HW01: ISA Review and RISC-V Introduction
CSE 30321 Computer Architecture
Names: Aaron Wang and Ethan Little

Preamble:

1. 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 insert images as long as they are 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.

2. You are encouraged to work in groups of **two**. Please type your names at the top of this document.

Problem 1: ISA Review

We will primarily work with the RISC-V instruction set architecture (ISA) this semester. This first problem is intended to get you comfortable writing assembly language in a familiar way with albaCore. You can reuse the albaCore setup you had from a previous semester, or set up your path again on one of the student machines (see "Path Setup" at the end of this document).

albaCore assembly will not appear on the exams in this class, though you may be tested on qualitative differences between albaCore and RISC-V.

Part 1: Pseudocode to albaCore

Convert the following two pseudocode snippets to albaCore assembly. Implement these in two separate files, assemble them, and test in the simulator to ensure they work correctly. For each pseudocode snippet, copy your assembly code, the assembled code from the memory image file, and the simulation results (for Part 1 only) into the boxes below.

Pseudocode snippet 1:

Assume you have three data variables `i`, `j` and `k` at memory locations `0x58`, `0x5C`, and `0x60`, respectively. Write albaCore instructions to move `j` to memory location `0x60`, `k` to memory location `0x58`, and `i` to memory location `0x5C`. For full points, use a minimal number of instructions to do this – it is possible to do this with 6 instructions. Assume `r1` holds the value `0x58`. When testing, start with the following instructions to setup memory and `r1`:

```
ldi r1, 0x58
ldi r2, 0x5
st r2, r1, 0      //set i = 5
add r2, r2, r2
st r2, r1, 4      //set j = 10
add r2, r2, r2
st r2, r1, 8      //set k = 20
```

These instructions will not count toward your total. You should add a `quit` instruction which will also not count. You will not be graded on following register usage conventions.

albaCore assembly language program

```
.text
ldi r1, 0x58
ldi r2, 0x5
st r2, r1, 0      //set i = 5
add r2, r2, r2
st r2, r1, 4      //set j = 10
add r2, r2, r2
st r2, r1, 8      //set k = 20
ld r3, r1, 0
ld r4, r1, 4
ld r5, r1, 8
st r5, r1, 0
st r3, r1, 4
st r4, r1, 8
quit
```

Memory image file output from assembler albaasmh

```
// .text
@0000 7158 // ldi r1, 88
@0001 7205 // ldi r2, 5
@0002 9021 // st r2, r1, 0
@0003 0222 // add r2, r2, r2
@0004 9421 // st r2, r1, 4
@0005 0222 // add r2, r2, r2
@0006 9821 // st r2, r1, 8
@0007 8301 // ld r3, r1, 0
@0008 8441 // ld r4, r1, 4
@0009 8581 // ld r5, r1, 8
@000a 9051 // st r5, r1, 0
@000b 9431 // st r3, r1, 4
@000c 9841 // st r4, r1, 8
@000d f000 // sys 0

// .data
```

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

```
0000: 7158  ldi r1, 88                      | r1 = 0x58 (88)
0001: 7205  ldi r2, 5                       | r2 = 0x5 (5)
0002: 9021  st r2, r1, 0                    | M[0058] = 0x0005 (5)
0003: 0222  add r2, r2, r2                  | r2 = 0xa (10)
0004: 9421  st r2, r1, 4                    | M[005c] = 0x000a (10)
0005: 0222  add r2, r2, r2                  | r2 = 0x14 (20)
```

```
0006: 9821  st r2, r1, 8                    | M[0060] = 0x0014 (20)
0007: 8301  ld r3, r1, 0                    | r3 = 0x5 (5)
0008: 8441  ld r4, r1, 4                    | r4 = 0xa (10)
0009: 8581  ld r5, r1, 8                    | r5 = 0x14 (20)
000a: 9051  st r5, r1, 0                    | M[0058] = 0x0014 (20)
000b: 9431  st r3, r1, 4                    | M[005c] = 0x0005 (5)
000c: 9841  st r4, r1, 8                    | M[0060] = 0x000a (10)
```

Pseudocode snippet 2:

Write albaCore assembly to implement the following statement:

```
x = y - (i - j) + (k << z)
```

Assumptions:

y is in `r1`; z is in `r2`; `i` is in `r3`; `j` is in `r4`; `k` is in memory location 0x24; x is in memory location 0x98

Initialize y to 20, z to 4, `i` to 88, `j` to 29, and k to 8 to test your program.
To initialize `k`, use a labeled memory location in the `.data` segment[1] , as shown in this skeleton:

```
.data
k: 8

.text

quit
```

You may add code and modify the skeleton as long as you can still refer to k using a sequence like this:

```
//load k
ldi r5, high(k)
ldi r0, 8
shl r5, r5, r0
ldi r0, low(k)
or r5, r5, r0      //address of k is now in r5
ld r5, r5, 0       //k is now in r5
```

For x, you do not need to use a named memory location - you can store directly to a hardcoded address 0x98.

