HW04: Procedures, Pipelining, and Sorting
CSE 30321 Computer Architecture
Name(s): Ethan Little and Aaron Wang

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 2. Please type your names at the top of this document.

Problem 1: Procedures

Question 1

In this problem, you will answer a question about a sequence of function calls. Indentation means that the indented function is called by the function listed above it. For example, the notation below suggests that `functionB()` is called by `functionA()`:

```
functionA()
      functionB()
```

Now, consider the following sequence of function calls:

```
main()
      functionA()
            functionB()
            functionC()
                  functionD()
      functionE()
            functionF()
      functionG()
```

Assume that the return address *from* main() does **not** need to be saved. Assume register x1 is always used as the return address.

What is the **minimum** number of times the return address register (x1) must be saved to the stack? Justify your answer.

```
The return address register (x1) must be saved to the stack 3 times.
When calling B save A.
When calling D save C.
When calling F save E.
```

Question 2

Consider the following RISC-V code:

```
_start:
      li x20, 0xFC
      li x10, 15
      li x11, 24
      jal x5, fun1
      sw x22, 0(x20)    //store result of fun1's calculation to 0xFC
      ebreak

//x10, x11 are arguments to fun1
//fun1 will return the sum of its two arguments
//    if the first argument is less than 55, and -1 otherwise
fun1: li x20, 55
      addi x22, x0, -1
      bge x10, x20, done
      add x22, x10, x11
done: jr x5
```

Rewrite the code – and update the comments, if needed – to follow the RISC-V register usage conventions. **You may not replace x20 with a different register in `_start` or `fun1`.** Write your corrected code in the box below:

```
_start:
      li x20, 0xFC       // load address to store result
      li x11, 15         // load first arg
      li x12, 24         // load second arg
      jal ra, fun1       // jump to function
      sw x10, 0(x20)     //store result of fun1's calculation to 0xFC; using
return value register
      ebreak

//x10, x11 are arguments to fun1
//fun1 will return the sum of its two arguments
//    if the first argument is less than 55, and -1 otherwise
fun1: addi sp, sp, -4 // move stack pointer
      sw x20, 0x0(sp) // callee-save x20 before function call
      li x20, 55      // x20 = 55
      addi x10, x0, -1 // x10 = -1
      bge x11, x20, done //arg1 >= 55; skip next
      add x10, x11, x12 // x10 = arg1+arg2
done: lw x20, 0x0(sp) // callee-restore x20 after function logic
      addi sp, sp, 4 // move stack pointer
      jr ra          // jump back to main code

// What we did
   1. Stored and removed x20 from the stack to save it
   2. Changed argument registers to x10 -> x11 and x11 -> x12 to free up
      x10 to use for return value
   3. changed x22 to x10
   4. changed jal x5 to ra the return address register
```

Problem 2: Pipelining

Question 1

a. Show how the following instructions flow through a 5-stage RISC-V pipeline.
   -Use F, D, E, M, W as the pipeline stage symbols.
   -Indicate stalls by repeating the stage where the hazard is detected. For example, if an instruction gets held in Fetch, that instruction has duplicate Fs (as done in class).
   -Type the answer in the pipe traces below. (If you would like, you can neatly fill a pipe trace by hand and upload, replacing the table in the box below.)
   -You may not need all columns in the pipe trace.
   -Assume forwarding is enabled in this pipeline to the full extent possible (as discussed in class).

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A:	`lw x11, 0(x1)`	F	D	E	M	W															
B:	`lw x12, 8(x1)`		F	D	E	M	W														
C:	`add x3, x11, x12`			F	D	D	E	M	W												
D:	`add x3, x11, x3`				F	F	D	E	M	W											
E:	`sw x3, 0(x5)`						F	D	E	M	W										
F:	`add x5, x5, x3`							F	D	E	M	W									
G:	`addi x9, x9, 18`								F	D	E	M	W								
H:	`lw x16, 0(x5)`									F	D	E	M	W							
I:	`blt x16, x9, A`										F	D	D	E	M	W					

b. What is the CPI of the loop (i.e., instructions A to `I`)? Ignore the fill time for the pipeline and the loop exit, i.e., focus on the CPI of the loop when the pipeline is full and as the loop is iterating. Tip: you don't need to think about what this loop does (it's nothing useful/interesting) or how many iterations it completes; recall in class how we defined ideal pipeline CPI and how we calculated the CPI for a pipe trace that deviated from the ideal.

