 Problem 1. Sequential Logic Timing (12 points)

(A) (4 points) Find the propagation delay ($t_{PD}$) and contamination delay ($t_{CD}$) of the circuit shown below, using the $t_{CD}$ and $t_{PD}$ information for the gate components shown in the table below.

<table>
<thead>
<tr>
<th>Gate</th>
<th>$t_{CD}$</th>
<th>$t_{PD}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>INV</td>
<td>0.1ns</td>
<td>1.0ns</td>
</tr>
<tr>
<td>NAND2</td>
<td>0.2ns</td>
<td>1.5ns</td>
</tr>
<tr>
<td>NAND3</td>
<td>0.3ns</td>
<td>1.8ns</td>
</tr>
<tr>
<td>XOR2</td>
<td>0.6ns</td>
<td>2.5ns</td>
</tr>
</tbody>
</table>

nand2 to nand3 $t_{CD} = 0.5 \text{ ns}$

2 XORs $t_{PD} = 5 \text{ ns}$
(B) (8 points) The circuit above implements a full subtractor (FS), which, similar to a full adder, computes X – Y – Bin in two’s complement. You combine two FS circuits as shown below to implement a two-bit self-decrementing counter.

The table below shows the timing specifications for the D registers and FS circuit (not the same implementation as above!). Find the maximum value for the D register’s hold time, $t_{\text{HOLD}}$, and the minimum clock cycle period, $t_{\text{CLK}}$, for which the circuit operates correctly.

<table>
<thead>
<tr>
<th></th>
<th>D reg</th>
<th>FS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{\text{CD}}$</td>
<td>0.2ns</td>
<td>0.1ns</td>
</tr>
<tr>
<td>$t_{\text{PD}}$</td>
<td>0.3ns</td>
<td>4.1ns</td>
</tr>
<tr>
<td>$t_{\text{SETUP}}$</td>
<td>1.2ns</td>
<td>–</td>
</tr>
<tr>
<td>$t_{\text{HOLD}}$</td>
<td>???</td>
<td>–</td>
</tr>
</tbody>
</table>

$t_{\text{CD,R}} + t_{\text{CD,FS}} \geq t_{\text{H}}$

$0.2 + 0.1 = 0.3$

maximum value for $t_{\text{HOLD}}$: 0.3 ns

$t_{\text{CLK}} \geq t_{\text{PD,D0}} + 2t_{\text{PD,FS}} + t_{\text{S,D1}} = 0.3 + 8.2 + 1.2 = 9.7$

minimum value for $t_{\text{CLK}}$: 9.7 ns
Problem 2. Finite State Machines (12 Points)

Consider the FSM described by the transition table and diagram below. The FSM is designed to process a stream of binary digits (0,1), one digit per cycle. The FSM has a single output that controls a light, which is either ON (☼) or OFF (●). The state transition diagram is in skeletal form, showing only the output for each state and the unlabeled transitions.

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>E</td>
<td>●</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>C</td>
<td>●</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>C</td>
<td>●</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>B</td>
<td>●</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>C</td>
<td>☼</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>F</td>
<td>☼</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>C</td>
<td>●</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>B</td>
<td>●</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>A</td>
<td>☼</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>D</td>
<td>☼</td>
</tr>
<tr>
<td>F</td>
<td>0</td>
<td>C</td>
<td>●</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>D</td>
<td>●</td>
</tr>
</tbody>
</table>

(A) (8 points) Using the state transition table as a guide, label the states in the diagram A through F. Also label each transition with either 0 or 1 to indicate which to take from each state when processing the next binary digit in the input sequence.

Add state and transition labels to diagram above

(B) (2 points) If we implemented the FSM above using the classic combinational logic + State register circuit (shown below), what would be the value of N?

Value of N: __3___

(C) (2 point) Assume one knows nothing about the current state of the FSM. Is there a binary input sequence guaranteed to turn on the light after the final digit of the sequence has been entered? Either give the sequence or write “NONE”.

