CS3220 - Processor Design
Final exam Fall 2015

The exam is due Wednesday December 9th at 2:50pm in the regular classroom (Instr Center 213). No late submissions will be accepted.

If a question is unclear or certain details are not specified, make an assumption and justify it. Any question regarding the exam, shall only be asked by direct email to the instruction (email: hadi@cc.gatech.edu). Please be mindful that the instructor is not obligated to respond immediately. Please also avoid inundating the instructor with email. **Piazza discussions about the exam are prohibited.**

In every sheet that you turn as your answers, you need to write your name and GTID. We will not correct the sheets that do not have name and GTID. Please staple all the answer sheets together.

This is an individual exam. You are free to use all resources available to you, including the web, but there is no collaboration with any other people. Sharing or discussing answers will be considered an honor code violation and will be reported to the Dean's office.

Submission instructions:

1. On t-square, submit the Quartus project folder in zip format named as final_exam-FistName_LastName.zip.
2. In class, hand-in the printed output of the test bench. We will provide you with the test bench. The printed output cannot exceed 10 pages.
3. In class, hand-in a brief report of your design. Explain the limitations if there are any, and design choices you made. This report should not exceed 1 page.

Question-Neural Processor - 100 points (with Potential 50 points extra if fixed point is supported throughout the design)

General design

The ISA and the general design of the neural processor is provided in this document. Your task is to implement a neural processor that support the described ISA in Verilog and simulate it with the provided test bench.

The following list describes the modules that you need to implement. You shall use the names that are provided below for both modules and their instances. The naming is very very important so that we can grade your exam. If you use different names, you are probably not going to pass the test bench and you will not get any credits. At the end of the document, we have provided the interfaces for some of the modules and their description.

- **Main top module:** instantiates all the other modules
  - Module name: neural_processor
  - Instance name: neural_proc
- **Controller:** Handles internal states and produces all the control signals
  - Module name: neural_controller
  - Instance name: controller

- **Instruction Memory:** Memory where the instructions will be stored. Memory of size 1024.
  - Module name: instruction_memory
  - Instance name: instr_mem
  - Memory name within the module: data_array
- **Program counter:** PC is initialized to the (number of instruction - 1) in the instruction memory and decrements until reaches zero. When PC reaches zero the program is finished.
  - Module name: counter
  - Instance name: pc
  - Register name within the module: cur_pc
- **Input FIFO**: Lifetime queue of size 256. This queue is initialized with the neural network's input values, where each input value's lifetime is equal to the number of neurons in the first hidden layer. For instance, if the neural network has 3 inputs and 2 neurons in the first hidden layer, then the input queue is initialized with 3 values all of which has a lifetime of 2.
  - Module name: fifo
  - instance name: input_fifo
  - Memory name within the module: data_array
- **Work fifo**: Lifetime queue of of depth 256.
  - Module name: fifo
  - Instance name: work_fifo
- **Macc fifo:** Lifetime queue of depth 128.
  - Module name: fifo
  - Instance name: macc_fifo
- **Output fifo**: Lifetime queue of of size 256.
  - Module name: fifo
  - Instance name: output_fifo
- **Sigmoid:**
  - Module name: sigmoid
  - Instance name: transfer_function
- **MACC**:
  - Module name: macc
  - Instance name: multadd

Grading

You will be provided a test bench that will initialize the instruction memory and input_fifo. This is why it is very important that you follow the naming convention. The test bench will then proceed to print debugging statements while executing your design. We will match your output to a reference output and grade you based on this.

ISA: In both cases, the instructions are 32 bits

ISA for Integer Neural Processor (100 % credit):

```
i: {fmt: "imm_integer",iword:"00 00000000000000 imm[15:0]"}
m: {fmt: "imm_integer",iword:"01 00000000000000 imm_int[15:0]"}
a: {fmt: "imm_integer",iword:"10 00000000000000 imm[15:0]"}
f: {fmt: "",           iword:"11 000000000000000000000000000000"}
```

ISA for Fixed-Point Neural Processor (potentially 150 % credit):

```
i: {fmt: "imm_integer",iword:"00 00000000000000 imm[15:0]"}
m: {fmt: "imm_fixedpt",iword:"01 00000000000000 imm_int[5:0]
imm_fraction[9:0]"}
a: {fmt: "imm_integer",iword:"10 00000000000000 imm[15:0]"}
f: {fmt: "",           iword:"11 000000000000000000000000000000"}
```

Instruction i. Performs initialization. The immediate integer value represents the number of neurons in the next layer. Moreover, instruction i, indicates that computation of the current layer is finished and the computation of the next layer is beginning.

Instruction m. Performs multiply and add. The immediate value will be fixed-point or integer depending on your design. Further, the immediate value represents the weight of the edge.

