 (87)
0007: f000  sys, 0
```

Does the newly-created *value* (Z + 2) exist in memory in the .mem file? In other words, does the assembler set this up in memory? If so, how do you know? If not, when/how is this value	The newly created value does not exist in the mem file. Instead it is stored in the next address after the data. I know

placed in memory, and how do you know?	that this value is placed in the next memory location because that is what the program tells the computer to do: add 2, and store in address of &Z+1

Problem 2
This problem extends the previous problem to reading from a global array. The basic operation of reading from a global array variable with a local variable index in high-level C is as follows:

```
int A[4] = {0x3339, 0xB128, 0x74A9, 0x227C};
int main()
{
    int  i = 2;
    int  d = A[i];
}
```

This may be translated to low-level C using pointer operations that resemble assembly code as follows:

```
int A[4] = {0x3339, 0xB128, 0x74A9, 0x227C};
int main()
{
    int* A_ptr = A;             // starting address of A
    int  i = 2;                 // array index variable
    int* Ai_ptr = A_ptr + i;    // same as &(A[i])
    int  d = *Ai_ptr;           // same as A[i]
}
```

Translate this low-level C program into albaCore assembly language following the example in the previous problem, using the C as comments to guide the development of the assembly language. Use the provided file `p2_global_array.s`, which has already included the C program as comments. Note that for this problem, you cannot use constant offsets with the load instruction to index into the array, since the index is a variable.

albaCore assembly language program

```
.data
// int A[4] = {0x3339, 0xB128, 0x74A9, 0x227C};
A: 0x3339, 0xB128, 0x74A9, 0x227C

.text
main:           // int main()
                // {
ldi r1, A       //     int* r1_A_ptr = A;
ldi r2, 2       //     int  r2_i = 2;
add r3, r1, r2  //     int* r3_Ai_ptr = r1_A_ptr + r2_i;
ld  r4, r3, 0   //     int  r4_d = *r3_Ai_ptr;
quit            // }
```

Assembled .mem file

```
// .text
@0000 7105 // ldi r1, 5    label: main    target label: A
```

```
@0001 7202 // ldi r2, 2
@0002 0312 // add r3, r1, r2
@0003 8403 // ld r4, r3, 0
@0004 f000 // sys 0

// .data
@0005 3339 //     label: A
@0006 b128 //
@0007 74a9 //
@0008 227c //
```

Simulation output from albasimh

```
0000: 7105  ldi r1, 5                    | r1 = 0x5 (5)
0001: 7202  ldi r2, 2                    | r2 = 0x2 (2)
0002: 0312  add r3, r1, r2               | r3 = 0x7 (7)
0003: 8403  ld r4, r3, 0                 | r4 = 0x74a9 (29865)
0004: f000  sys, 0
```

Problem 3

In an albaCore assembly language program, we can either calculate the displacements for branches manually and specify them as part of the branch instructions (i.e., in the immediate field), or we can let the assembler figure this out for us (much easier!) using labels. The program `p3_branch.s` (found in the course directory listed at the top of the assignment) illustrates the use of both approaches:

```
.text
    ldi r5, 0
L1: bn  r5, 2
    bz  r5, L2
    quit
L2: not r5, r5
    br L1
```

The assembler is smart enough to know the difference between an integer displacement and a label and deals with them differently. If an integer displacement is specified, it uses it directly in generating the machine code for the branch instruction. If a label is specified, it calculates the displacement from the label address. Assemble and simulate the program above and answer the questions below.

Terminology reminder: a branch is "taken" if it adjusts the program counter (`PC`) to `PC + immediate` because the condition the branch is testing is true. If the condition is not true, the branch is considered "not taken" and the instruction immediately following the branch is executed (i.e., `PC + 1`).

Memory image file (.mem) generated by albaasmh

```
// .text
@0000 7500 // ldi r5, 0
@0001 c025 // bn r5, 2    label: L1
@0002 b025 // bz r5, 2    target label: L2
```

