HW02: Introduction to albaCore and Assembly Language
CSE 20221 Logic Design
Name(s): Aaron Wang & Ethan Little

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.

Path Setup:

Before starting this assignment, make sure you get the albaCore assembler and simulator up and running from your account on a CSE Linux machine – use one of student10 - student13 (running RHEL 8), these were used to test this assignment. The following instructions are repeated from Section 3.2.2 of your *Practical Logic and Processor Design* textbook (updated for our course's path).

Add the following line to your `.bashrc` file in your home directory (i.e., `~/.bashrc`) so that you don't need to retype it at the command line each time you log in.

```
export PATH=$PATH:/escnfs/courses/fa24-cse-20221.01/public/bin
```

To test that you now can access the tools, type:

```
source ~/.bashrc
which albaasmh
```

and it should display the path to the tools. It should look something like this:

```
adingler@student12:~$ source ~/.bashrc
adingler@student12:~$ which albaasmh
/escnfs/courses/fa24-cse-20221.01/public/bin/albaasmh
adingler@student12:~$
```

When you've got the path to the tools set properly, run through the example in Section 3.2.3.2 of *Practical Logic and Processor Design* to learn how to use the albaCore assembler and simulator.

Problem 1: Using C notation, a bitwise exclusive-or operation `r1 ^ r2` may be expressed as:

```
(r1 & ~r2) | (~r1 & r2)
```

Write an albaCore assembly language program that writes `0xEF` into `r1` and `0x1A` into `r2` and then evaluates their exclusive-or and writes the result to `r0`. Use additional registers for temporary values as needed.

albaCore assembly language program

```
.text
ldi r1, 0xEF
ldi r2, 0x1A
not r3, r2
and r3, r1, r3
not r4, r1
and r4, r4, r2
or  r0, r3, r4
quit
```

albasim simulation trace (run in interactive mode and enter 'c' to continue to end)

```
0000: 71ef  ldi r1, 239                  | r1 = 0xef (239)
0001: 721a  ldi r2, 26                   | r2 = 0x1a (26)
0002: 4320  not r3, r2                   | r3 = 0xffe5 (-27)
0003: 2313  and r3, r1, r3               | r3 = 0xe5 (229)
0004: 4410  not r4, r1                   | r4 = 0xff10 (-240)
0005: 2442  and r4, r4, r2               | r4 = 0x10 (16)
0006: 3034  or r0, r3, r4                | r0 = 0xf5 (245)
0007: f000  sys, 0
```

Problem 2: Complete the table below by either assembling the albaCore assembly language to machine code, or disassembling the machine code to assembly language. Check your answers by assembling a program with these instructions and comparing to the machine code memory image file. Note: see the *Practical Logic and Processor Design* textbook page 33 for all instructions and encodings, or the albaCore Reference Card (I strongly recommend you bookmark this page).

assembly language	machine code (hex)
`ldi r7, 0xAE`	`0x77AE`
`ldi r3, 3`	`0x7303`
`ldi r0, 0x11`	`0x7011`
`add r5, r7, r3`	`0x0573`
`st r5, r7, 5`	`0x9557`
`ld  r9, r0, 0x0`	`0x8900`
`sub r7, r9, r7`	`0x1797`
`quit`	`0xF000`

Memory image file output from assembler albaasmh

```
// .text
@0000 77ae // ldi r7, 174
@0001 7303 // ldi r3, 3
@0002 7011 // ldi r0, 17
@0003 0573 // add r5, r7, r3
@0004 9557 // st r5, r7, 5
@0005 8900 // ld r9, r0, 0
@0006 1797 // sub r7, r9, r7
@0007 f000 // sys 0

// .data
```

What value is in `r7` upon completion of the program? (Hint: the albaCore simulator can tell you.)

```
-174
```

Problem 3: Write a function in C that "disassembles" an albaCore arithmetic/logic instruction and prints each of its fields as a single hex digit. The input to the function should be an `int`, where the 16 upper bits are assumed to be 0 and the 16 lower bits contain the machine code of the instruction. For example, the call: `disassemble(0x1234)`

should print: `opcode = 1  rw = 2  ra = 3  rb = 4`

You may only use the C boolean and shift operators (`&, |, ~, <<, >>`) to extract the fields from the instruction – no arithmetic operations (+, -, etc.) or loops of any kind are allowed.

While not all instructions fit the format "`opcode rw ra rb`", for simplicity your program only needs to print in this format.

Test your program with the following instructions: `0xABBA, 0xDABA, 0xD000`.

C program with disassemble function and main with test cases

```c
#include <stdio.h>

void disassemble(int);

int main(){
   disassemble(0xABBA);
   disassemble(0xDABA);
   disassemble(0xD000);
}

void disassemble (int instruction){
   int opcode = (instruction & 0x0000F000) >> 12;
   int rw =     (instruction & 0x00000F00) >> 8;
   int ra =     (instruction & 0x000000F0) >> 4;
   int rb =     (instruction & 0x0000000F);

   printf("opcode = %x\t", opcode);
   printf("rw = %x\t", rw);
   printf("ra = %x\t", ra);
   printf("rb = %x\n", rb);
}
```

Printed results of test cases

```
opcode = a      rw = b  ra = b  rb = a
opcode = d      rw = a  ra = b  rb = a
opcode = d      rw = 0  ra = 0  rb = 0
```

A few important things to remember as you work with the hex encodings this semester: 1) hex is simply a more compact way to write binary; 2) in an instruction set architecture, there may be encodings that don't match valid instruction opcodes – not the case in albaCore, which uses all 16 opcodes; 3) there may be encodings that contain superfluous information. The technical term for "superfluous information" is a "don't care", meaning the value for this field does not impact the processing of the instruction. Don't cares are denoted by an "X" in the 16-bit encoding column in *Practical Logic and Processor Design*.