You will not be graded on following register usage conventions.

albaCore assembly language program

[1] How? albaCore has no way to specify a memory location directly, you will have to come up with a “hack” to force k to be at address 0x24.

```
.data
k: 8

.text
or r1, r1, r1 //filler to get to 0x24
or r1, r1, r1
or r1, r1, r1
or r1, r1, r1
or r1, r1, r1
or r1, r1, r1
or r1, r1, r1
or r1, r1, r1
or r1, r1, r1
or r1, r1, r1
or r1, r1, r1
or r1, r1, r1
or r1, r1, r1
or r1, r1, r1
or r1, r1, r1
or r1, r1, r1
or r1, r1, r1
or r1, r1, r1
or r1, r1, r1
ldi r1, 20
ldi r2, 4
ldi r3, 88
ldi r4, 29
ldi r5, high(k)
ldi r0, 8
shl r5, r5, r0
ldi r0, low(k)
or r5, r5, r0     //address of k is now in r5
ld r5, r5, 0      //k is now in r5
sub r6, r3, r4
shl r7, r5, r2
sub r8, r1, r6
add r9, r8, r7
ldi r10, 0x98
st r9, r10, 0
quit
```

Memory image file output from assembler albaasmh

```
// .text
@0000 3111 // or r1, r1, r1
@0001 3111 // or r1, r1, r1
@0002 3111 // or r1, r1, r1
@0003 3111 // or r1, r1, r1
@0004 3111 // or r1, r1, r1
```

```
@0005 3111 // or r1, r1, r1
@0006 3111 // or r1, r1, r1
@0007 3111 // or r1, r1, r1
@0008 3111 // or r1, r1, r1
@0009 3111 // or r1, r1, r1
@000a 3111 // or r1, r1, r1
@000b 3111 // or r1, r1, r1
@000c 3111 // or r1, r1, r1
@000d 3111 // or r1, r1, r1
@000e 3111 // or r1, r1, r1
@000f 3111 // or r1, r1, r1
@0010 3111 // or r1, r1, r1
@0011 3111 // or r1, r1, r1
@0012 3111 // or r1, r1, r1
@0013 7114 // ldi r1, 20
@0014 7204 // ldi r2, 4
@0015 7358 // ldi r3, 88
@0016 741d // ldi r4, 29
@0017 7500 // ldi r5, 0    target label: k
@0018 7008 // ldi r0, 8
@0019 5550 // shl r5, r5, r0
@001a 7024 // ldi r0, 36    target label: k
@001b 3550 // or r5, r5, r0
@001c 8505 // ld r5, r5, 0
@001d 1634 // sub r6, r3, r4
@001e 5752 // shl r7, r5, r2
@001f 1816 // sub r8, r1, r6
@0020 0987 // add r9, r8, r7
@0021 7a98 // ldi r10, 152
@0022 909a // st r9, r10, 0
@0023 f000 // sys 0

// .data
@0024 0008 //  label: k
```

