ELEC3342: Digital System Designs
First Semester, 2021–22

Homework 3 (r1.1)

Due:

- Part (A) -- 26 Nov, 2021, 11:59pm
- Part (B) -- 26 Nov, 2021, 11:59pm

Instruction: Submit your answers electronically through Moodle.

There are 2 major parts in this homework. Part A includes questions that aim to help you with understanding the lecture materials. They resemble the kind of questions you will encounter in quizzes and the final exam. Some of these questions may also ask you to implement and test small designs in VHDL. This part of homework must be completed individually.

Part B of this homework contains a part of the project that you should work in groups of 4. Your submitted work will be graded by an auto tester and therefore you should make sure your submitted files conform to the required format.

The following summarize the 2 parts.

Part	Type	Indv/Grp
A	Basic problem set	Individual
B	Project	Group of 4 (same as HW2)

In all cases, you are encouraged to discuss the homework problems offline or online using Piazza. However, you should not ask for or give out solution directly as that defeat the idea of having homework exercise. Giving out answers or copying answers directly will likely constitute an act of plagiarism, which is a serious offense.

Part A: Problem Set

A.1 Registered Circuit

The following shows a circuit that computes a registered six-input AND function.

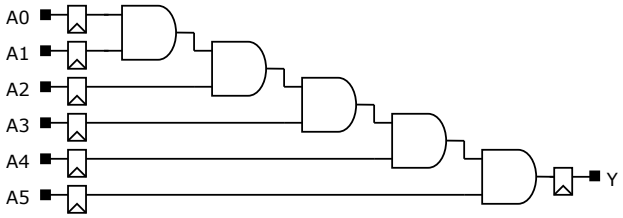


Figure A.1: A registered 6-input AND gate

Each two-input AND gate has a propagation delay of 100 ps and a contamination delay of 45 ps. Each flip-flop has a setup time of 57 ps, a hold time of 17 ps, a clock-to-Q maximum delay of 60 ps, and a clock-to-Q minimum delay of 40 ps. Determine the following:

A.1.1 If there is no clock skew, what is the maximum operating frequency of the circuit?

A.1.2 How much clock skew can the circuit tolerate if it must operate at 1.6 GHz?

A.1.3 How much clock skew can the circuit tolerate before it might experience a hold time violation?

A.1.4 Redesign the combinational logic between the registers so they perform the same 6-input AND function but are able to allow the circuit to run faster and tolerate more clock skew.

What is the maximum frequency that your improved circuit can run at? How much clock skew can it tolerate before it might experience a hold time violation?

A.2 Synchronizer MTBF

For your next design, you would like to build a synchronizer that can receive asynchronous inputs with an MTBF of 1 year. You are building the synchronizer using a sampling flip-flop with $\tau = 100$ ps, $T_0 = 110$ ps, and $t_{\text{setup}} = 70$ ps. The synchronizer receives a new asynchronous input on average 4 times per second. What is the required probability of failure to satisfy this MTBF? How long would you have to wait before reading the sampled input signal to give that probability of error? [*based on DDCA 3.37*]

A.3 Circuit Pipelining

In theory, any feedforward circuits can be sped up by pipelining the datapath. For instance, Figure A.2 shows several circuits with different pipelined datapaths that perform the same mathematical function:

$$Y = A(B + C) - BC + A - B \times B$$

The diagram shows a correctly pipelined circuit that performs the same function except it has an additional pipeline stage and thus incurs an additional cycle of latency. The diagram also shows an *incorrectly pipelined* circuit that looks similar but produces wrong answers.

Let the setup and hold time of each register to be t_{setup} and t_{hold} respectively. The register has a clock-to-Q propagation delay t_{pcq} and a clock-to-Q contamination delay of t_{ccq}. Also, let the propagation and contamination delay of an adder/subtractor be t_{pd} and t_{cd} respectively. Finally, let the propagation and contamination delay of a multiplier be $4t_{pd}$ and $4t_{cd}$

A.3.1 For each of the circuit (both (a) and (b)) in Figure A.2, determine the following:

(i) The maximum *clock frequency* that the circuit may run at without violating any of the setup and hold time constraints.

(ii) The minimum clock frequency that the circuit may run at without violation any of the setup and hold time constraints.

(iii) Throughput of the circuit.

(iv) Latency of the circuit.