Which of the given instructions (`0xABBA, 0xDABA, 0xD000`), if any, specify superfluous information?

`0xABBA: In this one the first A points to direction "br" which renders the last 4 bits (the second A) as superfluous information.`
`0xDABA 0xD000: These do not have superfluous information.`

Problem 4: Write a function in C that adds two 32-bit signed integers, prints the values and the sum in decimal and hex, and then prints whether or not an overflow occurred. You may only use the C boolean operators (`&, |, ~`) in your test for overflow; you cannot use the relational operators (e.g., `<, >, ==`), arithmetic operations (other than addition to get the sum), or loops. Note that any non-zero value evaluates to 'true' as the condition for an `if` statement in C. Test your function using the cases given below. **Tip:** see our definitions of overflow from 9/3/24 for ways to approach the overflow detection.

A. 481 + 516
B. -200 + -150
C. 0x7FFFFFFF + 1
D. 0x80000000 + -1
E. 0x7FFFFFFF + 0x80000000

C program with overflow function and main with test cases

```c
#include <stdio.h>
void additionOverflow(int, int);


int main(){
   additionOverflow(481, 516);
   additionOverflow(-200, -150);
   additionOverflow(0x7FFFFFFF, 1);
   additionOverflow(0x80000000, -1);
   additionOverflow(0x7FFFFFFF, 0x80000000);
}


void additionOverflow(int a, int b){
   int sum = a + b;
   printf("Sum: Decimal: %11d; Hex: %8x ", sum, sum);
```

```
    //only store most significant bit(sign) of each num
    int a1 = a & 0x80000000;
    int b1 = b & 0x80000000;
    int sum1 = sum & 0x80000000;

    //if a sign and b sign are diff no overflow
    //if they are the same as the sum no overflow
    // if they are same and sum is diff overflow

  if ((a1 & b1 & ~sum1) | (~a1 & ~b1 & sum1)){
      printf("Overflow\n");
  } else {
      printf("No Overflow\n");
  }
}
```

Printed results of test cases

```
Sum: Decimal:         997; Hex:      3e5 No Overflow
Sum: Decimal:        -350; Hex: fffffea2 No Overflow
Sum: Decimal: -2147483648; Hex: 80000000 Overflow
Sum: Decimal:  2147483647; Hex: 7fffffff Overflow
Sum: Decimal:          -1; Hex: ffffffff No Overflow
```

Problem 5: Write an albaCore assembly language program that stores the value `0xBABA` to memory locations `0x31` and `0x35`. Take advantage of the `st` offset field to minimize the number of instructions required for the store operations. (We will not cover this until Th 9/5, but you can work ahead by referring to 3.2.4 in *Practical Logic and Processor Design*.)

albaCore assembly language program

```
.text
ldi r0, 0xBA    //load in first half
ldi r1, 8       //load how many bits to shift
shl r0, r0, r1  //shift first half
ldi r1, 0xBA    //load second half
or r0, r0, r1   //add second half to first half
ldi r2, 0x31    //load memory address
st r0, r2, 0    //store at 0x31
st r0, r2, 4    //store at 0x35
quit
```

albasim simulation trace (run in interactive mode and enter 'c' to continue to end)

```
0000: 70ba  ldi r0, 186                  | r0 = 0xba (186)
0001: 7108  ldi r1, 8                    | r1 = 0x8 (8)
0002: 5001  shl r0, r0, r1               | r0 = 0xba00 (-17920)
0003: 71ba  ldi r1, 186                  | r1 = 0xba (186)
0004: 3001  or r0, r0, r1                | r0 = 0xbaba (-17734)
0005: 7231  ldi r2, 49                   | r2 = 0x31 (49)
0006: 9002  st r0, r2, 0                 | M[0031] = 0xbaba (-17734)
0007: 9402  st r0, r2, 4                 | M[0035] = 0xbaba (-17734)
0008: f000  sys, 0
```

Problem 6: Upload a brief bio of yourself to Gradescope that includes your:

- Name
- Photo
- Major
- Hometown/Country
- Academic/Professional Interests
- Other Interests
- Other Courses this Semester
- Anything else you want to share

While the rest of the assignment should be uploaded as a group, you should upload this individually. (Problem 6 is ungraded; if you are uncomfortable providing certain information for some reason, you may omit it without explanation and without penalty.)

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 (10 points)

	Deliverable	Points
1.	albaCore program for XOR	6
2.	Simulation trace	4

Problem 2 (10 points)

	Deliverable	Points
1.	Memory image file from albaasmh	8
2.	Value of r1	2

Problem 3 (12 points)

	Deliverable	Points
1.	C code for disassemble with test cases	7
2.	Test results	3
3.	Don’t cares/superfluous information	2

Problem 4 (10 points)

	Deliverable	Points
1.	Integer overflow detection using &, |, ~	7.5

2.	Test results	2.5

Problem 5 (10 points)

	Deliverable	Points
1.	albaCore program to store data to memory	6
2.	Simulation trace	4

Problem 6 (ungraded)

	Deliverable	Points
1.	Upload bio individually to separate Gradescope assignment	-