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

```
albasim> c
0000: 3111  or r1, r1, r1                 | r1 = 0x0 (0)
0001: 3111  or r1, r1, r1                 | r1 = 0x0 (0)
0002: 3111  or r1, r1, r1                 | r1 = 0x0 (0)
0003: 3111  or r1, r1, r1                 | r1 = 0x0 (0)
0004: 3111  or r1, r1, r1                 | r1 = 0x0 (0)
0005: 3111  or r1, r1, r1                 | r1 = 0x0 (0)
0006: 3111  or r1, r1, r1                 | r1 = 0x0 (0)
0007: 3111  or r1, r1, r1                 | r1 = 0x0 (0)
0008: 3111  or r1, r1, r1                 | r1 = 0x0 (0)
0009: 3111  or r1, r1, r1                 | r1 = 0x0 (0)
```

```
000a: 3111  or r1, r1, r1                   | r1 = 0x0 (0)000b: 3111  or r1,
r1, r1                  | r1 = 0x0 (0)
000c: 3111  or r1, r1, r1                   | r1 = 0x0 (0)
000d: 3111  or r1, r1, r1                   | r1 = 0x0 (0)
000e: 3111  or r1, r1, r1                   | r1 = 0x0 (0)
000f: 3111  or r1, r1, r1                   | r1 = 0x0 (0)
0010: 3111  or r1, r1, r1                   | r1 = 0x0 (0)
0011: 3111  or r1, r1, r1                   | r1 = 0x0 (0)
0012: 3111  or r1, r1, r1                   | r1 = 0x0 (0)
0013: 7114  ldi r1, 20                      | r1 = 0x14 (20)
0014: 7204  ldi r2, 4                       | r2 = 0x4 (4)
0015: 7358  ldi r3, 88                      | r3 = 0x58 (88)
0016: 741d  ldi r4, 29                      | r4 = 0x1d (29)
0017: 7500  ldi r5, 0                       | r5 = 0x0 (0)
0018: 7008  ldi r0, 8                       | r0 = 0x8 (8)
0019: 5550  shl r5, r5, r0                  | r5 = 0x0 (0)
001a: 7024  ldi r0, 36                      | r0 = 0x24 (36)
001b: 3550  or r5, r5, r0                   | r5 = 0x24 (36)
001c: 8505  ld r5, r5, 0                    | r5 = 0x8 (8)
001d: 1634  sub r6, r3, r4                  | r6 = 0x3b (59)
001e: 5752  shl r7, r5, r2                  | r7 = 0x80 (128)
001f: 1816  sub r8, r1, r6                  | r8 = 0xffd9 (-39)
0020: 0987  add r9, r8, r7                  | r9 = 0x59 (89)
0021: 7a98  ldi r10, 152                    | r10 = 0x98 (152)
0022: 909a  st r9, r10, 0                   | M[0098] = 0x0059 (89)
0023: f000  sys, 0
```

Part 2: C to albaCore

Convert the following C snippet to albaCore assembly. Implement the code in a new file, assemble it, and test in the simulator to ensure it works correctly. Copy your assembly code, the assembled code from the memory image file, and the simulation results into the boxes below.

C snippet 1:

```
for (i = 0; i < 18; i++)
      if (x == i)
            y--;
      else
            y++;
```

Assumptions:

i is in r1; x is in r2; y is in r3; z is in memory location 0x74, initially 0
(You do not need to put z in the data section)

Initialize i to 0, x to 6, and y to 13 to test your program.

Try to use as few instructions as possible, though there will not be a hard minimum number for full credit. Include detailed comments for your assembly program that connect it to the C program. For example:

```
      ldi r1, 0   //initialize i = 0, for the for loop
```

You will not be graded on following register usage conventions.

albaCore assembly language program

```
.text
ldi r1, 0       // initialize i
ldi r2, 6       // initialize x
ldi r3, 13      // initialize y
ldi r4, 17      // use for for loop check branch negative
ldi r6, 1       // constant to use for incrementing
sub r5, r4, r1  // calculate for branch negative
bn r5, 8        // if for loop condition is false jump to end
sub r5, r2, r1  // use to check if condition
bz r5, 3        // if condition satisfied jump to if block
add r3, r3, r6  // this is the else
br 2            // jump to end of if-else
sub r3, r3, r6  // if block
add r1, r1, r6  //  increment i
br -8           //  sent back to top of for loop
quit
```

Memory image file output from assembler albaasmh

```
// .text
@0000 7100 // ldi r1, 0
@0001 7206 // ldi r2, 6
@0002 730d // ldi r3, 13
@0003 7411 // ldi r4, 17
@0004 7601 // ldi r6, 1
@0005 1541 // sub r5, r4, r1
@0006 c085 // bn r5, 8
@0007 1521 // sub r5, r2, r1
@0008 b035 // bz r5, 3
@0009 0336 // add r3, r3, r6
@000a a020 // br 2
@000b 1336 // sub r3, r3, r6
@000c 0116 // add r1, r1, r6
@000d af80 // br -8
@000e f000 // sys 0

// .data
```

You are *not* required to include simulation output (it will be quite long).

Problem 2: RISC-V Introduction

QtRVSim setup:
QtRVSim is an open source RISC-V simulator developed for teaching at Czech Technical University. QtRVSim is stable and has a good visualization of various RISC-V machines.

Installing QtRVSim:
Download the v0.9.8 snapshot or source from: https://github.com/cvut/qtrvsim/releases
I have tested QtRVSim in MacOS, Windows and Linux and found no significant issues - but please let me know if you run into any!