Assume the values of all input arrives on every cycle, i.e., the values $A[n], B[n], C[n]$ are available at the *input* of the (leftmost) input register in cycle n where $n = 0, 1, \ldots$.

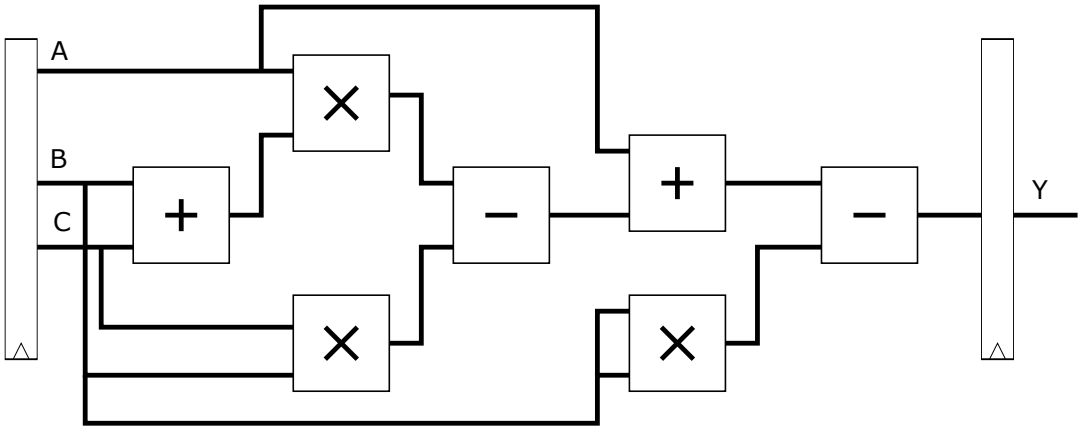
For (ii), the answer may be infinitely slow, i.e., no constraint on minimum. For (iii), processing *throughput* is measured as the number of answer ($Y[n]$) produces per second. For (iv), processing *latency* of the circuit is L if output $Y[0]$ is ready at the *output* of the final register in cycle L.

A.3.2 Explain, by tracing the number of added latency to the circuit that affect the index n, why the *incorrectly pipelined* circuit does not produce the same result?

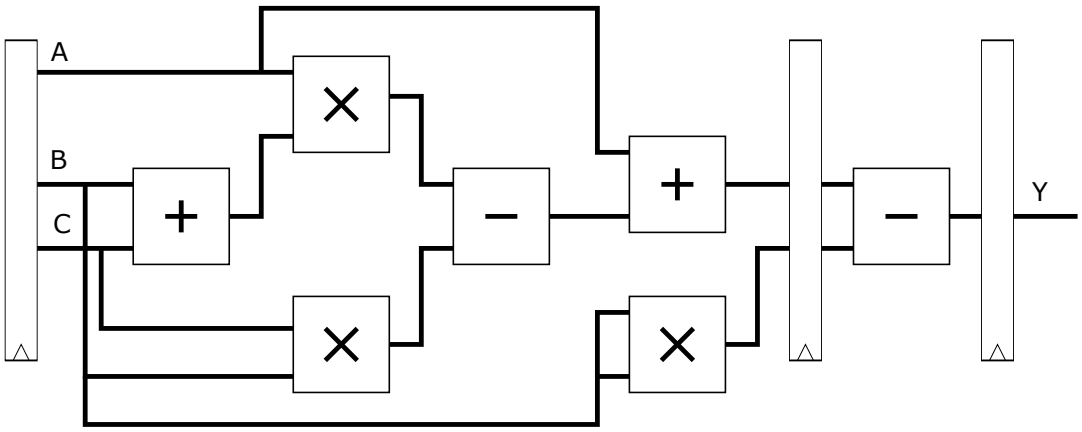
A.3.3 Redesign the circuit in Figure A.2 such that is would run with maximum processing throughput (number of results per second) while keeping the overall function unchanged. You may make zero or more of the following changes:

- Add or remove 0 or more registers
- Add or remove 0 or more Adder/Subtractor/Multiplier
- Rearrange the Adder/Subtractor/Multiplier connection

