Lab 2: Simple iPod

Part 1: program the flash memory

~~Done, need to test the FPGA solution~~

The behavior of the program:

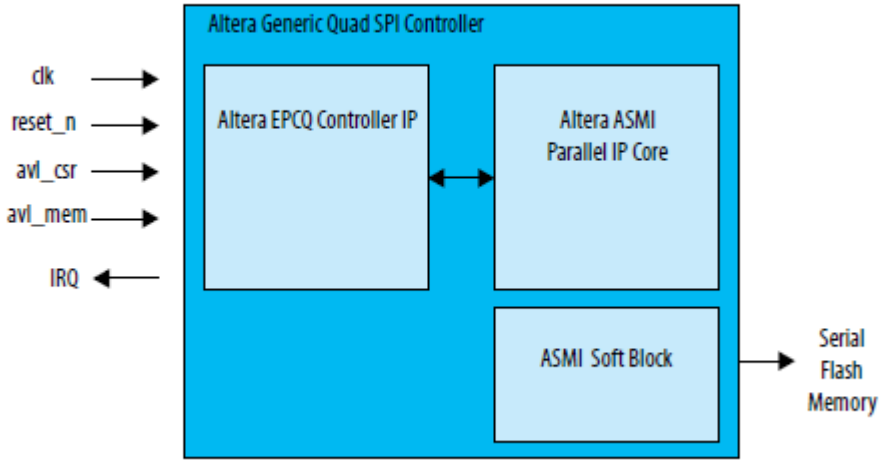
1. LCD updated with **someting**
2. // TODO: figure out that something
3. Action table

action	Functionality	Comments	Implementation
Keyboard E	Start the music	need to press this before starting music	as the start or enable of the fsm, or get the address start updating
Keyboard D	Stop the music	I think it restarts?	stop the address updating
KEY0	slow up music		change the speed variable
KEY1	speed up the music		change the speed variable
KEY2	reset the speed		reset the speed
Keyboard B	Play the music backwards		change the direction of the address updating to negative
Keyboard F	Play the music forward		change the direction of the address updating to positive
Keyboard R	Restart the music	start from the beginning	set the address to 0

4. Interface
    - speed count (change the sameple space)

- read keyboard input
  - keyboard praser
  - keyboard controller
    - The output is kbd_data. I think we can assume that the kbd_recevied_ascii_code is the final data we get, after prasing.
- Clock_divider
- audio_data
  - Audio data is that: the sample is a 16 bits data that is not signed, means that it's ranged from -8 bits of audio amplitude to + 8 bits amplitude. We need to tanslate that 16 bits to 8 bits.
- flash memory interface
- Describsion of the interface of Altera Generic QUad SPI Controller