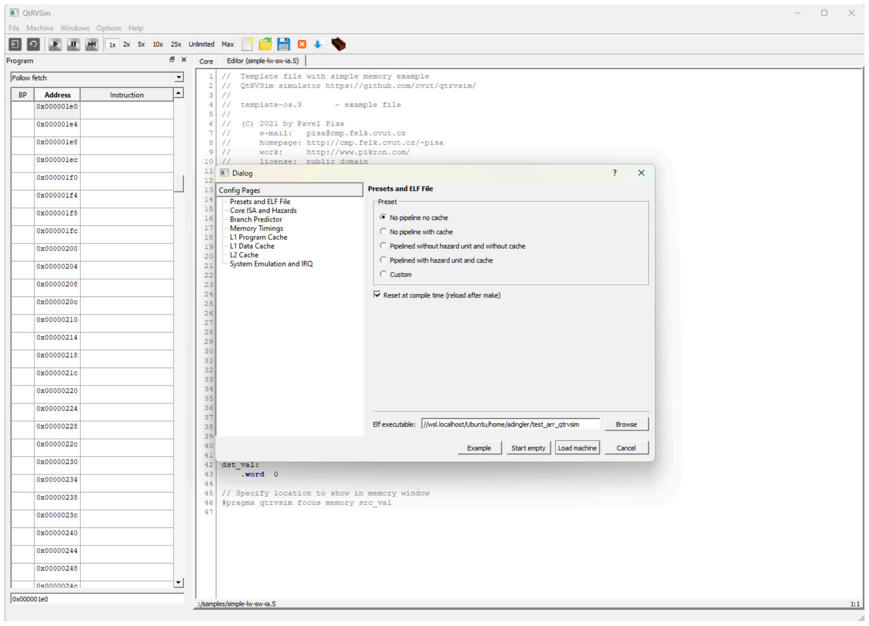
There is a WebAssembly version for use in your browser, but this is *experimental* and may not work well. It is **highly recommended** you use the desktop version.

Editor/IDE suggestions:
The built-in editor is fine, but limited. You can instead use your preferred editor and then close/reopen the file in QtRVSim each time you make changes – or just copy the code into QtRVSim after each change. For RISC-V syntax highlighting, you can try this extension for VS Code, and this plugin for vim (or neovim). I have tested both on my Macbook and they worked fine at the time; there are of course other options out there.

Part 1: Testing QtRVSim

This is a step-by-step tutorial to get you started, there are no deliverables until Part 2.

1. Open QtRVSim, you should see something like this – the windows shown may differ at first, that is OK. (These screenshots were taken on Windows 11, so yours may look a bit different.)



BP	Address	Instruction
	0x000001e0	
	0x000001e4	
	0x000001e8	
	0x000001ec	
	0x000001f0	
	0x000001f4	
	0x000001f8	
	0x000001fc	
	0x00000200	
	0x00000204	
	0x00000208	
	0x0000020c	
	0x00000210	
	0x00000214	
	0x00000218	
	0x0000021c	
	0x00000220	
	0x00000224	
	0x00000228	
	0x0000022c	
	0x00000230	
	0x00000234	
	0x00000238	
	0x0000023c	
	0x00000240	
	0x00000244	
	0x00000248	

0x000001e0

```
//  Template file with simple memory example
//  QtRVSim simulator https://github.com/cvut/qtrvsim/
//
//  template-os.S       - example file
//
//  (C) 2021 by Pavel Pisa
//      e-mail:   pisa@cmp.felk.cvut.cz
//      homepage: http://cmp.felk.cvut.cz/~pisa
//      work:     http://www.pikron.com/
//      license:  public domain
```

```
dst_val:
    .word  0

// Specify location to show in memory window
#pragma qtrvsim focus memory src_val
```

Presets and ELF File

Preset
- [✓ No pipeline no cache]
- No pipeline with cache
- Pipelined without hazard unit and without cache
- Pipelined with hazard unit and cache
- Custom

✓ Reset at compile time (reload after make)