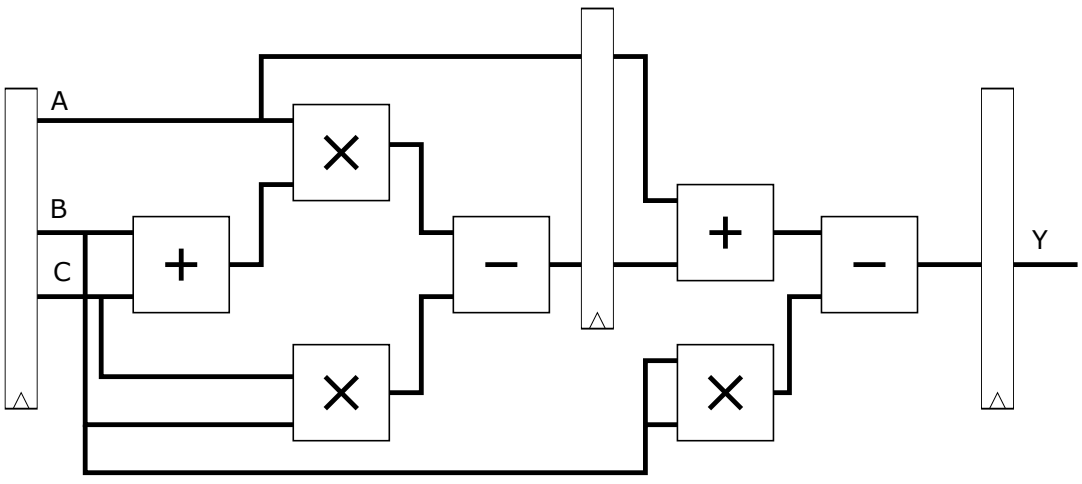
Submit a schematic of your new circuit, and determine its throughput.



(a) Original Circuit



(b) Pipelined Circuit

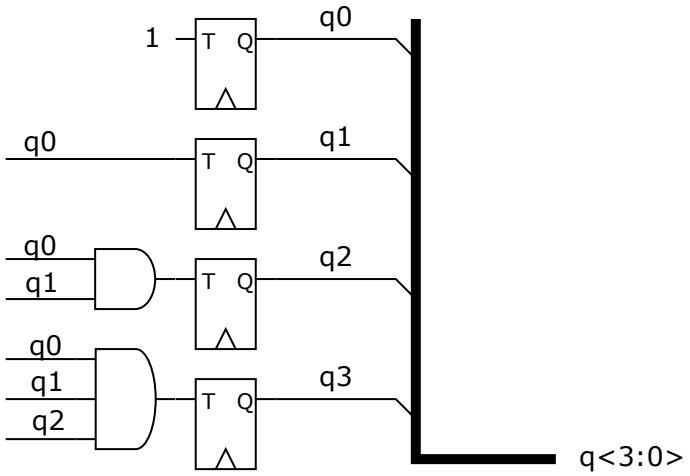


(c) Incorrectly Pipelined Circuit

Figure A.2: Pipelining a feedforward circuit. (a) original circuit; (b) with 1 pipeline register; (c) incorrectly pipelined circuit.

A.4 Counter Timing

You are evaluating the following design of a 4-bit binary counter:



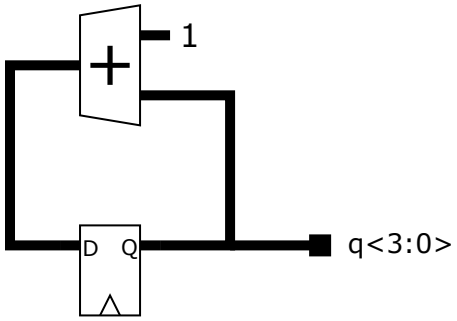
This basic design, called `CntrP` is built using 4 T-flip-flops (TFF). It is designed based on the following observation: The value of bit i of the counter (`q`i) toggles only when ALL the bits in position $i-1, i-2, \ldots, 0$ are '1'. For example, bit 2 toggles the first time during the transition `0011` $\rightarrow$ `0100`, and it only toggles back to 0 during the transition `0111` $\rightarrow$ `1000`. In both cases, bit 1 and 0 are both HIGH for bit 2 to toggle. All TFFs are connected to the same clock signal with no clock skew (clock not drawn for clarity). All TFFs reset to '0' when the system is powered up.