Binary sequence to turn on light or NONE: __10____
Problem 3. Sequential Circuits (14 points)

Implement a sequential circuit to compute the sum of squares of all numbers ranging from 0 to n. For example, for n = 4, your circuit should compute:

\[ 0^2 + 1^2 + 2^2 + 3^2 + 4^2 = 30. \]

Design the circuit as a sequential Minispec module by filling in the skeleton code below. The circuit should start a new computation when a Valid input is given. When the computation is finished, the getResult method should return a Valid result; while the computation is ongoing, getResult should return Invalid. Initially (before the circuit has received any input), getResult output should always be Invalid. You can use the multiplication operator (*). * performs unsigned multiplication of Bit#(n) inputs. Assume that inputs and results are unsigned.

```
module SumOfSquares;
    Reg#(Bool) initialized(False);
    RegU#(Bit#(16)) n;
    RegU#(Bit#(16)) sum;

    input Maybe#(Bit#(16)) in default = Invalid;

    rule tick;
        if (isValid(in)) begin
            initialized <= _True__________;
            n           <= fromMaybe(? , in)__;
            sum         <= _0________________;
        end
        else if (___n > 0______) begin
            n           <= _n – 1___________;
            sum         <= _sum + n * n___________;
        end
    endrule

    method Maybe#(Bit#(16)) getResult =
        (_initialized && n == 0__) ? Valid(sum) : Invalid;

endmodule
```
Problem 4. Pipelining Combinational Circuits (18 points)

For each of the questions below, please create a valid K-stage pipeline of the given circuit. Each component in the circuit is annotated with its propagation delay. **Show your pipelining contours** and place large black circles (●) on the signal arrows to **indicate the placement of pipeline registers**. Give the latency and throughput of each design, assuming ideal registers ($t_{pd}=0$, $t_{setup}=0$). Remember that our convention is to place a pipeline register on each output.

(A) (3 points) Show the **maximum-throughput 1-stage pipeline**.

Latency (ns): ___11_____
Throughput (ns$^{-1}$): ___1/11_____

$t_{CLK} = 11$ ns

(B) (5 points) Show the **maximum-throughput 2-stage pipeline** using a minimal number of registers.

Latency (ns): ___14_____
Throughput (ns$^{-1}$): ___1/7_____

$t_{CLK} = 7$ ns; latency = 2 * 7 = 14

(C) (5 points) Show the **maximum-throughput pipeline** using a minimal number of registers.

Latency (ns): ___12_____
Throughput (ns$^{-1}$): ___1/4_____

$t_{CLK} = 4$ ns; latency = 3 * 4 = 12
(D) (5 points) You manage to reimplement the slowest combinational component in the previous circuit (the one with a propagation delay of 4 ns) using two components with propagation delays of 2 ns, as shown right. Show the **maximum-throughput pipeline** using a minimal number of registers.

Latency (ns): ___12________

Throughput (ns⁻¹): ___1/3________

\[ t_{CLK} = 3 \text{ ns}; \text{ latency} = 4 \times 3 = 12 \]
Problem 5. Processor Implementation (24 points)

While our RISC-V processor has 32-bit registers and operates on 32-bit values, it is very common for programs to manipulate larger integer values, e.g., 64 or 128 bits long. In this problem, you will study the cost of using our 32-bit RISC-V processor to add these large integers, and extend our processor to make these additions faster.

To add large (e.g., 64-bit) numbers using 32-bit addition, we can for the most part use a sequence of 32-bit add instructions, adding large numbers in 32-bit chunks. The key issue is propagating the carry bit between these additions. Remember the design of a ripple-carry adder, shown on the right. You can build a large ripple-carry adder from two smaller ones by connect the carry-out of one to the carry-in of the other. The example on the right shows how to connect two 1-bit full adders to form a 2-bit ripple-carry adder. We will leverage this same principle, but using the 32-bit adder in our ALU.