Elf executable: //wsl.localhost/Ubuntu/home/adingler/test_arr_qtrvsim

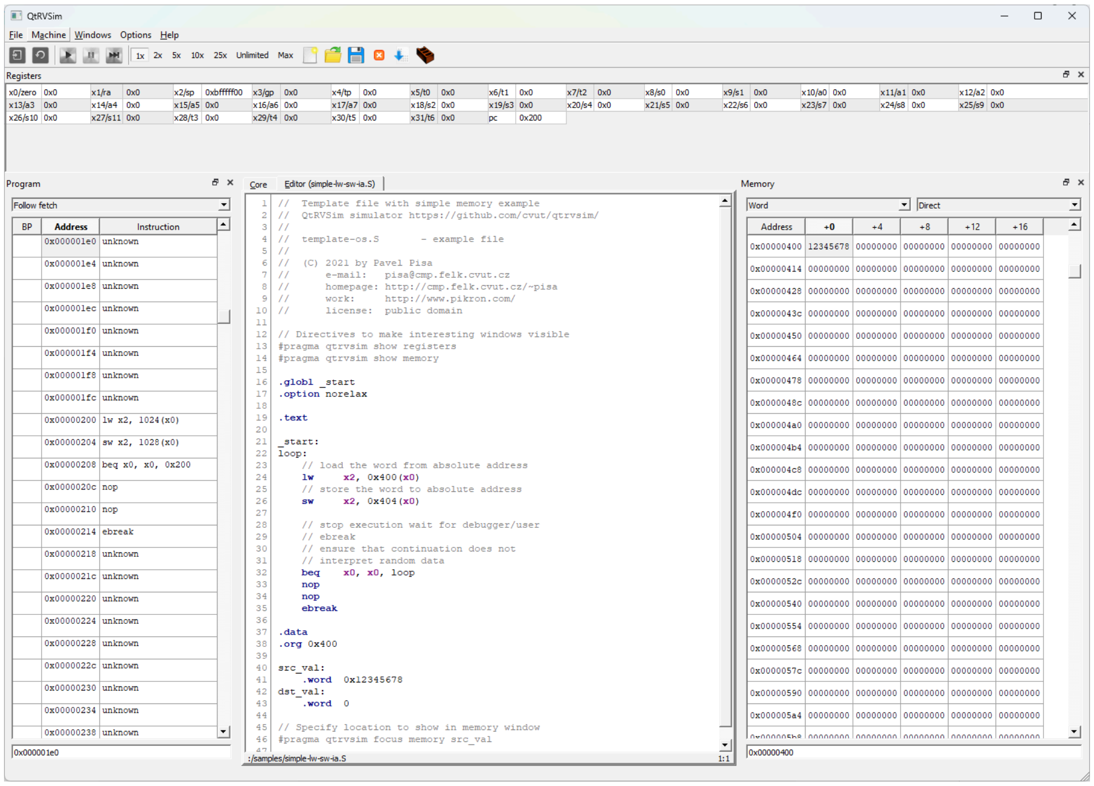
:/samples/simple-lw-sw-ia.S

1:1

2. Select the “No pipeline no cache” radio button, ensure the checkbox for “Reset at compile time” is checked, and then click “Start empty”. This will start a new simulation with no program loaded into memory.
3. Next, click “File → Examples → simple-lw-sw-ia.S”
4. This will open a simple example program in the built-in editor

5. Click the “Compile source and update memory”[2] button:

   (or click “Machine → Compile Source”; note the menus list keyboard shortcuts to use)
6. You should now see the Program, Registers, and Memory windows as in the screenshot below.



x0/zero	0x0	x1/ra	0x0	x2/sp	0xbfffff00	x3/gp	0x0	x4/tp	0x0	x5/t0	0x0	x6/t1	0x0	x7/t2	0x0	x8/s0	0x0	x9/s1	0x0	x10/a0	0x0	x11/a1	0x0	x12/a2	0x0
x13/a3	0x0	x14/a4	0x0	x15/a5	0x0	x16/a6	0x0	x17/a7	0x0	x18/s2	0x0	x19/s3	0x0	x20/s4	0x0	x21/s5	0x0	x22/s6	0x0	x23/s7	0x0	x24/s8	0x0	x25/s9	0x0
x26/s10	0x0	x27/s11	0x0	x28/t3	0x0	x29/t4	0x0	x30/t5	0x0	x31/t6	0x0	pc	0x200												

BP	Address	Instruction
	0x000001e0	unknown
	0x000001e4	unknown
	0x000001e8	unknown
	0x000001ec	unknown
	0x000001f0	unknown
	0x000001f4	unknown
	0x000001f8	unknown
	0x000001fc	unknown
	0x00000200	lw x2, 1024(x0)
	0x00000204	sw x2, 1028(x0)
	0x00000208	beq x0, x0, 0x200
	0x0000020c	nop
	0x00000210	nop
	0x00000214	ebreak
	0x00000218	unknown
	0x0000021c	unknown
	0x00000220	unknown
	0x00000224	unknown
	0x00000228	unknown
	0x0000022c	unknown
	0x00000230	unknown
	0x00000234	unknown
	0x00000238	unknown