A.4.1 The following table lists the timing characteristics of the circuit elements that you are using to build this counter:

Parameter	**Value** (ns)	**Remark**
t_{ccq}	1	C-to-Q Contamination delay of TFF and DFF
t_{pcq}	2	C-to-Q Propagation delay of TFF and DFF
t_{hold}	1	hold time of TFF and DFF
t_{setupT}	7	setup time of TFF (for input `T`)
t_{setupD}	3	setup time of DFF (for input `D`)
$t_{\mathrm{pd,2}}$	15	Propagation delay of any 2-input gate
$t_{\mathrm{pd,3}}$	25	Propagation delay of any 3-input gate
$t_{\mathrm{pd,4}}$	45	Propagation delay of any 4-input gate
$t_{\mathrm{pd,k}}$	$5 + 10 \times 2^{k-2}$	Propagation delay of an k-input gate where $k \geq 2$
t_{cd}	5	Contamination delay of a gate regardless of number of input

What is the maximum clock frequency that the circuit `CntrP` may run at? Show where the critical path is, and show how it affects your calculation of the maximum clock frequency.

A.4.2 The following is an alternative implementation of a binary counter. Call this circuit `CntrA`.



Note the D-flip-flops are used in `CntrA`. Also, the adder in `CntrA` is a ripple-carry adder. Each full adder cell in the adder has a delay of $t_{\text{FA}} = 20\,\text{ns}$ from *any* input to *any* output.

What is the maximum clock frequency that the circuit `CntrA` may run at? Draw the internal of a ripple adder and show where the critical path is. Also show how this critical path affects the maximum clock frequency that this counter may run at.

A.4.3 Let n be the width in the counter. For example, $n = 4$ in the above counters. As n increases from 4 to 1024, how would the maximum operating frequencies of `CntrP` and `CntrA` change? Which counter can operate at higher frequency for different values of n?

Explain your answer by:

(i) Draft the design of an general n-bit `CntrA` and `CntrP` design;
(ii) Describe the critical path in these n-bit counters and determine the maximum clock frequency;
(iii) Compare the results of the two designs.

Note: the designs of `CntrP` and `CntrA` shown above are for the case when $n = 4$. Both of them can be extended to larger values of n using the same design pattern.

Part B: Mini-Project

The goal of this part of the homework is to complete the Morse code decoder project and implement the design on an FPGA as shown in Figure B.1

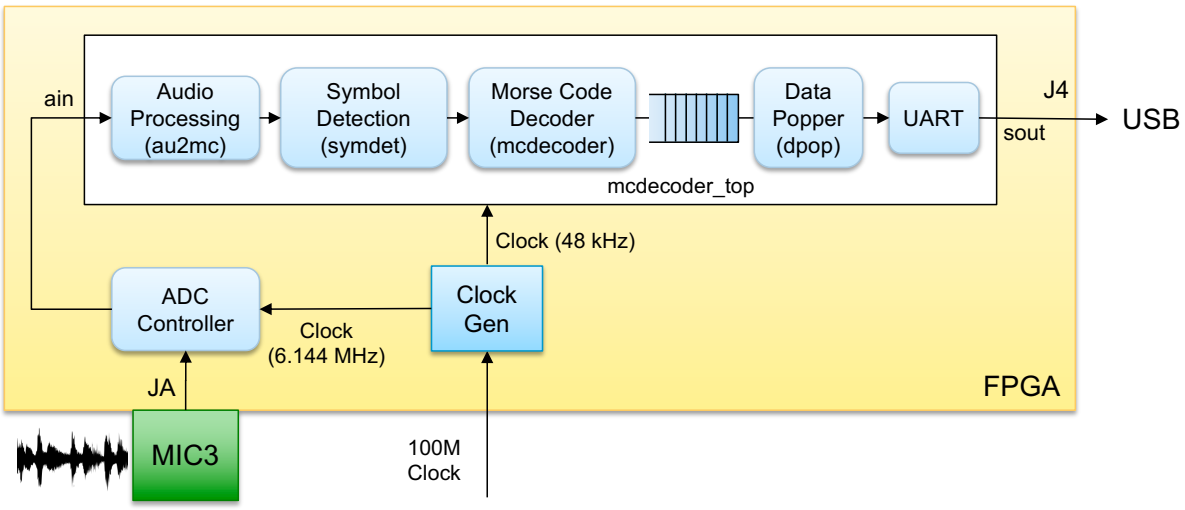


Figure B.1: Top-level block diagram of the entire design as implemented on an FPGA.