CPI $\frac{11\ cycles}{9\ inst} \approx 1.22$

c. How many instruction pairs rely on forwarding to minimize pipeline stalls that may arise as a result of data dependencies? Explain your answer by listing which pairs of instructions are involved and which register's contents is received via forwarding, for example, "X forwards to Y (into register x3)". Make sure to include any cases where an instruction receives two values from forwarding.

```
There are 4 pairs that rely on forwarding:
   1. B forwards to C (into register x12)
   2. C forwards to D (into register x3)
   3. D forwards to E (into register x3)
   4. D forwards to F (into register x3)
   5. F forwards to H (into register x5)
   6. H forwards to I (into register x16)
```

d. Could this code be written differently to improve CPI? If so, how / why, and what is the new CPI? If not, why not?

```
Change
C: add x3, x11, x11
D: add x3, x3, x12.
This way there is no stall at C waiting for x12

Swap
G and H
This way x16 is loaded 1 cycle earlier and blt will not stall. Additionally
with forwarding from F, there will not be a stall to H.
```

Question 2

Consider a 5-stage pipeline with the following minimum times to complete the *logic* within the given pipe stage:

F: 200 ps
D: 80 ps
E: 100 ps
M: 180 ps
W: 90 ps

An *additional*, fixed, 20 ps is required to write the result of that logic into the pipe register. For example, when an add instruction executes in the ALU it takes 100 ps, and it takes 20 ps longer to write the result of the add into the pipe register. Assume the WriteBack (W) stage already includes the time to write the register file in the 90 ps. The cycle time for the machine will be set based on these delays; think back to our discussion on how the cycle time is determined in a pipelined machine.

To try to improve performance, you have decided to break up one of the 5 stages into 2 shorter stages. In other words, you are going to increase the depth of your pipeline. You should assume that any new stage would require half of its original time plus a 5% increase. For example, if pipe stage X takes 500 ps and you split it into Y and Z, each will take ((500/2)*1.05) = 262.5 ps.

a. What is the cycle time of the original machine? Show your work/explain your reasoning.

```
The longest stage, fetch, takes 200 + 20 ps
So, the cycle time of the original machine is 220 ps
```

b. Which pipe stage will you break up, and why? What are the lengths of the two new stages? Show your work/explain your reasoning.

I broke up the fetch stage because it is the longest:
$F_1 = F_2 = 200/2 \times 1.05 + 20 = 125$ ps

c. What is the new clock cycle time for the machine with the split pipe stage? Show your work/explain your reasoning.

```
The new clock cycle time is 180 + 20 ps because the memory stage is now the
longest at 200 ps
```

Problem 3: Sorting the Linked List in RISC-V
For this problem, you will complete your program to sort the doubly linked list. As in previous assignments, you do **not have to follow the register usage conventions.**

Functions and Requirements
1. Building the list
   a. You should use the code from HW02 to build the linked list via repeated calls to the insert function.
   b. You may use your HW02 code, or the code from the solution.
2. Functions: `swap`, and `delete`
   a. You will call the functions from HW03 to swap elements as part of the sorting algorithm. Once the list is sorted, you will scan the list to delete some elements.

   b. You may use your HW03 code, or the code from the solution.
3. You will write an **`insertionSort`** function to sort the items in the list. Below is some simple (array-based) pseudocode. (You can do a Google search to find alternative pseudocode if you'd like, look at examples on the Wikipedia page, etc.).
   a. Your `insertSort` function must call the `swap` function to swap elements.
      i. In other words, don't in-line the `swap` function as is often done.
   b. This function should take the head of the linked list as an argument.

```
i ← 1
while i < length(A)
    j ← i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j ← j - 1
    end while
    i ← i + 1
end while

https://en.wikipedia.org/wiki/Insertion_sort
```

4. *Important:* you must not treat the linked list like an array. Instead, use the next/previous pointers to access adjacent list elements, to track "i" and "j", and to test for head/tail. Some examples for how the array-based pseudocode can work for a linked list:
   a. Rather than looking at "`i < length(A)`", you should look at whether you have reached the tail (the tai's next pointer is always 0). Similarly, you can test when you reach the head rather than looking at j > 0.
   b. Instead of setting "i ← 1", you can initialize "i ← (head→next)"; "i ← i + 1" thus becomes "i ← i → next".
   c. Note the arguments to swap are unchanged from HW03, so when calling "swap A[j] and A[j-1]", the address of the first or "leftmost" element involved in the swap should be passed (A[j - 1] in this case).