0x000001e0

```
//  Template file with simple memory example
//  QtRVSim simulator https://github.com/cvut/qtrvsim/
//
//  template-os.S       - example file
//
//  (C) 2021 by Pavel Pisa
//      e-mail:   pisa@cmp.felk.cvut.cz
//      homepage: http://cmp.felk.cvut.cz/~pisa
//      work:     http://www.pikron.com/
//      license:  public domain

// Directives to make interesting windows visible
#pragma qtrvsim show registers
#pragma qtrvsim show memory

.globl _start
.option norelax

.text

_start:
loop:
    // load the word from absolute address
    lw      x2, 0x400(x0)
    // store the word to absolute address
    sw      x2, 0x404(x0)

    // stop execution wait for debugger/user
    // ebreak
    // ensure that continuation does not
    // interpret random data
    beq     x0, x0, loop
    nop
    nop
    ebreak

.data
.org 0x400

src_val:
    .word  0x12345678
dst_val:
    .word  0

// Specify location to show in memory window
#pragma qtrvsim focus memory src_val
```

:/samples/simple-lw-sw-ia.S 1:1

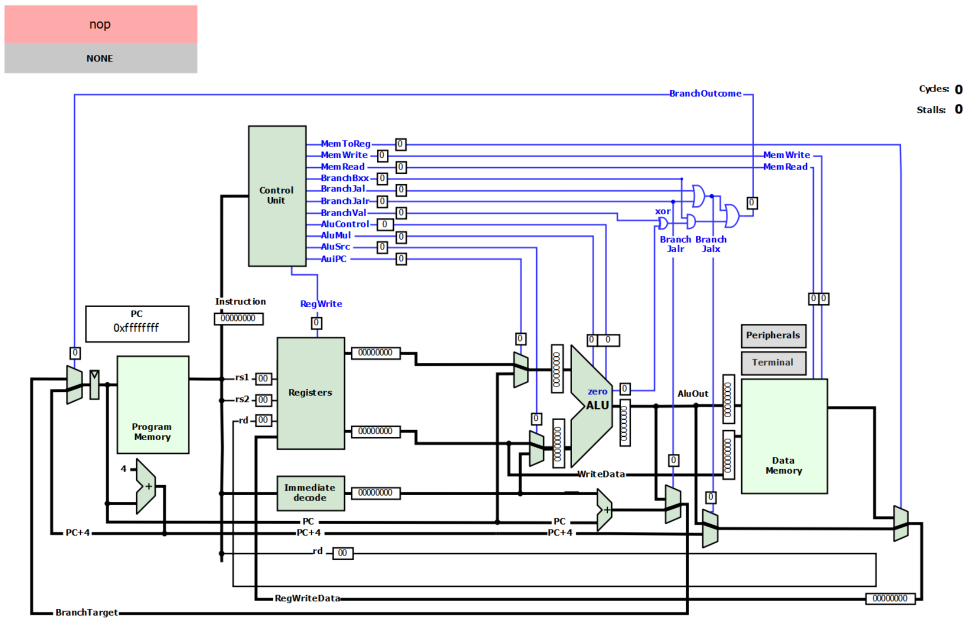
Address	+0	+4	+8	+12	+16
0x00000400	12345678	00000000	00000000	00000000	00000000
0x00000414	00000000	00000000	00000000	00000000	00000000
0x00000428	00000000	00000000	00000000	00000000	00000000
0x0000043c	00000000	00000000	00000000	00000000	00000000
0x00000450	00000000	00000000	00000000	00000000	00000000
0x00000464	00000000	00000000	00000000	00000000	00000000
0x00000478	00000000	00000000	00000000	00000000	00000000
0x0000048c	00000000	00000000	00000000	00000000	00000000
0x000004a0	00000000	00000000	00000000	00000000	00000000
0x000004b4	00000000	00000000	00000000	00000000	00000000
0x000004c8	00000000	00000000	00000000	00000000	00000000
0x000004dc	00000000	00000000	00000000	00000000	00000000
0x000004f0	00000000	00000000	00000000	00000000	00000000
0x00000504	00000000	00000000	00000000	00000000	00000000
0x00000518	00000000	00000000	00000000	00000000	00000000
0x0000052c	00000000	00000000	00000000	00000000	00000000
0x00000540	00000000	00000000	00000000	00000000	00000000
0x00000554	00000000	00000000	00000000	00000000	00000000
0x00000568	00000000	00000000	00000000	00000000	00000000
0x0000057c	00000000	00000000	00000000	00000000	00000000
0x00000590	00000000	00000000	00000000	00000000	00000000
0x000005a4	00000000	00000000	00000000	00000000	00000000

0x00000400

If at any time a window does not appear, click the Windows menu and then select the missing window (e.g., “Windows → Memory”).