To make this project manageable, you should approach it by achieving the following milestones:

- **Milestone 1**: Input Audio Signal Processing
- **Milestone 2**: Full Morse code decoder simulation
- **Milestone 3**: FPGA implementation

B.1 Milestone 1: Input Audio Processing

The first milestone focuses on the audio input of the project. Your goal is to complete a module called `au2mc` that performs simple processing on the audio input in order to detect any Morse code transmission. You achieve this milestone by completing the design and simulating the behavior of the module `au2mc` alone.

First, a very brief introduction to audio processing. Audio signal travels in the form of mechanical wave over the air, which causes change in air pressure. A microphone detects these subtle changes in air pressure and converts that into analog voltage that reflects the waveform similar to the one shown in Figure B.2. The properties of this wave and its corresponding voltage signal thus determines the characteristics of the audio signal:

Wave Property	Audio Property
frequency	pitch
amplitude	volume
shape	timbre

For instances, a pure square wave will sound differently from a pure sine wave (timbre). A high frequency sine wave will result in a high pitch, and a loud voice will have wave with high amplitude.

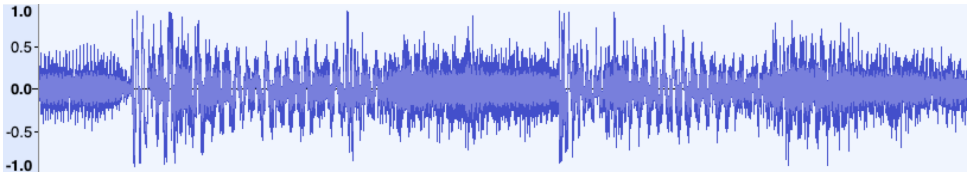


Figure B.2: A typical audio waveform.

In your project, the input Morse code will be transmitted as on-off pulses of audio tones. In the simplest form, these tones are simply pure sine wave of a particular frequency. Figure B.3 shows an ideal input Morse code waveform to your circuit.

B.1.1 Data Transmission & Encoding In this project, you will use an analog-to-digital convertor (ADC) to convert the analog voltage produced by the microphone into digital signals (numbers) for processing on the FPGA. The external ADC is included in a small board called MIC3. Details can be found from the course website.

The sampled data from the ADC are encoded as simple binary numbers from `0000_0000_0000` to `1111_1111_1111`. A study of the MIC3 schematics show that the code `0000_0000_0000` represents the *most negative* air pressure while `1111_1111_1111` represents the *most positive* pressure. In other word, when there is no sound (0 air pressure, the reading is *around* `0111_1111_1111`. It is just an estimate as it depends on actual calibration of the circuit.