Figure 39-1: Altera Generic Quad SPI Controller Block Diagram



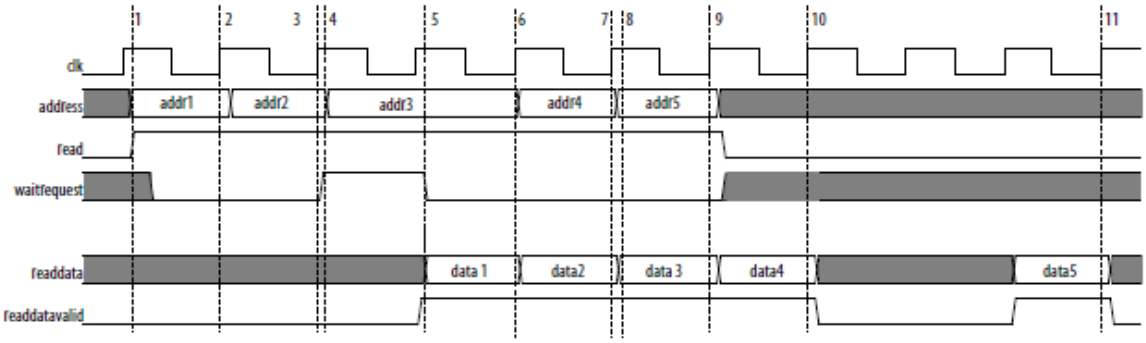
- Specification on the signals
-

Signal	Width	Direction	Description
avl_mem_addr	*	Input	Avalon-MM address bus. The address bus is in word addressing. The width of the address will depends on the flash memory density minus 2. If you are using Arria 10, then the MSB bits will be used for chip select information. User is allowed to select the number of chip select needed in the GUI. If user selects 1 chip select, there will be no extra bit added to avl_mem_addr. If user select 2 chip selects, there will be one extra bit added to avl_mem_addr. Chip 1 – b'0 Chip 2 – b'1 If user select 3 chip selects, there will be two extra bit added to avl_mem_addr. Chip 1 – b'00 Chip 2 – b'01 Chip 3 – b'10
avl_mem_read	1	Input	Avalon-MM read control to memory
avl_mem_write	1	Input	Avalon-MM write control to memory
avl_mem_wrdata	32	Input	Avalon-MM write data bus to memory
avl_mem_byteenble	4	Input	Avalon-MM write data enable bit to memory. During bursting mode, byteenable bus bit will be all high always, 4'b1111.
avl_mem_burstcount	7	Input	Avalon-MM burst count for memory. Value range from 1 to 64
avl_mem_waitrequest	1	Output	Avalon-MM waitrequest control from memory
avl_mem_rddata	32	Output	Avalon-MM read data bus from memory

- Examples of Piplining reading.
-

Figure 3-5: Pipelined Read Transfers with Variable Latency

The following figure shows several slave read transfers. The slave is pipelined with variable latency. In this figure, the slave can accept a maximum of two pending transfers. The slave uses `waitrequest` to avoid overrunning this maximum.