[2] Sidenote: calling this “compiling” is a bit misleading, since we are not compiling from C to assembly, but there is more going on here than just assembling (evaluating the pragmas, for instance).

7. You can also click Core to see what is going on in the datapath as the program executes. This is what it should look like now, i.e., before any instructions have been executed:



8. Click the Run button ▶. By default, this should execute 1 instruction per second.
9. Other useful controls:
   a. ⟲ will reset the machine (cycles to 0, registers reset to initial state, etc.) *but not the state of memory*. To start over from the initial state, click the ⬇ button again.
   b. You can use the step button ⏭ to step one cycle at a time.
   c. Click ⏸ to pause a simulation that is in progress. This is useful for stepping the simulation cycle-by-cycle, e.g., to debug once reaching a certain instruction, or when an infinite loop is encountered.

Part 2:

1. Explain what this program does in your own words. As part of your explanation, include the following information: what are the memory *addresses* of `src_val` and `dst_val` in memory, and how did you find this?

```
This program is an infinite loop that copies src_val to dst_val.
```

```
src_val: 0x400
dst_val: 0x404
.org sets the data to start at 0x400. Since src_val is one word, dst_val
takes the next address 0x404.
Additionally, you can see this in the memory tab as 0x400 is initialized to
12345678 and sw x2, 0x404(x0) writes 0x12345678 to 0x404
```

Now, you will make a change to the example file and re-run the program. First, save the example file as a new file called “HW01_example.S” – click “File → Save source as” to do this. Then, replace the `beq` instruction with a `j` (jump) pseudoinstruction[3]. The format of the jump pseudoinstruction is “`j LABEL`”. Recompile and run the program and answer the following questions.

2. Does the program have the same effect? Specifically, are the results in the “Registers” and “Memory” window the same, or different?

```
Yes they are the same. The registers and the memory window are the same.
```

3. What real instruction is used to implement the `j` pseudoinstruction? List the instruction and its operands. Hint: look at the Program window or RISC-V Reference Data Card to see what instruction is actually being executed.

```
jal x0, 0x200
```

4. Explain what the following pragma does:
   `#pragma qtrvsim focus memory src_val`

   Tip: try changing the `src_val` on this line to `dst_val`, or to a memory address (e.g., `0x800`), re-compile, and see what happens.

```
This directions moves the memory tab to focus on that specific memory
address.
```

Part 3:
Finally, implement pseudocode snippet 1 in RISC-V. Your approach should be the same – 6 instructions, minimally, not counting setup instructions – but you will need to make a number of changes to match the new ISA.

1. Start with the template_cse30321.S file.

[3] Pseudoinstructions are not true instructions implemented in the hardware. The assembler will convert a pseudoinstruction to one or more real instructions to accomplish the desired task. For example, the `mv` (move) instruction uses an `addi`. See the RISC-V Reference Data Card (“Green Sheet”) for details.

2. Change the .data section of your code to the following:

```
.data
.org 0x58
i:
      .word 0x5
j:
      .word 0xA
k:
      .word 0x14
```

   This will place `i` at `0x58`, `j` at `0x5C`, and `k` at `0x60`.

   Recall: in a 32-bit system, a *word* is 4 bytes (32 bits). We typically work with the load word (`lw`) and store word (`sw`) instructions which operate on data words, so addresses are specified in multiples of 4.

3. The first instruction in your program should set `x1` to `0x58`. Rather than hardcoding this, use the following instruction to load the address of i into `x1`:

```
la x1, i
```

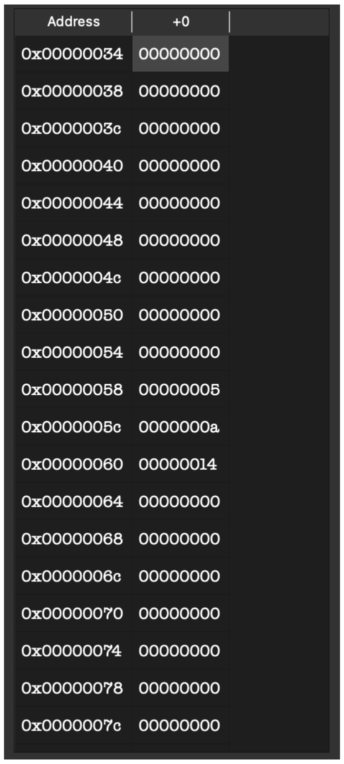
   (You may have noticed that `x1` is used for the return address; ignore this for now, we will discuss register usage conventions later this semester.)