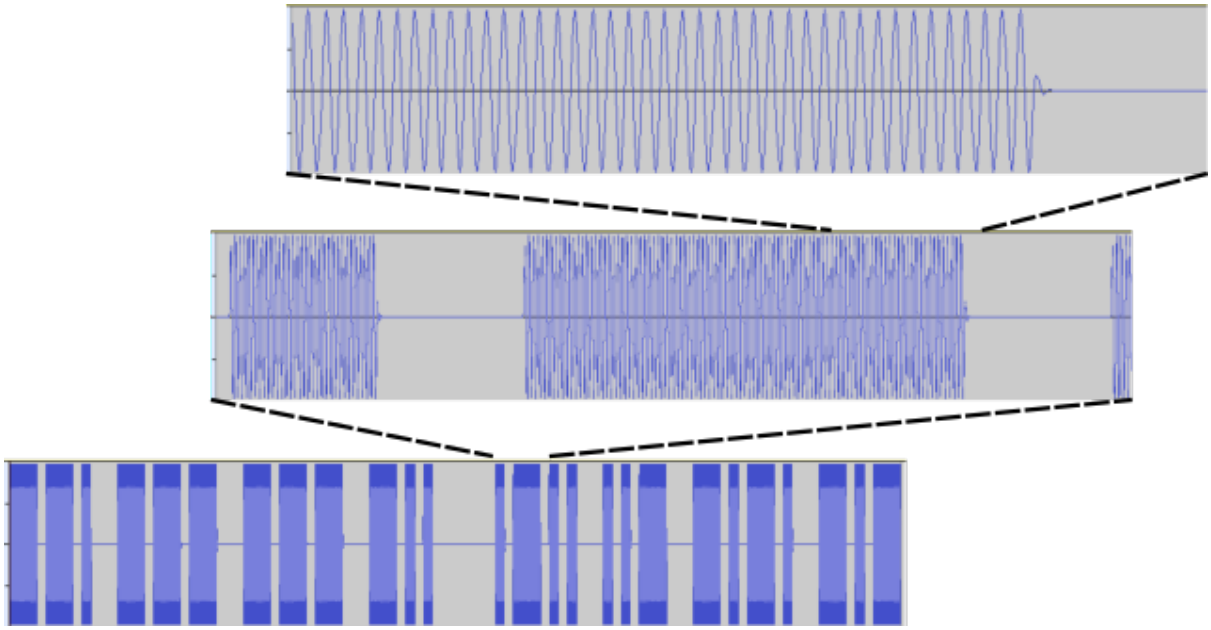


Figure B.3: Sample input waveform with Morse code transmitted. The input will be a tone around 50 Hz sine wave. The tone is pulsed as Morse code.

In other words, the data values you are receiving from the ADC are not represented by the usual 2s complement representation. You will need to subtract `1000_0000_0000` from the input to obtain the 2s complement representation of the data values.

B.1.2 Detecting Sound Source There are many ways you can process the audio samples to determine if there is a valid sound input. Here, a very simple thresholding scheme is shown. You are free to imagine your own scheme, or to apply sophisticated audio signal processing schemes here if you wish.

In this simple scheme, the central idea is to notice that in the ideal Morse code waveform (similar to that shown in Figure B.3), all your circuit needs to perform is to detect if there is *any* sound. If there is **NO** sound (silence), then it should output '0'. If there is **ANY** sound, it should output '1'.

Since the amplitude of a wave represents the level of sound, this simple scheme essentially performs a moving average of the input voltage amplitude over a small sampling window. If the average is above certain threshold, it regards there is audio input. If the average amplitude is below the threshold, it should be treated as silence.

B.1.3 Average Amplitude Let $a[n]$ be the sampled value (amplitude) at time step n. Then the moving average of the amplitude at step n over a window size of W is defined as:

$$\overline{a} = \frac{1}{W} \sum_{i=0}^{W-1} |a[n-i]| \tag{B.1}$$

For your project, set W to a suitable value that is power of 2 and you will then be able to compute (B.1) with simple add and shift operations without performing divisions.

Note that there is an *absolute operation* in (B.1). You may consider either implementing a true absolute function (which is not difficult), or you may additionally use an *approximation*:

$$|a| \approx \max(0, a) \tag{B.2}$$

You should explore different values of W or different ways to compute moving average to obtain better results.

B.1.4 Audio Processing Output The goal for the audio processing module `au2mc` is to produce the necessary `dbin` signal, which is the input to the `symdet` module. As mentioned, you should process audio signals with the `clk_48k` clock, which results in a 1 or 0 at 48 kHz. As you will be sampling at 48 kHz, it means you simply need to determine the output every cycle, possibly based on the result of the moving average computation and a threshold that you set internally.

B.1.5 Testing & submission A testbench program `tb_au2mc` has been provided to you. Note: although the testbench program is written in VHDL, just as any other testbench programs, it is **NOT** part of the final hardware design.

In this case, the code inside `tb_au2mc` reads an audio file from your computer and send the audio sample data to your hardware module `au2mc` for simulation. You can check the output of `au2mc` in the simulator to make sure it produces the expected output, i.e., sequence of 1 and 0 representing Morse code.

3 audio files are provided to you for testing. You can also record your own if you wish. Ask on Piazza for the procedures to create these audio files[1].

Submit the simulation waveform output in your report for this Milestone.

[1]The process is basically to convert standard WAV files with integer encoding using standard audio processing tools.

B.2 Milestone 2: Full Morse code decoder simulation

The target of this milestone is to complete the full Morse code decoder and to *simulate* the function of the full system. Specifically, you will implement and simulate the function of the block `mcdecoder_top` shown in Figure B.1.

Except for the `clk` and `clr` signals, `mcdecoder_top` simply takes audio samples as input (`ain`) and produces a serial signal (`sout`) for UART transmission. To complete this milestone, you need to demonstrate through simulation, that your `mcdecoder_top` design can take in the audio sample from a testbench (same as the ones in Milestone 1), decode the signal to a valid ASCII character, and produce the necessary serial output from the UART.