Instruction a. Performs the sigmoid function. Its immediate value specifies the lifetime of the value that will be enqueued in the work_fifo if the immediate is greater than zero. If the immediate is zero, the output of the sigmoid unit will be enqueued to the output fifo with the lifetime of 1.

Instruction f. Finishes the computation.

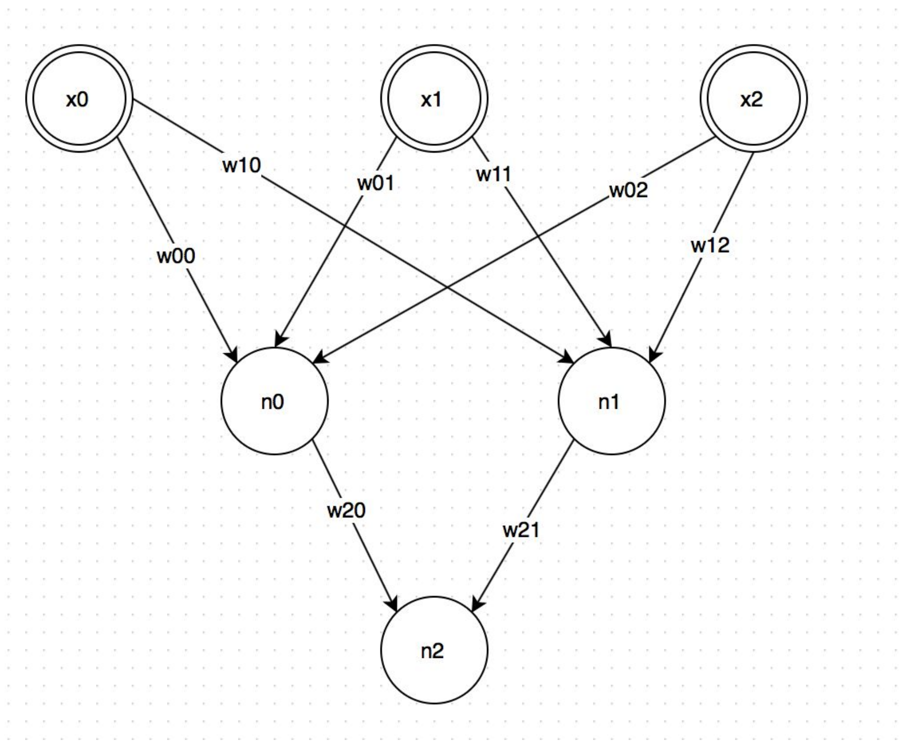
Pseudocode

We describe the details of the design with pseudocode. This is provided to point you in the right direction. The pseudocode assumes the lifetime queues internally handle the removal of entries that have reached a lifetime of 0.

```
def i_instr(imm):
      pc = pc - 1
      macc_fifo.enqueue(0, imm)

def m_instr(imm):
      pc = pc - 1
      if input_fifo.empty()
            temp = work_fifo.dequeue()
      else
            temp = input_fifo.dequeue()

      macc_fifo.enqueue(macc_fifo.dequeue() + (imm * temp), 1)

def a_instr(imm):
      pc = pc - 1
      if imm == 0:
            output_fifo.enqueue(sigmoid(macc_fifo.dequeue()), 1)
      else:
        work_fifo.enqueue(sigmoid(macc_fifo.dequeue()), imm)

def f_instr():
    pc = pc -1
    wait_forever()
```

Example

The following example demonstrates the operation of the neural processor. The computation starts at the top of the network, and proceeds downwards.



Assume that the instructions are already loaded in the instruction memory, and the input_fifo looks like this:

value	lifetime
x0	2
x1	2
x2	2

Where x0 is the top of the queue, and x1 is the next input element and x2 is that last input element.

The following instructions are the ones required to compute the output of the neural network. We explain the internal state after executing each instruction to provide you with some intuition:

- **i 2**
    - Enqueues {0, 2} to macc_fifo.

- **m w00**
    - Decrease lifetime of x0 in input_fifo
    - temp = x0
    - a = w00*x0 + 0
    - enqueue {a, 1} to macc_fifo.
- **m w10**
    - Decrease lifetime of x0 in input_fifo (which should be removed because its lifetime is now 0)
    - temp = x0
    - b = w10*x0 + 0
    - enqueue {b, 1} to macc_fifo.
- **m w01**
    - Decrease lifetime of x1 in input_fifo
    - temp = x1
    - c = w01*x1 + a
    - enqueue {c, 1} to macc_fifo.
- **m w11**
    - Decrease lifetime of x1 in input_fifo (which is now 0, remove)
    - temp = x1
    - d = w11*x1 + b
    - enqueue {d, 1} to macc_fifo.
- **m w02.**
    - Decrease lifetime of x2 in input_fifo
    - temp = x2
    - e = w02*x2 + c
    - enqueue {e, 1} to macc_fifo.
- **m w12**.
    - Decrease lifetime of x2 in input_fifo (which is now 0, remove)
    - temp = x2
    - f = w12*x2 + d
    - enqueue {f, 1} to macc_fifo.