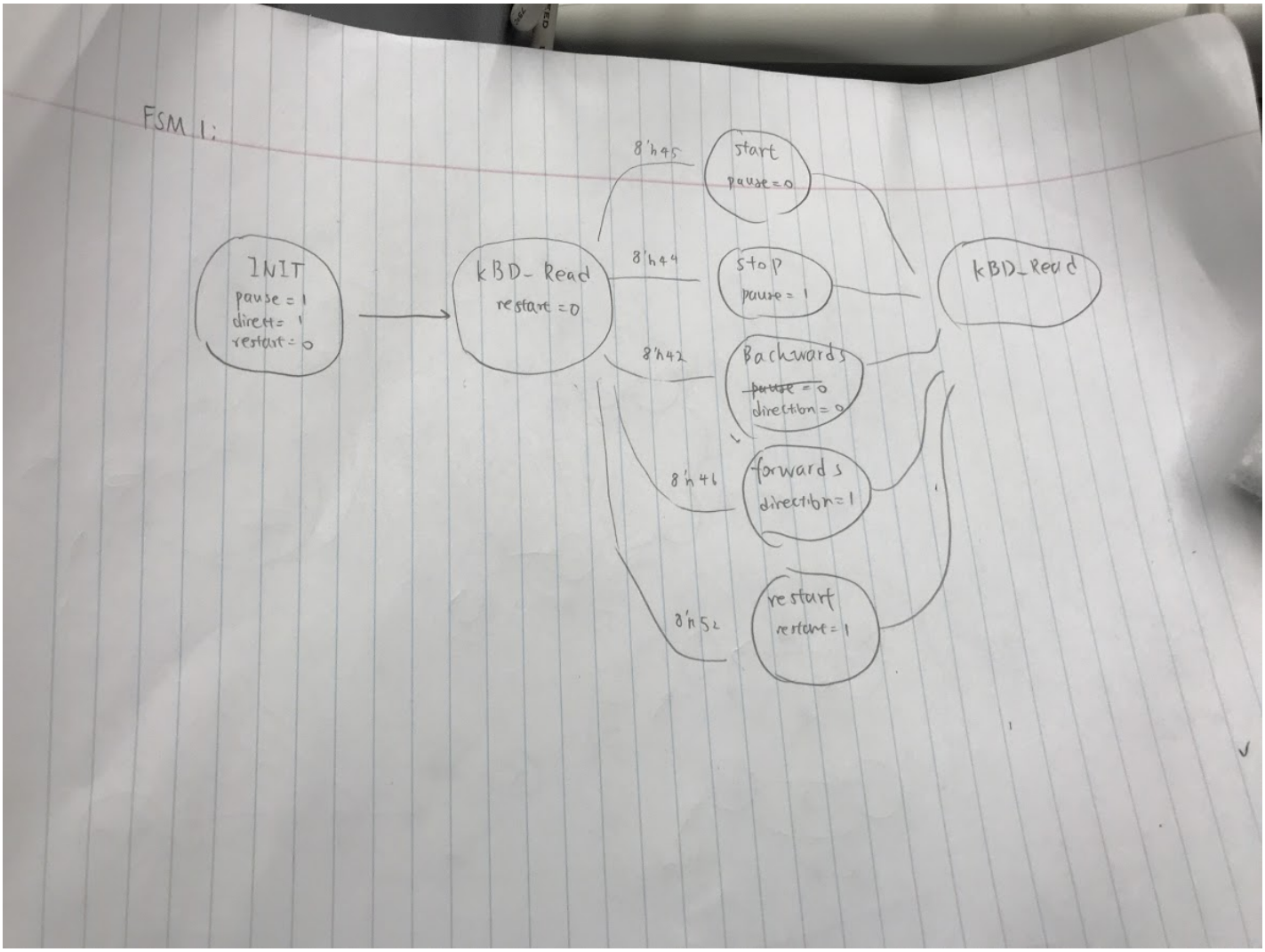


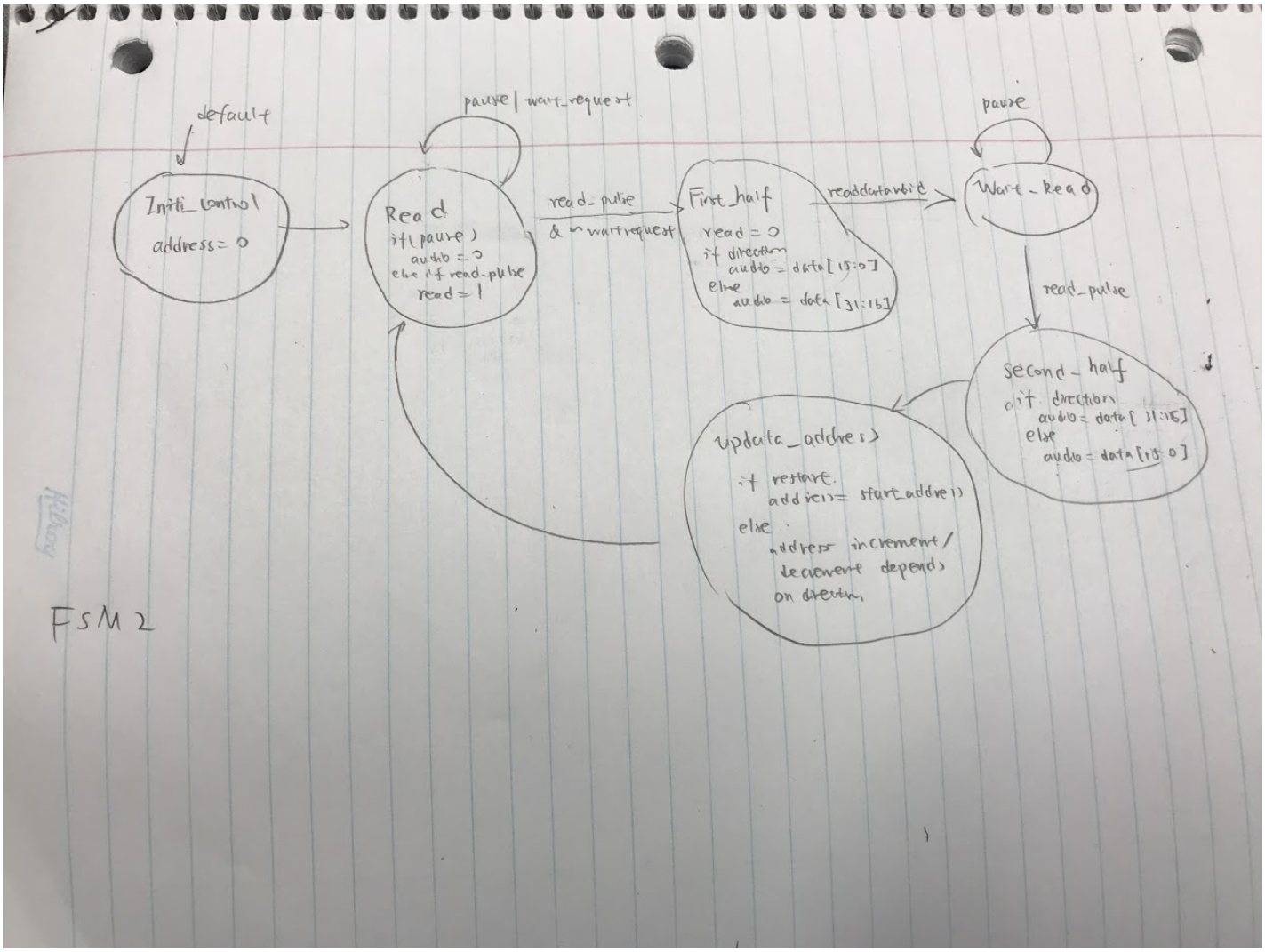
The numbers in this timing diagram, mark the following transitions:

