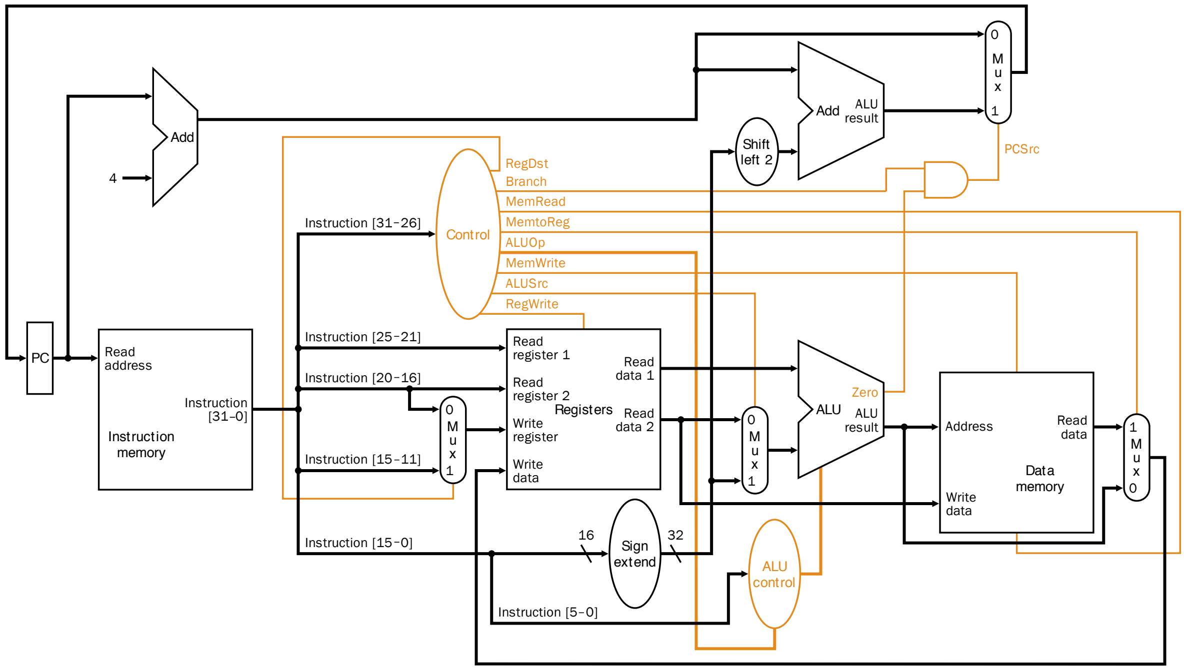
1.2 Single-Cycle Processor Datapath [10 points]

Modify the single-cycle processor datapath to include a version of the `lw` instruction, called `lw2`, that adds two registers to obtain the effective address. The datapath that you will modify is provided below. Your job is to implement the necessary data and control signals to support the new `lw2` instruction, which we define to have the following semantics:

`lw2`: Rd $\leftarrow$ Memory[Rs + Rt]
PC $\leftarrow$ PC + 4

Add to the datapath any necessary data and control signals (if necessary) to implement the `lw2` instruction. Draw and label all components and wires very clearly (give control signals meaningful names; if selecting a subset of bits from many, specify exactly which bits are selected; and so on).



ALU opcode	Operation
00	Add
01	Subtract
10	Controlled by `funct`
11	Not used

There is no need for new components and wires. The main difference is that the ALU must use "Read data 2", instead of the output of the sign extend unit. The new `lw2` will be R-type, not I-type.
The values of the control signals need to be:
RegDst = 1;
ALUScr = 0;
MemtoReg = 1;
RegWrite = 1;
MemRead = 1;
MemWrite = 0;
ALUop = 00;
Branch = 0.

1.3 Performance Evaluation [10 points]

The execution time of a given benchmark is 100 ms on a 500 MHz processor. An ETH alumnus, designing the next generation of the processor, notices that a new implementation enables the processor to run at 750 MHz. However, the modifications increase the CPI by 20% for the same benchmark.

(a) [4 points] What is the execution time expressed in terms of the number of cycles taken for the **old** generation of the processor (i.e., before the modifications)?

Assuming that the IPC is 2, what is the number of instructions in the benchmark?

Answer: Execution time is **50 Million cycles**. The benchmark has **100 Million instructions**.

Explanation:
Clock frequency is 500 MHz. Then each cycle takes $1/(500 \times 10^{-6}) = 2ns$.
Total execution time in cycles is $100ms/2ns = 50Million$ cycles.

2 instructions per cycle. Then, the total number of instructions: $2x50M = 100M$

(b) [3 points] What is the execution time of the benchmark in *milliseconds* for the **new** generation of the processor?

Answer: 80 ms.