```
@0003 f000 // sys 0
@0004 4550 // not r5, r5, (null)    label: L2
@0005 afc0 // br -4    target label: L1

// .data
```

albasimh simulation output

```
0000: 7500  ldi r5, 0                    | r5 = 0x0 (0)
0001: c025  bn r5, 2
0002: b025  bz r5, 2
0004: 4550  not r5, r5                   | r5 = 0xffff (-1)
0005: afc0  br -4
0001: c025  bn r5, 2
0003: f000  sys, 0
```

What are the memory addresses for each label (L1, L2)? Find this by inspecting the .mem file.	`L1: 0x0001` `L2: 0x0004`

What is the meaning of each of the hex digits of the value at memory address `0x0001` in the memory image file?	`Value at memory address 0x0001: 0xc025`
	`C: bn; branch if input (rb) is negative`
	`02: jump directions by 02 if condition is satisfied`
	`5: r5 is input(rb)`

What is the displacement in hex for the `bz` (branch-if-zero) instruction in the memory image file? How was it determined?	`The bz instruction was translated into 0xb025. This means that the displacement for this was '02.' THis was determined by the compiler who knew that the L2 label was 2 lines below the bz instruction.`
What is the displacement in hex for the `br` (unconditional branch) instruction in the memory image file? How was it determined?	`The br command is stored as 0xafc0. Thus we know that the displacement is 0xfc = -4 by twos complement notation.`
Is the `bn` (branch-if-negative) branch taken the first time the instruction is executed in the simulator? Why or why not?	`No it is not because the input r5 is 0 which is not negative.`
Is the `bn` (branch-if-negative) branch taken the second time the instruction is executed in the simulator? Why or why not?	`Yes it is because the input r5 is -1 which is negative.`

Problem 4

In a high-level language such as C, the same function may be called from anywhere in the program, and regardless of where they are called from, they must return back to the point of the call after the function is done executing. This means that the function call mechanism must somehow save the return address. albaCore uses two instructions to support functions: the "jump and link" instruction `jal` to call

functions and the “jump register” instruction `jr` to return from them. The jump and link (`jal`) instruction does two things: it changes the program counter (`PC`) to the starting address of the function, and it saves a copy of the return address `PC+1` in register `r15`. The jump register (`jr`) instruction changes the `PC` to the value in a specified register, which should be `r15` to return from a function called using `jal`.

The program `p4_fun.s` calls two functions `fun1` and `fun2`. The calling convention is that input arguments are passed to the function in registers `r1` through `r4` and that return values are placed in `r0`.

```
.text
        ldi r1, 1
        ldi r2, 2
        ldi r3, 7
        jal fun1
        and r1, r0, r0
        jal fun2
        quit

fun1:   add r0, r1, r2
        add r0, r0, r3
        jr r15

fun2:   add r0, r1, r1
        jr r15
```

Assemble and simulate the program, then answer the questions below.

Memory image file (.mem) generated by albaasmh

```
// .text
@0000 7101 // ldi r1, 1
@0001 7202 // ldi r2, 2
@0002 7307 // ldi r3, 7
@0003 d007 // jal 0x7    target label: fun1
@0004 2100 // and r1, r0, r0
@0005 d00a // jal 0xA    target label: fun2
@0006 f000 // sys 0
@0007 0012 // add r0, r1, r2    label: fun1
@0008 0003 // add r0, r0, r3
@0009 e0f0 // jr r15
@000a 0011 // add r0, r1, r1    label: fun2
@000b e0f0 // jr r15

// .data
```

albasimh simulation output

```
0000: 7101  ldi r1, 1                    | r1 = 0x1 (1)
0001: 7202  ldi r2, 2                    | r2 = 0x2 (2)
0002: 7307  ldi r3, 7                    | r3 = 0x7 (7)
0003: d007  jal, 7                       | r15 = 0x4 (4)
```