1. The master asserts `address` and `read`, initiating a read transfer.
2. The slave captures `addr1`.
3. The slave captures `addr2`.
4. The slave asserts `waitrequest` because it has accepted a maximum of two pending reads, causing the third transfer to stall.
5. The slave asserts data1, the response to addr1. It deasserts `waitrequest`.
6. The slave captures `addr3`.
7. The slave captures `addr4`. The interconnect captures `data2`.
8. The slave drives `readdatavalid` and `readdata` in response to the third read transfer.

-

5. FSM:


FSM 1:
INIT
pause = 1
direct = 1
restart = 0
kBD- Read
restart = 0
8'h45
start
pause = 0
8'h44
stop
pause = 1
8'h42
Backwards
~~pause = 0~~
direction = 0
8'h46
forwards
direction = 1
8'h52
restart
restart = 1
kBD_Read



Part 2: Look at the solution file to understand what to be done

Part 3: Just do it

Open the file "simple_ipod_solution.v". Look inside this file Write_Kbd_To_Scope_LCD modules, Kbd_ctrl and key2ascii. These modules are already written and work well, no need t change them. However, I recommend that you enter the modules to see how they work. Overall, Kbd_ctrl manages the interface with the keyboard, and outputs the variable "kbd_scan_code", which is converted by the module key2ascii to an ASCII code (which is the same code of the parameters

"character_A", "character_B" etc., that you already know).
This ASCII code is used by the module Write_Kbd_To_Scope_LCD
to write the letters in the LCD. The module
Write_Kbd_To_Scope_LCD is interesting because it contains an
FSM - try to understand how it works. To understand the interface with the keyboard, you can look at the DE1 manual.

Part 4: Start to write code, after designing

1. interface with the flash memory

It's already written here: gotocode

2. design a FSM

2.1. FSM for reading flash

make it glicth free. Use the output included covention.

2.2 FSM for control.

Part 5: final check

name	percent %	comment
flowchart	10	
clock divider		
functionality	40	
read/comments	10	
modulization	10	
explaination	10	
efficiency	10	

name	percent %	comment
Bonus	15	