1. (6 points) Complete the assembly code below to add two 64-bit unsigned integers and produce a 64-bit unsigned integer result. Consider the following:
   1. Each 64-bit input and output is stored in two 32-bit registers. We use the following notation: \{a1, a0\} denotes that 32-bit registers a1 and a0 represent a single 64-bit number, a1 stores the upper 32 bits, and a0 stores the lower 32 bits.
   2. The numbers to be added are in \{a1, a0\} and \{a3, a2\}, and your code should produce the result in \{a5, a4\}.
   3. The code below shows the first two instructions of the solution. This code does not work as-is because it does not propagate the carry bit from the sum of the lower 32 bits into the sum of the upper 32 bits. Your code should handle this carry bit.
   4. For full credit, your code should use a total of at most four instructions (implementations with more instructions will earn partial credit).
   5. Hint: Although the add instruction does not store the carry bit anywhere, you can find the carry with a single comparison of the destination register and any one of the source registers.

```
add a4, a2, a0
add a5, a3, a1
sltu t0, a4, a2
add a5, a5, t0
```

2. (3 points) Does your code also work on signed integers in two’s complement encoding? Briefly explain why or why not.

Yes, because addition works in the same way whether numbers are unsigned or in two’s complement representation (note the comparison to find the carry bit must be unsigned in both cases).
Dissatisfied with the performance of this code, Ben Bitdiddle proposes to extend the RISC-V ISA to store and use the carry bit. Specifically, Ben proposes the following changes:

1. Add a new 1-bit register, `carry`.
2. Add a new `addc` instruction (add-with-carry) that uses the value in the `carry` register as its carry-in, and also writes its carry-out bit to the `carry` register:

   ```
   addc rd, rs1, rs2 :
   {carry, reg[rd]} <= {1'b0, reg[rs1]} + {1'b0, reg[rs2]} + carry
   ```

3. (4 points) Write assembly code that uses `addc` to add two 64-bit unsigned integers and produce a 64-bit unsigned integer result as in question (A). **Assume the carry bit is initially 0.** For full credit, your code should use the smallest possible number of instructions.

   ```
   addc a4, a2, a0
   addc a5, a3, a1
   ```

Let’s add Ben’s ISA extensions to our single-cycle processor implementation. Assume that we have added a new IType `ADDC` as shown below, and have already modified the decode function to decode `addc` instructions appropriately.

```
typedef enum { OP, OPIMM, BRANCH, LUI, JAL, JALR, LOAD, STORE, AUIPC, ADDC, Unsupported } IType;
```

We want to modify the `execute` function to provide the carry output and update the carry register every instruction, as shown in the module below. New code is highlighted in red.

```
typedef struct {
   IType iType;
   Maybe#(RIndx) dst;
   Word data;
   Word addr;
   Word nextPc;
   Bit#(1) carry;
} ExecInst;

module Processor;
   Reg#(Bit#(1)) carry(0);
   Reg#(Word) pc(0);
   ...
   rule doProc;
   ...
   let eInst = execute(dInst, rVal1, rVal2, pc, carry);
   
   // Always update the carry bit
   carry <= eInst.carry;
   
   // Other state updates
   ...
```
4. (12 points) Modify the execute function below to implement the `addc` instruction.

```verilog
function ExecInst execute(DecodedInst dInst, Word rVal1, Word rVal2, Word pc, Bit#(1) carry);

    let imm = dInst.imm;
    let brFunc = dInst.brFunc;
    let aluFunc = dInst.aluFunc;

    Bit#(33) addcSum = {1'b0, rVal1} + {1'b0, rVal2} + carry;

    Word data = case (dInst.iType)
        AUIPC: pc + imm;
        ...
        ADDC: addcSum[31:0];
        default: 0;
    endcase;

    Word nextPc = case (dInst.iType)
        JALR: (rVal1 + imm) & ~1; // "& ~1" clears the bottom bit.
        ...
        ADDC: pc + 4;
        default: pc + 4;
    endcase;

    Word addr = case (dInst.iType)
        ...
        default: 0;
    endcase;

    Word newCarry = case (dInst.iType)
        ...
        ADDC: addcSum[32];
        default: carry;
    endcase;

    return ExecInst{iType: dInst.iType, dst: dInst.dst, data: data, addr: addr, nextPc: nextPc, eInst.carry = newCarry};
endfunction
```