```
0007: 0012  add r0, r1, r2                | r0 = 0x3 (3)
0008: 0003  add r0, r0, r3                | r0 = 0xa (10)
0009: e0f0  jr, r15
0004: 2100  and r1, r0, r0                | r1 = 0xa (10)
0005: d00a  jal, 10                       | r15 = 0x6 (6)
000a: 0011  add r0, r1, r1                | r0 = 0x14 (20)
000b: e0f0  jr, r15
0006: f000  sys, 0
```

At what memory addresses (found in the .mem file) did the assembler place `fun1` and `fun2`?	`Fun1: 0x0007` `Fun2: 0x000a`
At what memory address is the `jal fun1` instruction located?	`0x0003`

What is the meaning of each of the hex digits of the value in memory at address `0x0005`?	`0x0003: 0xd007`
	`D: jal; jump and link`
	`007: this is the address of where to jump and link too.`

What is the value of `r15` (from the simulator) when the first instruction of `fun1` is executed?	`0x4 because it puts it at pc+1 and pc is 0x3 when it is called`
At what memory address is the `jal fun2` instruction located?	`0x0005`
What is the value of `r15` when the first instruction of `fun2` is executed?	`0x0006; same reasoning as above`

Problem 5

Disassemble the following albaCore machine code instructions into an assembly language instruction. Check your solutions by assembling a program with the instructions - no need to simulate anything, the program doesn't do anything interesting. Assume these instructions are in memory at address 0, 1, 2, 3, and 4, respectively. Express any immediates in decimal - make sure to choose the appropriate interpretation of the immediate (signed vs. unsigned).

Machine Code (hex)	albaCore Assembly Language Instruction
EA3A	`jr r3`
D418	`jal 1048`
998F	`st r8, r15, 9`
A924	`br -110`
832B	`ld r3, r10, 2`

albaCore Program

```
.text
jr r3
jal 1048
st r8, r15, 9
br -110
ld r3, r11, 2
```

Memory image file (.mem) generated by albaasmh

```
// .text
@0000 e030 // jr r3
@0001 d418 // jal 0x418
@0002 998f // st r8, r15, 9
@0003 a920 // br -110
@0004 832b // ld r3, r11, 2

// .data
```

Problem 6

Unlike a typical commercial microprocessor such as those made by Intel, AMD, ARM, etc., albaCore does not have a native multiply instruction. Instead, we can write a function to multiply numbers using the shift-and-add method that is typically taught in elementary school for decimal numbers. For this problem, you will hand compile an albaCore assembly language function to multiply two (unsigned) 4-bit numbers from an original C program.

The shift-and-add method emulates the way multi-digit multiplication is done by hand.

```
    1001   multiplicand
  x 1011   multiplier
   -----
    1001   partial products
   1001
  0000
 1001
 -------
 1100011   product
```

Binary multiplication is much simpler than decimal multiplication: each partial product is either a shifted version of the multiplicand or else it is 0. A summary of the algorithm is as follows:

Initialize the product to 0
For each bit of the multiplier from least to most significant
    If the bit is a 1, add the multiplicand to the product
    Shift the multiplicand left by 1 bit

To test if a bit in a given position of the multiplier is a 1, we can perform an AND operation with a mask that has one bit set to a 1 and the others set to 0. For example, to determine if the 2nd-to-least-significant bit of the multiplier is a 1, you would AND the multiplier with the mask 0010. If the result is non-zero, then the bit in the multiplier must have been a 1. To test each bit of the multiplier in turn, shift

the mask with each iteration. Here is a C implementation, using a `while` loop, which you can find in `p6_mult_while.c` (in the course directory listed at the top of the assignment):