Since most of the blocks needed to implement `mcdecoder_top` have already been designed in HW1, HW2 and Milestone 1, you only need the following to achieve Milestone 2:

- Create a new VHDL module `mcdecoder_top` that instantiate all the blocks;
- Modify `symdet` from HW2 such that it works with input with a large base unit u;
- Design a small data popper (`dpop` to connect the output of the FIFO to the input of UART.

B.2.1 Top level module To design and simulate the top-level module `mcdecoder_top`, you need to create a new project that includes all the modules that you will needed. A few notes about this integration:

- You need to create your own top level VHDL file, `mcdecoder_top`, that combines all the modules. This top level VHDL entity has the same input as `au2mc` (i.e., audio sample, clock and clear), and produces output the same as the UART (a single bit).
- The modules `symdet` and `mcdecoder` are from Homework 1 and 2. You need to include the design from your own group from homework 1 and 2. You may, and you should, improve the functionality of these modules if they were not fully functional when you turned in homework 1 and 2. You may use the posted sample answer as a starting point to help improve your original design.
- You can assume your entire system operates on a 48 kHz clock, which will be passed to your system through the clock port (`clk`).

To simulate the behavior of the system, you need to build a new testbench file `tb_mcdecoder_top`, which is mostly the same as `tb_au2mc`, except you need to make sure you can reset (`clr`) the system correctly in the simulation. See below section about FIFO.

B.2.2 Modify Symbol Detector In homework 2, you had the choice to implement `symdet` at different levels. When working with real-world input, you need a `symdet` module at L3 in theory. For this project, the audio test input will always have a fixed value of u. As a result, you will need only a L2 `symdet` with a large but almost fixed value of u. The exact value of u depends on your `au2mc`, but should be around 3000. That is, a dot will be about 3000 cycles and a dash is about 9000 cycles. The value of u will not be exact even when the input audio is produced by a computer due to factors such as how the audio is recorded, what is the phase of the audio wave, how your `au2mc` is implemented etc. For this homework, you will need to modify your `symdet` from homework to work with this range of u values.

B.2.3 Data popper In `mcdecoder_top`, there is a FIFO between the output of `mcdecoder` and the UART. If you use a standard or built-in FIFO (like in HW2), the data that you have pushed into the FIFO will simply reside inside the FIFO until they are externally popped. At the same time, the UART would passively wait for data to be transmitted.

As a result, you need to implement a simple circuit that connect the two. In this project, we call it a "data popper" (`dpop`). The design of `dpop` is quite straightforward:

`dpop` should monitor the control signals (e.g. `empty` from the FIFO and the UART (e.g. `ready`), and when both of them are asserted (FIFO is not empty and UART is ready, then it should pop a data from the FIFO and send the data to the UART.

Implement `dpop` and use it in `mcdecoder_top`. To help with your debugging effort, you probably want to design and test it separately before using it in the over system.

B.2.4 FIFO If you use the built-in FIFO from the Vivado tools, then you must do the following in order to simulate the operation of the FIFO:

- Make sure you have the following library included (uncomment them if you produced your top level VHD file from Vivado) in your top level module `mcdecoder_top`:

```
library UNISIM;
use UNISIM.VComponents.all;
```

- Instantiate the FIFO in your design. HINT: You can copy-and-paste the port definition from the produced VHDL (`fifo_generator_0`) for your convenience.
- IMPORTANT: Make sure you RESET the system in the beginning of your simulation (by setting your `clr` signal high for a short time before turning it back to '0'. The Xilinx produced FIFO **MUST** be initially reset at least once for it to behave correctly in simulation.

If you designed your own FIFO, then you may ignore the part about the use of `UNISIM` library.

B.2.5 Simulating Whole System With all the pieces, you should now be able to simulate the function of the entire system in Vivado. Remember to change the simulation duration to a larger value so you can simulate the operation of the system for the complete audio input.

B.2.6 Milestone Submission Show the simulation waveform using the 3 provided audio files as inputs. Make sure you highlight (i) the output from the module `mcdecoder` with correctly decoded ASCII characters, and (ii) the serial output from the UART when the corresponding character is sent to the computer. *Hint:* If you fail to implement the whole system, try to demonstrate at least part of the systems. For example, if you fail to implement an improved `symdet` that works with large values of u, then you may consider simulating a design without `au2mc` and use the L0 testbench from HW2 to simulate the rest of the design.