5. *After* calling your `insertionSort` function, you should call a function called `deleteTheLesser` to traverse the sorted list and delete all data entries whose values are less than a given value.
   a. This function should take as arguments:
      i. The (new/current) head of the linked list,
      ii. The cutoff value – e.g., if a cutoff value of 0x7000 is passed in, all elements with *value* 0x0000 - 0x6FFF will be deleted. Assume values are unsigned. You may also assume that the largest value stored in a linked list element is 0xFFFF (65535).
   b. `deleteTheLesser` must call `delete` when deleting an element.

6. As in HW02-HW03, use a dedicated register to keep track of the **original** head of the linked list throughout the program (or stash it in memory at a known location, or on the

stack). You will use this **original list head pointer** when printing the list so the graders can see the entire range of memory from 0x1204 onward.

7. Tip: the requirements state that `insertionSort` calls `swap`, and `deleteTheLesser` calls `delete`, so think carefully about how to handle the return address when you use `jal`.

Testing your Code
As noted above, your HW04 code should be integrated with the code from HW02-HW03 – either your code, or from the solution. Write test code in `_start` (“`main`”) to call `insertionSort` followed by `deleteTheLesser(0x6789)` on the same list from HW02/HW03. You are encouraged to test with additional test cases, but this is the only required one.

Problem 2 Deliverables (Screenshots)

RISC-V assembly program, Printing the List, Screenshot Output

*First, copy your assembly code with test cases to one group member’s dropbox on the student machines. This should **not** include any of the code to print the list to the terminal.*
*List the **full path and filename:***

```
/escnfs/home/awang27/esc-courses/sp25-cse-30321.01/dropbox/HW04/sort.s
```