```
#include <stdio.h>
int mult(int m, int n);

int A[] = {9, 11, 0};

int main()
{
    int m, n, prod;

    m = A[0]; n = A[1];
    prod = mult(m, n);
    A[2] = prod;
    printf("%d (0x%x) x %d (0x%x) = %d (0x%x)\n", m, m, n, n, A[2], A[2]);
}

int mult(int m, int n)
{
    int prod = 0;
    int i = 4;
    int mask = 1;
    int nbit;

    while (i != 0) {
        nbit = mask & n;
        if (nbit != 0)
            prod = prod + m;
        //fill each table row BEFORE going on to next line
        m = m << 1;
        mask = mask << 1;
        i = i - 1;
    }
    return prod;
}
```

As a first step to make sure that you understand the algorithm and to obtain results for checking your albaCore implementation, trace through the C program that multiplies 9 x 11 (decimal) by hand and write down the values for `m`, `mask`, `nbit`, and `prod` at each iteration *before* updating values with shifts and subtraction (see the **bolded comment** in the code). Fill each table with decimal **and** binary values.

i	m	mask	nbit	prod
4	9 (1001)	1 (0001)	1 (0001)	9 (1001)
3	18 (10010)	2 (0010)	2 (0010)	27 (11011)
2	36 (100100)	4 (0100)	0 (0000)	27 (11011)
1	72 (1001000)	8 (1000)	8 (1000)	99 (1100011)

Note that you can check your results by adding `printf` statements to the program.

The first step in hand-compiling the original high-level C program is to rewrite it as a low-level C program that more closely resembles assembly language, replacing loops and conditional statements with `goto` statements and labels. The below program `p6_mult.c` does this for you (found in the course directory listed at the top of the assignment). The input and output values for the multiplication are stored in a global array `A[]` that will be placed in the data segment in memory. All local variables in both `main` and `mult` are assumed to be placed in registers, where the variable names in the low-level C program indicate which registers to use. The variable declarations in the rewritten C program are thus not directly translated to albaCore code, they just indicate what registers are associated with each variable. Registers `r1` and `r2` are used to pass the input values to `mult` and register `r0` is used to pass the return value back to `main`. Examine the program, compile it, and run it to make sure that you understand how it works.

```
#include <stdio.h>
#include <stdlib.h>

int mult(int, int);

int A[] = {9, 11, 0};

int main()
{
    int  r0_prod;
    int  r1_m;
    int  r2_n;
    int* r3_A;

    r3_A = A;
    r1_m = r3_A[0];
    r2_n = r3_A[1];
    r0_prod = mult(r1_m, r2_n);
    r3_A[2] = r0_prod;
    exit;
}

int mult(int r1_m, int r2_n)
{
    int r0_prod;
    int r4_i;
    int r5_mask;
    int r6_nbit;
    int r7_1;

    r0_prod = 0;
    r4_i = 4;
```

```
    r5_mask = 1;
    r7_1 = 1;

loop:
    if (r4_i == 0) goto finish;
    r6_nbit = r5_mask & r2_n;
    if (r6_nbit == 0) goto shift;
    r0_prod = r0_prod + r1_m;

shift:
    r1_m = r1_m << r7_1;
    r5_mask = r5_mask << r7_1;
    r4_i = r4_i - r7_1;
    goto loop;

finish:
    return r0_prod;
}
```

The file `p6_mult.s` provides a template for translating the C program into albaCore assembly language, where each of the lines of the C program serves as comments for 1 or more corresponding assembly language instructions. Complete the program `p6_mult.s`, assemble and simulate it, and then answer the questions below.

When writing and debugging an assembly language program of this complexity, it helps greatly to write and test the program in pieces. For example, you might start just by reading the values from array `A[]` and quitting. Once that is working, you can add a dummy version of `mult` that just returns a constant to get the function call and return working. Then you can complete the details inside of `mult`. This incremental approach is generally the fastest approach to success in any coding assignment.

albaCore program

```
.data
A: 9, 11, 0         // int A[] = {9, 11, 0};