    - In addition, at this point, the macc_fifo contains e and f only both with a lifetime of 1.
    - input_fifo is now empty
- **a 1.**
    - Dequeue e from macc_fifo.
    - g = sigmoid(e)
    - enqueue {g, 1} to work_fifo
- **a 1.**
    - Dequeue f from macc_fifo (now empty).
    - h = sigmoid(f)
    - enqueue {h, 1} to work_fifo.
- **i 1**
    - Enqueue {0, 1} to macc_fifo
- **m w20**
    - Dequeue g from work_fifo
    - temp = g.
    - i = w20*g + 0
    - Enqueue {i, 1} to macc_fifo

- **m w21**
    - Dequeue h from work_fifo
    - temp = h; work_fifo is now empty.
    - dequeue i from macc_fifo; macc_fifo is now empty.
    - j = w21*h + i
    - Enqueue {j, 1} to macc_fifo.
- **a 0**
    - Dequeue j from macc_fifo. macc_fifo is now empty.
    - Let k = sigmoid(j). Enqueue {k} to output_fifo.
- **f**
    - Quite easily done :-).

The variables defined within the explanation of the internal state (e.g. a, b, c) are only for illustrative purposes. These variables do not need to exist in the actual design.

At the end of these instructions, output_fifo should have a single entry with the result.

Lifetime FIFO queue module
You must design a verilog module that works as a regular FIFO queue, with the addition that it should have lifetime semantics. That is, every element in the queue is a value-lifetime tuple. Each data element is 48-bits, where bits [47:16] store the value and bits [15:0] store the lifetime. In a regular queue without lifetime semantics, when an element is dequeued, it is removed from the queue. In a lifetime queue, when an element is dequeued, that element's lifetime is decremented by one, and only if the lifetime is 0 the element is removed from the queue.

Start (head = 0, tail = 0)		**Enqueue {42, 3} (head = 0, tail = 1)**		**Enqueue {27, 1} (head = 0, tail = 2)**		**Dequeue (head = 0, tail = 2)**		**Dequeue AND Enqueue {3, 4} (head = 0, tail = 3)**		**Dequeue (head = 1, tail = 3)**		**Dequeue (head = 2, tail = 3)**	
FIFO QUEUE		**FIFO QUEUE**		**FIFO QUEUE**		**FIFO QUEUE**		**FIFO QUEUE**		**FIFO QUEUE**		**FIFO QUEUE**	
Val	**Life**	**Val**	**Life**	**Val**	**Life**	**Val**	**Life**	**Val**	**Life**	**Val**	**Life**	**Val**	**Life**
								3	4	3	4	3	4
				27	1	27	1	27	1	27	1		
		42	3	42	3	42	2	42	1				

```
module fifo(clk, reset, enqueue, dequeue, data_in, data_out, full,
          empty);
```

```
parameter ADDR_LEN = 11;
parameter DATA_WIDTH = 48;
input clk, reset, enqueue, dequeue;
input[DATA_WIDTH - 1:0] data_in;
output[DATA_WIDTH - 1:0] data_out;
output full, empty;
```

However, the internal implementation details are up to you. You might use two separate internal data structures to store values and lifetimes, or you might use a single combined data structure.

Signals description:

- clk: input clock.
- reset: if high, initialize module and don't do anything else
- enqueue: on a clock edge, if enqueue is high, enqueue data_in
- dequeue: on a clock edge, if dequeue is high, perform dequeue
- data_in: 32 bits input. The 16 MSB correspond to the value, and the 16 LSB correspond to the lifetime of the entry.
- data_out: always set to the data at the head of the FIFO.

MACC module
You must satisfy the following interface:

```
module macc(input a, input b, input c, output o)
```

You must compute o = a * b + c

- In the integer neural processor, all signals are 16-bit 2's complement values.
- For the fixed point neural processor, all signal (**including the output**) are 16-bit 2's complement fixed values, with 6 bits for the integer part and 10 bits for the fraction.

Sigmoid module
You must satisfy the following interface:

```
module sigmoid(x, y)
```

- In the integer neural processor:
    - if $x \geq 4 \Rightarrow y = 1$
      if $x < -4 \Rightarrow y = 0$
      else $y = x >> 3$ (signed shift to the right by three bits)

- In the fixed-point neural processor:
    - if $x \geq 4.0 \Rightarrow y = 1.0$
      if $x \leq -4.0 \Rightarrow y = 0.0$
      else $y = 0.125x$

x is the input, y is the output. Both are represented as 16-bit signed, fixed-point values where 6 bits are to the left of the decimal point (integer part) and 10 bits are to the right (fraction part).