*In addition, **copy and paste** the entirety of the code in the box below.*
*This should **not** include any of the code to print the list to the terminal.*

```
_start:
      jal x20, create_linked_list
      li x2, 0x1204
      jal x1, insertSort
      li x8, 0x1204
      mv x10, x2
      li x2, 0x6789
      jal x1, deleteTheLesser
      ebreak
//HW04 code
//arg: Head in x2
insertSort:
      lw x3, 0x8(x2) // pointer = head.next

      li x19, 0x12
insert_outer_loop:
      beq x3, x0, end_sort // leave loop if pointer is null; aka end of
list
      mv x4, x3 // cur = pointer
      lw x5, 0x0(x4) // cur_prev = cur.prev
      lw x8, 0x8(x3) // next = pointer.next
```

```
insert_inner_loop:
      beq x5, x0, end_cur_sort
      lw x6, 0x4(x4) // x6 = cur.val
      lw x7, 0x4(x5) // x7 = cur_prev.val
      bge x6, x7, end_cur_sort
      //load argument for swap function
      mv x12, x5
      li x10, 0x0
      jal x20, swap
      // if head pointer is switched: set new head
      beq x10, x0, skip_set_head
      mv x2, x10
skip_set_head:
      lw x5, 0x0(x4) // cur_prev = cur.prev
      beq x0, x0, insert_inner_loop
end_cur_sort:
      mv x3, x8 // cur = cur_next
      beq x0, x0, insert_outer_loop
end_sort:
      jr x1

//arg: values less than to delete.
deleteTheLesser:
      beq x10, x0, end_deleteTheLesser
      lw x3, 0x4(x10)
      bge x3, x2, end_deleteTheLesser
      mv x12, x10
      jal x20, delete
      beq x0,x0, deleteTheLesser
end_deleteTheLesser:
      jr x1



//HW02 Code
create_linked_list:
      la x4, num_elements
      lw x2, 0x0(x4) # num elements
      la x4, array_data # address of array_data
      addi x3, x4, 0x200 # pointer to head of list
      li x5, 0x0 # for loop i=0
      or x8, x4, x4 # address of array element to insert
      or x9, x3, x3 # address where list element is created
      li x10, 0x0 # address of last list element / pointer to tail
insertLoop:
      beq x2, x5, end_create_linked_list # if len == i end loop
      jal x12, insert # call func insert
      addi x5, x5, 1 # update counter
      addi x8, x8, 0x4 # p = p+i; pointer for array of words increased
      or x10, x9, x9 # prev = curr
      addi x9, x9, 0x40 # curr = next element creation address
      j insertLoop # jump to top of loop
insert:
      lw x11, 0x0(x8) # set val to the word
      sw x11, 0x4(x9) # this.val = val
      sw x0, 0x8(x9) # this.nex = 0
      sw x10, 0x0(x9) # this.prev = prev
      beq x5, x0, first # if first element, dont do following
      sw x9, 0x8(x10) # prev.next = this
      jr x12 # jump back to part in loop
first:
      jr x12 # jump back to part in loop
end_create_linked_list:
      jr x20
//HW03 Code
swap:
      // need argument at x12
      // new head returned at x10
      lw x11, 0x0(x12) // prev
      lw x13, 0x8(x12) // next
      beq x13, x0, end_swap // if next is null, no swap to be done return
      lw x14, 0x8(x13) //next_next
      beq x11, x0, skip_head //if prev is null (aka head) skip prev.next =
next
      sw x13, 0x8(x11) //prev.next = next
skip_head:
      beq x13, x0, skip_tail //if next_next is null (aka adjacentis tail)
skip next_next.prev = cur
      sw x12, 0x0(x14) //next_next.prev = cur
skip_tail:
      sw x12, 0x8(x13) // next.next = cur
      sw x14, 0x8(x12) // cur.next = next_next
      sw x13, 0x0(x12) // cur.prev = next
      sw x11, 0x0(x13) // next.prev = prev
      bne x11, x0, end_swap // if prev is not null we good
      or x10, x13, x13 // else head = next
end_swap:
      jr x20 // changed this cause i didnt wanna stack pointer and all


delete:
      // need element to delete at x12
      // need head at x10
      lw x11, 0x0(x12) // prev
      lw x13, 0x8(x12) // next
      bne x11, x0, del_not_head //branch if prev is not null
      or x10, x13, x13 // since prev null, head = next
      beq x0, x0, del_tail //skip to next part
```

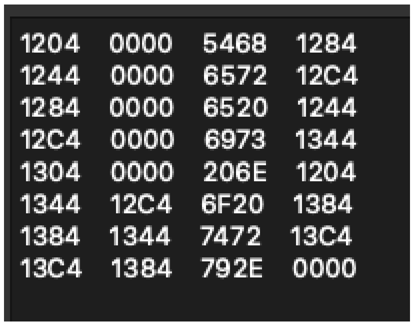
```
del_not_head:
      sw x13, 0x8(x11) // otherwise prev.next = next
del_tail:
      beq x13, x0, end_delete // if next is null return
      sw x11, 0x0(x13) // otherwise next.prev = prev
end_delete:
      jr x20
```

To demonstrate your code works, combine it with the same skeleton code from HW02/03 that will print the contents of the linked list for each test case. Instructions are here: *Using linked_list_print_skeleton.S* .

*List the **full path and filename:***

```
/escnfs/home/awang27/esc-courses/sp25-cse-30321.01/dropbox/HW04/print_sort.
s
```

Copy a screenshot of the terminal window from QtRVSim showing the final updated list:



```
1204  0000  5468  1284
1244  0000  6572  12C4
1284  0000  6520  1244
12C4  0000  6973  1344
1304  0000  206E  1204
1344  12C4  6F20  1384
1384  1344  7472  13C4
13C4  1384  792E  0000
```

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 (15 points)

	Deliverable	Points
1.	Question 1	5
2.	Question 2	10

Problem 2 (35 points)

	Deliverable	Points
1a	Filled pipe trace	8
1b	CPI of the loop	4
1c	Forwarding pairs	4
1d	Potential improvement (and new CPI if possible)	4
2a	Original cycle time	5
2b	Which stage to break up, why, and new lengths	5
2c	New cycle time	5

Problem 3 (55 points)

	Deliverable	Points
1.	RISC-V assembly program	40
2.	Screenshot of updated list (sorted, elements < 0x6789 deleted)	15