.text
main:               // int main()
                    // {
                    //   int r0_prod;
                    //   int r1_m;
                    //   int r2_n;
                    //   int *r3_A;
ldi r3, A           //   r3_A = A;
ld r1, r3, 0        //   r1_m = r3_A[0];
ld r2, r3, 1        //   r2_n = r3_A[1];
jal mult            //   r0_prod = mult(r1_m, r2_n);
st r0, r3, 2        //   r3_A[2] = r0_prod;
quit                //   exit;
```

```
                    // }

mult:               // int mult(int r1_m, int r2_n)
                    // {
                    //   int r0_prod;
                    //   int r4_i;
                    //   int r5_mask;
                    //   int r6_nbit;
                    //   int r7_1;
ldi r0, 0           //   r0_prod = 0;
ldi r4, 4           //   r4_i = 4;
ldi r5, 1           //   r5_mask = 1;
ldi r7, 1           //   r7_1 = 1;

loop:               // loop:
bz r4, finish       //   if (r4_i == 0) goto finish;
and r6, r5, r2      //   r6_nbit = r5_mask & r2_n;
bz r6, shift        //   if (r6_nbit == 0) goto shift;
add r0, r0, r1      //   r0_prod = r0_prod + r1_m;

shift:              // shift:
shl r1, r1, r7      //   r1_m = r1_m << r7_1;
shl r5, r5, r7      //   r5_mask = r5_mask << r7_1;
sub r4, r4, r7      //   r4_i = r4_i - r7_1;
br loop               //   goto loop;

finish:             // finish:
jr r15              //   return r0_prod;
                    // }
```

.mem file produced by albaasmh

```
// .text
@0000 7313 // ldi r3, 19    label: main    target label: A
@0001 8103 // ld r1, r3, 0
@0002 8213 // ld r2, r3, 1
@0003 d006 // jal 0x6    target label: mult
@0004 9203 // st r0, r3, 2
@0005 f000 // sys 0
@0006 7000 // ldi r0, 0    label: mult
@0007 7404 // ldi r4, 4
@0008 7501 // ldi r5, 1
@0009 7701 // ldi r7, 1
@000a b084 // bz r4, 8    label: loop    target label: finish
@000b 2652 // and r6, r5, r2
@000c b026 // bz r6, 2    target label: shift
@000d 0001 // add r0, r0, r1
@000e 5117 // shl r1, r1, r7    label: shift
@000f 5557 // shl r5, r5, r7
@0010 1447 // sub r4, r4, r7
@0011 af90 // br -7    target label: loop
@0012 e0f0 // jr r15    label: finish
```

```
// .data
@0013 0009 //  label: A
@0014 000b //
@0015 0000 //
```

albasimh simulation output