4. Now, implement the data movement in RISC-V assembly, compile, and run your program. You will include a "before" and "after" screenshot of memory.

RISC-V assembly language program

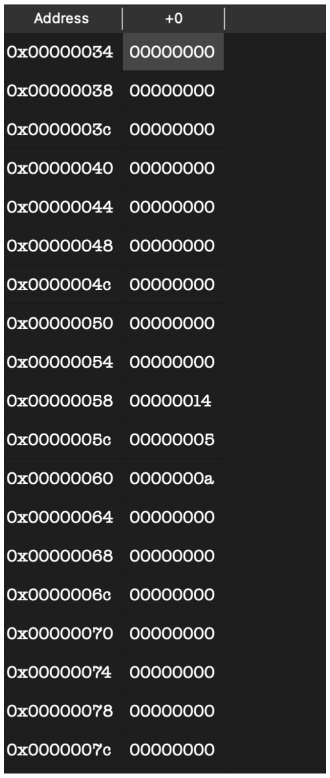
```
.globl _start
.option norelax
.text
_start:
      la x1, i
      lw x2, 0x00(x1)
      lw x3, 0x04(x1)
      lw x4, 0x08(x1)
      sw x4, 0x00(x1)
      sw x2, 0x04(x1)
      sw x3, 0x08(x1)
      ebreak
.data
.org 0x58
i:
      .word 0x5
j:
      .word 0xA
k:
      .word 0x14
```

Screenshot showing the original state of memory, focused on address `0x34`



Address	+0
0x00000034	00000000
0x00000038	00000000
0x0000003c	00000000
0x00000040	00000000
0x00000044	00000000
0x00000048	00000000
0x0000004c	00000000
0x00000050	00000000
0x00000054	00000000
0x00000058	00000005
0x0000005c	0000000a
0x00000060	00000014
0x00000064	00000000
0x00000068	00000000
0x0000006c	00000000
0x00000070	00000000
0x00000074	00000000
0x00000078	00000000
0x0000007c	00000000

Screenshot showing the final state of memory, focused on address `0x34`



Address	+0
0x00000034	00000000
0x00000038	00000000
0x0000003c	00000000
0x00000040	00000000
0x00000044	00000000
0x00000048	00000000
0x0000004c	00000000
0x00000050	00000000
0x00000054	00000000
0x00000058	00000014
0x0000005c	00000005
0x00000060	0000000a
0x00000064	00000000
0x00000068	00000000
0x0000006c	00000000
0x00000070	00000000
0x00000074	00000000
0x00000078	00000000
0x0000007c	00000000

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 (30 points)

	Deliverable	Points
1.	Part 1, pseudocode snippet 1: albaCore assembly code	5
2.	Part 1, pseudocode snippet 1: memory image file	2.5
3.	Part 1, pseudocode snippet 1: simulation trace	2.5
4.	Part 1, pseudocode snippet 2: albaCore assembly code	6.5
5.	Part 1, pseudocode snippet 2: memory image file	2.5
6.	Part 1, pseudocode snippet 2: simulation trace	2.5
7.	Part 2, C snippet 1: albaCore assembly code	6
8.	Part 2, C snippet 1: memory image file	2.5

Problem 2 (25 points)

	Deliverable	Points
1.	Part 2, explain what the program does	3
2.	Part 2, replace `beq` with `j` and note behavior	3
3.	Part 2, list instruction that implements `j`	3
4.	Part 2, explain `pragma`	3
5.	Part 3, pseudocode snippet 1: RISC-V assembly code	5
6.	Part 3, pseudocode snippet 1: original memory state	4
7.	Part 3, pseudocode snippet 1: final memory state	4

albaCore Path Setup:

This is a copy of the path setup instructions from CSE 20221, modified for our Comp Arch course directory. If you need a refresher, read the textbook section 3.2.3.2 (PDF textbook is in Drive here).

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).

Type the following at the Linux command line to add them to your path. You should also add it to your `.bashrc` file so that you don't need to retype it at the command line each time you log in.

```
export PATH=$PATH:/escnfs/courses/sp25-cse-30321.01/public/bin
```

To test that you now can access the tools type:

```
which albasimh
```

and it should display the path to the tools.