Write your code below. Please add arrows to indicate where your code fits in the `execute` function.
Problem 6. Caches (20 points)

Consider a direct-mapped cache, Cache1, shown on the right. It has 8 lines with a block size of 1 (1 word per line). Assume that data and addresses are 32 bits. Also assume that the cache has a valid bit and a dirty bit per line.

1. (2 points) To ensure the best cache performance for Cache1, which address bits should be used for the block/byte offset? Which bits should be used for the cache index? Which address bits should be used for the tag field?

   address bits used for block/byte offset: A[___1___:__0___]

   address bits used for cache index: A[___4___:__2___]

   address bits used for tag field: A[___31___:__5___]

2. (2 points) Explain why your selection of bits in part (A) should result in the best cache performance.

   The lower-order bits of the address change more often, so using them as the index bits reduces conflict misses.

3. (2 points) How many bits of SRAM are required to store the contents of Cache1? Include all the state of each line, not just the data.

   Bits per line = valid + dirty + tag + data = 2 + 27 + 32 = 61.
   Total bits = 61 * 8 lines = 488.

   Bits of SRAM for Cache1: ___488_____

4. (3 points) Could the four 32 bit addresses 0x00, 0x10, 0x18, and 0x3C all be resident in the cache at the same time? Explain why you chose your answer.

   All four addresses can be resident at the same time: YES NO
Now consider **Cache2**, which also stores a total of 8 words but its configuration is a 4 line direct-mapped cache with block size of 2 (2 words per line).

5. (2 points) For **Cache2**, which address bits should be used for the block/byte offset? Which bits should be used for the cache index? Which address bits should be used for the tag field?

   address bits used for block/byte offset: A[2:0]

   address bits used for cache index: A[4:3]

   address bits used for tag field: A[31:5]

6. (3 points) Could the four 32-bit addresses 0x00, 0x10, 0x18, and 0x3C all be resident in **Cache2** at the same time? **Explain why you chose your answer.**

   0x00: index 0; 0x10: index 2; 0x18: index 3; 0x3C: index 3
   0x18 and 0x3C conflict with each other because they both map to line 3.

   All four addresses can be resident at the same time: **YES** **NO**

7. (6 points) Find the cache hit ratio for the following loop on each of the two caches. The loop is executed for many iterations, so report the average behavior of a loop iteration and ignore cold-start effects. Assume the cache is empty before execution begins (all valid bits are 0). Remember the cache is used for both instruction and data accesses.

   ```
   . = 0x100  // Code starts at address 0x100
   // Assume x1 is initialized to 0, x2 is initialized to 2000 (number of array elements), and x3 is initialized to the base address of array A.
   
   loop: addi x2, x2, -1
       slli x4, x2, 2   // x4 holds byte offset of array element x2
       add x4, x3, x4   // x4 holds address of element x2
       lw x5, 0(x4)    // x5 holds element x2 of array
       add x1, x1, x5   // x1 holds sum of array elements
       add x0, x0, x0   // dummy instruction
       add x0, x0, x0   // dummy instruction
       bnez x2, loop
   ```

   Per loop iteration: 8 instr fetches and 1 data fetch. In **Cache1**, data word keeps replacing one of the instructions so each iteration you have 7 instr hits, 1 instr miss, and 1 data miss. **Cache2 has the same behavior** (a data line replaces a line with instructions on each iteration).

   **Cache1 hit ratio:** __7/9________

   **Cache2 hit ratio:** __7/9________

**END OF QUIZ 2!**