```
0000: 7313  ldi r3, 19                  | r3 = 0x13 (19)
0001: 8103  ld r1, r3, 0                | r1 = 0x9 (9)
0002: 8213  ld r2, r3, 1                | r2 = 0xb (11)
0003: d006  jal, 6                      | r15 = 0x4 (4)
0006: 7000  ldi r0, 0                   | r0 = 0x0 (0)
0007: 7404  ldi r4, 4                   | r4 = 0x4 (4)
0008: 7501  ldi r5, 1                   | r5 = 0x1 (1)
0009: 7701  ldi r7, 1                   | r7 = 0x1 (1)
000a: b084  bz r4, 8
000b: 2652  and r6, r5, r2              | r6 = 0x1 (1)
000c: b026  bz r6, 2
000d: 0001  add r0, r0, r1              | r0 = 0x9 (9)
000e: 5117  shl r1, r1, r7              | r1 = 0x12 (18)
000f: 5557  shl r5, r5, r7              | r5 = 0x2 (2)
0010: 1447  sub r4, r4, r7              | r4 = 0x3 (3)
0011: af90  br -7
000a: b084  bz r4, 8
000b: 2652  and r6, r5, r2              | r6 = 0x2 (2)
000c: b026  bz r6, 2
000d: 0001  add r0, r0, r1              | r0 = 0x1b (27)
000e: 5117  shl r1, r1, r7              | r1 = 0x24 (36)
000f: 5557  shl r5, r5, r7              | r5 = 0x4 (4)
0010: 1447  sub r4, r4, r7              | r4 = 0x2 (2)
0011: af90  br -7
000a: b084  bz r4, 8
000b: 2652  and r6, r5, r2              | r6 = 0x0 (0)
000c: b026  bz r6, 2
000e: 5117  shl r1, r1, r7              | r1 = 0x48 (72)
000f: 5557  shl r5, r5, r7              | r5 = 0x8 (8)
0010: 1447  sub r4, r4, r7              | r4 = 0x1 (1)
0011: af90  br -7
000a: b084  bz r4, 8
000b: 2652  and r6, r5, r2              | r6 = 0x8 (8)
000c: b026  bz r6, 2
000d: 0001  add r0, r0, r1              | r0 = 0x63 (99)
000e: 5117  shl r1, r1, r7              | r1 = 0x90 (144)
000f: 5557  shl r5, r5, r7              | r5 = 0x10 (16)
0010: 1447  sub r4, r4, r7              | r4 = 0x0 (0)
0011: af90  br -7
000a: b084  bz r4, 8
0012: e0f0  jr, r15
0004: 9203  st r0, r3, 2                | M[0015] = 0x0063 (99)
0005: f000  sys, 0
```

At what address did the assembler place the function `main`?	`0x0000`
At what address did the assembler place the function `mult`?	`0x0003`
At what address did the assembler place the array `A`?	`0x0013`
What displacement did the assembler determine for the branch instruction associated with the C statement `goto finish`?	`0x0012-0x000a = 0x0008` `Displacement 8`
What displacement did the assembler determine for the branch instruction associated with the C statement `goto loop`?	`Loop was -7`
At what address did the program execution call the function `mult`?	`0x0003 calls mult`
At what address did program execution resume after it returned from the call to `mult`?	`It returns to 0x0004`

What to Turn In

Fill in the boxes on the previous pages with answers to all the questions. Save your Google Doc as a PDF and upload it to **Gradescope** for grading. Note Gradescope can be accessed via our Canvas site, or by visiting gradescope.com. Make sure all code you write has sufficient comments so that the graders can clearly identify the parts of the program. Use a fixed-width font (Consolas is preferred) for your code with proper indentation. Below is a checklist for this assignment:

Problem 1 (13 points)

	Deliverable	Points
1.	.mem file contents for program	0.5
2.	Simulation trace	0.5
3.	Hex address for X	2
4.	Hex for high(X) and low(X)	2
5.	.mem file contents for single ldi version of program	0.5
6.	Simulation trace for single ldi version of program	0.5
7.	Hex address for X	2
8.	Explanation for differences between program versions	2
9.	.mem file contents for Z version of program	0.5
10.	Simulation trace for Z version of program	0.5

11.	Explanation of when “Z + 2” is created in memory	2

Problem 2 (10 points)

	Deliverable	Points
1.	albaCore assembly for program	8
2.	.mem file contents for program	1
3.	Simulation trace	1

Problem 3 (10 points)

	Deliverable	Points
1.	.mem file contents for program	0.5
2.	Simulation trace	0.5
3.	**Six** questions about the program	1.5 each = 9

Problem 4 (10 points)

	Deliverable	Points
1.	.mem file contents for program	0.5
2.	Simulation trace	0.5
3.	**Six** questions about the program	1.5 each = 9

Problem 5 (10 points)

	Deliverable	Points
1.	**Five** disassembled instructions	1.5 each = 7.5
2.	albaCore program	1
3.	.mem file, should match the original hex	1.5

Problem 6 (22 points)

	Deliverable	Points
1.	Table for `m`, `mask`, `nbit`, and `prod`	3
2.	albaCore program	10
3.	.mem file contents for program	1
4.	Simulation trace	1
5.	**Seven** questions about the program	1 each = 7