Explanation:
$Execution\ Time = [Number\ of\ Instructions] \times [CPI] \times [Frequency^{-1}]$
Let's say that the CPI of baseline is c, and number of instructions is i.
Then the execution time of baseline:
$(c \times i)/(500x10^6) = 100x10^{-3}\ seconds \quad => \quad (c \times i) = 5 \times 10^7$

The execution time after modifications: $((1.2 \times c) \times i)/(750x10^6)$
$T = ((1.2 \times (c \times i))/(750 \times 10^6)\ seconds.$
$T = ((1.2 \times (5 \times 10^7))/(750 \times 10^6)\ seconds.$
$T = 8 \times 10^{-2} = 80ms.$

(c) [3 points] What is the speedup or slowdown of the new generation processor *over* the old generation?

Answer: 25% speedup

Explanation:
$Speedup\ =\ (OldExecutionTime\ /\ [NewExecutionTime]) - 1$
$Speedup\ = 100/80 - 1$
$Speedup\ = 0.25$

Then the modification introduces 25% speedup.

2 Verilog

Please answer the following four questions about Verilog.

(a) [6 points] Does the following code result in a sequential circuit or a combinational circuit? Explain why.

```
module concat (input clk, input data_in1, input data_in2,
                                  output reg [1:0] data_out);
 always @ (posedge clk, data_in1)
   if (data_in1)
      data_out = {data_in1, data_in2};
   else if (data_in2)
      data_out = {data_in2, data_in1};
endmodule
```

Answer and concise explanation:

Sequential circuit.

Explanation.
This code results in a sequential circuit because `data_in2` is *not* in the sensitivity list, and thus a latch is inferred for `data_out`.

(b) [6 points] In the following code, the input `clk` is a clock signal. What is the hexadecimal value of the output `c` right after the third positive edge of `clk` if initially `c = 8'hE3` and `a = 4'd8` and `b = 4'o2` during the entire time?

```
1 module mod1 (input clk, input [3:0] a, input [3:0] b, output reg [7:0] c);
2 always @ (posedge clk)
3   begin
4     c <= {c, &a, |b};
5     c[0] <= ^c[7:6];
6   end
7 endmodule
```

Please answer below. Show your work.

8'hC4.

Explanation.
Cycle 1: c <= {c, &a, |b} → c <= {1110_0011, 0, 1} → c <= {1000_1101}
c[0] <= ^c[7:6] → c[0] <= ^{11} → c[0] <= 0
At the first positive edge of clk, $c = 8'b1000\_1100$
Cycle 2: c <= {c, &a, |b} → c <= {1000_1100, 0, 1} → c <= {0011_0001}
c[0] <= ^c[7:6] → c[0] <= ^{10} → c[0] <= 1
At the second positive edge of clk, $c = 8'b0011\_0001$
Cycle 3: c <= {c, &a, |b} → c <= {0011_0001, 0, 1} → c <= {1100_0101}
c[0] <= ^c[7:6] → c[0] <= ^{00} → c[0] <= 0
At the third positive edge of clk, $c = 8'b1100\_0100 \rightarrow c = 8'hC4$

Note that since the assignments to c are non-blocking, $c[7:6]$ in line 5 is not affected by the assignment to c in line 4 in the same cycle.

(c) [6 points] Is the following code syntactically correct? If not, please explain the mistake(s) and how to fix it/them.

```
module 1nn3r ( input [3:0] d, input op, output[1:0] s);
  assign s = op ? (d[1:0] - d[3:2]) :
                        (d[3:2] + d[1:0]);
endmodule

module top ( input wire [6:0] instr, input wire op, output reg z);

  reg[1:0] r1, r2;

  1nn3r i0 (.instr(instr[1:0]), .op(instr[7]), .z(r1) );
  1nn3r i1 (.instr(instr[3:2]), .op(instr[0]), .z(r2) );
  assign z = r1 | r2;

endmodule
```

Answer and concise explanation:

The code is not syntactically correct.

Explanation.

- Module names cannot start with a number → '1nn3r' is not a legal module name.
- The output signal 'z' has to be declared as a 'wire' but not 'reg'.
- 'r1' and 'r2' has to be declared as 'wire's.
- The module '1nn3r' does not have ports named 'instr' and 'z'. Those need to be changed to 'd' and 's', respectively.

(d) [6 points] Does the following code correctly implement a counter that counts from 1 to 11 by increments of 2 (e.g., 1, 3, 5, 7, 9, 11, 1, 3 ...)? If so, say "Correct". If not, correct the code with minimal modification.

```
module odd_counter (clk, count);
  wire clk;
  reg[2:0] count;
  reg[2:0] count_next;

  always@*
  begin
    count_next = count;
    if(count != 11)
      count_next = count_next + 2;
    else
      count_next <= 1;
  end

  always@(posedge clk)
    count <= count_next;
endmodule
```

Answer and concise explanation:

No, the implementation is not correct.

Explanation.
The correct implementation:

```
module odd_counter (input clk, output count);
  wire clk;
  reg[3:0] count;
  reg[3:0] count_next;

  always@*
  begin
    count_next = count;
    if(count != 11)
      count_next = count_next + 2;
    else
      count_next = 1;
  end

  always@(posedge clk)
    count <= count_next;
endmodule
```