B.3 Milestone 3: Implementation on FPGA

In this part of the project, your task is to implement your design from homework 2 on the actual FPGA board. For your reference, the board you will be using is the Digilent Basys 3 board. The main FPGA on your board is a Xilinx Artix-7 FPGA (XC7A35T- 1CPG236C). For details of the board, please refer to the Basys 3 FPGA Board Reference Manual. (Note: on Digilent website the link is labeled as Datasheet.)

A sample project, called `au2mcdecoder` is provided to you with all of the necessary additional circuits and configurations correctly setup to help you implement the design on the FPGA board.

B.3.1 Clock An important part of a real project, as oppose to a simulation project, is that you need to run with an actual clock signal. On the Basys 3 board, a 100 MHz clock source is available and you will use that as the main clock course in your design.

In your design, you will need 2 clock signals to drive different parts of the circuits:

- `clk_6144`: A 6.144 MHz clock used in the audio ADC controller;
- `clk_48k`: A 48 kHz clock used for the main design.

For those of you who are curious, the above clock frequencies are chosen specifically for ease-of-implementation in your design: 48 kHz is a common audio processing sampling rate, and is also 5 times higher than the 9600 baud rate needed for UART communication. 6.144 MHz is 128 times higher than the main clock frequency, and can be precisely produced from the 100 MHz clock source with the FPGA's on-board multi-mode clock management (MMCM) circuit.

The circuits required to produce these clock signals are already provided to you. Feel free to explore how they work but DO NOT modify any of these circuits.

B.3.2 Audio Input On the actual FPGA board, an external microphone is used to listen to your real world audio input. The audio module you will used is the Digilent Pmod MIC3 modules, which contains a Knowles Acoustics SPA2410LR5H-B microphone and a Texas Instruments ADCS7476 analog-to-digital converter.

A basic controller (`adccntlr` is provided to you. It's job is to communicate with the ADC to command the ADC to produce an audio *sample*. For this project, the controller sample the audio signal at a fixed rate of 48 kHz with $12-bit$ accuracy. In other words, it produces a 12-bit sample every 1/48000 seconds. These audio data sample will then become the digital audio samples that can be processed in later steps. Recall that your system is also running at 48 kHz by design. Therefore, it means the module is produce exactly 1 sample every cycle. It is designed specifically this way to make your design simple, and to make it matches the expected behavior from Milestone.

B.3.3 Implement Design Once you have all the files set up, click the "Run Synthesis" button to start the synthesis process. If your synthesis runs correctly (check for Error and Warning messages), you should also run the "Generate Bitstream" step of the Vivado tool flow.

B.3.4 Configure FPGA To test your design on the Basys 3 FPGA board, do the following:

- Connect your Basys 3 board to your computer with a micro-USB cable. Connect your cable to connector **J4** (labeled as **PROG** on the board).
- Turn on the FPGA board. By default you will see numbers showing up on the 7-segment display and increases every second. It is the default designs shipped from the factory.
- To configure the FPGA, select "**Open Hardware Manager**" from Vivado. Your FPGA board should show up under the **Hardware** pane.

- Right click on the FPGA showed up (Artix-7) and select "**Program Device**...". Select the bitstream that you have just created (usually with the same name as your project with a `.bit` extension.)

B.3.5 Serial Terminal In order to receive data from your Basys 3 board, you need to run a serial terminal program on your host computer. Make sure you set up your serial terminal with 9600 baudrate and 8-N-1 configuration. Refer to homework 2 handout for more details.

The default configuration for the Basys 3 board will send a text string to the host computer when it first starts up through its UART. You should use this as a test of your serial connection before you test your own design.

If you set up is correct, when the Basys 3 starts up you should see a text string show on your computer screen.

Congratulation! If you can see the correctly decoded message on your computer screen, your Morse code decoder's decoding logic is completed!

B.4 Submission

For Part B of this homework, turn in, as a group, a report describing your design. Include a top-level block diagram of your audio processing module. Include all the items mentioned from each Milestone above. Specifically, explain (i) the algorithm that you have implemented to perform audio detection, (ii) any modification of `symdet` from hw2 (iii) your data popper at the output of the FIFO.

Then during project demo, be prepared to show a working design. Before you attend the interview, setup the board and load the correct bitstream to the board for demo.