ECE532 Project Proposal

January 30, 2008.

Project Title

NES Audio Processing Unit

Project Team

- *Cedomir Segulja, seguljac@eecg.toronto.edu*
- *Bill Dai, bill.dai@utoronto.ca*

Project Description

The goal of our project is to implement Nintendo Audio Processing Unit in the FPGA. The three[1] major processing units of the original NES hardware are Central Processing Unit (CPU), Picture Processing Unit (PPU) and the Audio Processing Unit (APU). The original CPU was modified 8-bit MOS 6502 microprocessor which used memory mapped I/O to communicate with the other components.

NES Audio Processing Unit

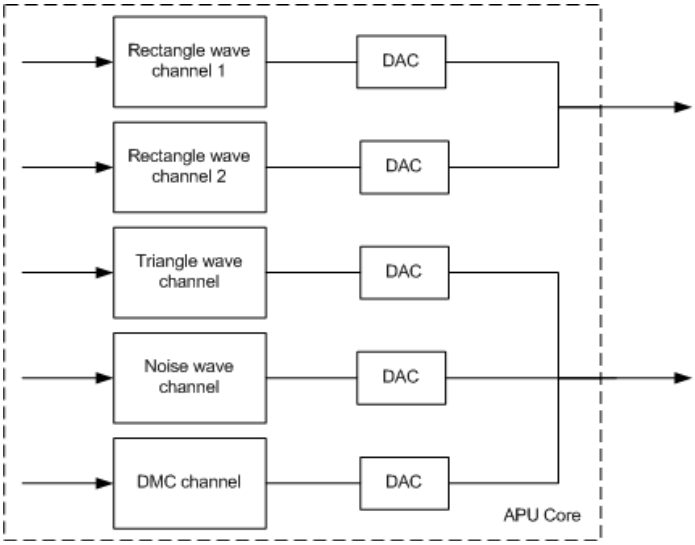


Figure 1: APU Core

The original APU (shown on Figure 1) has 5 internal channels (components). Each channel is configured with memory-mapped registers (typically three or four 8-bit registers per channel), and connected to a Digital-Analog Converter (DAC) which generates the final, analog signal. The four of them generate simple waveforms: rectangle, triangle and the noise wave forms. The fifth channel is the Delta-Modulation Channel (DMC) which is able to play samples. The DCM has two modes of operation. With the first, DMA mode, samples are fetched directly from the memory (without CPU's intervention). In this case the modulation used is delta-modulation: each bit in 8-bit sample represents the increase or decrease of the current output. With the second mode, samples are fed directly to the DCM in 7-bit PCM form.

[1] The original NES CPU, the 2A03, incorporated the APU. However, we will be looking at it like two separate components.

The APU has two sound outputs. One carries the two square wave channels and the other one carries the triangle wave channel, the noise channel and the DCM channel. However, in the original NES the final output is mono sound as both intermediate outputs are mixed together.

For detailed channel information, please refer to Appendix A.

System Design

Proposed system design is shown in Figure 1. We will be using the Xilinx XUP Virtex™-II Pro Development System. MicroBlaze soft-processor will play the role of NES CPU, driving the APU. The APU is designed as peripheral unit that communicates with the CPU through the OPB bus. CPU writes and reads the APU internal registers using memory-mapped interface. Using OPB bus offers modularity of the system; in that way other components (primary the PPU) will be easily added. Although using the bus could potentially cause bandwidth issues, this is not the case in this system as the original NES CPU operates on 1.7 MHz's.

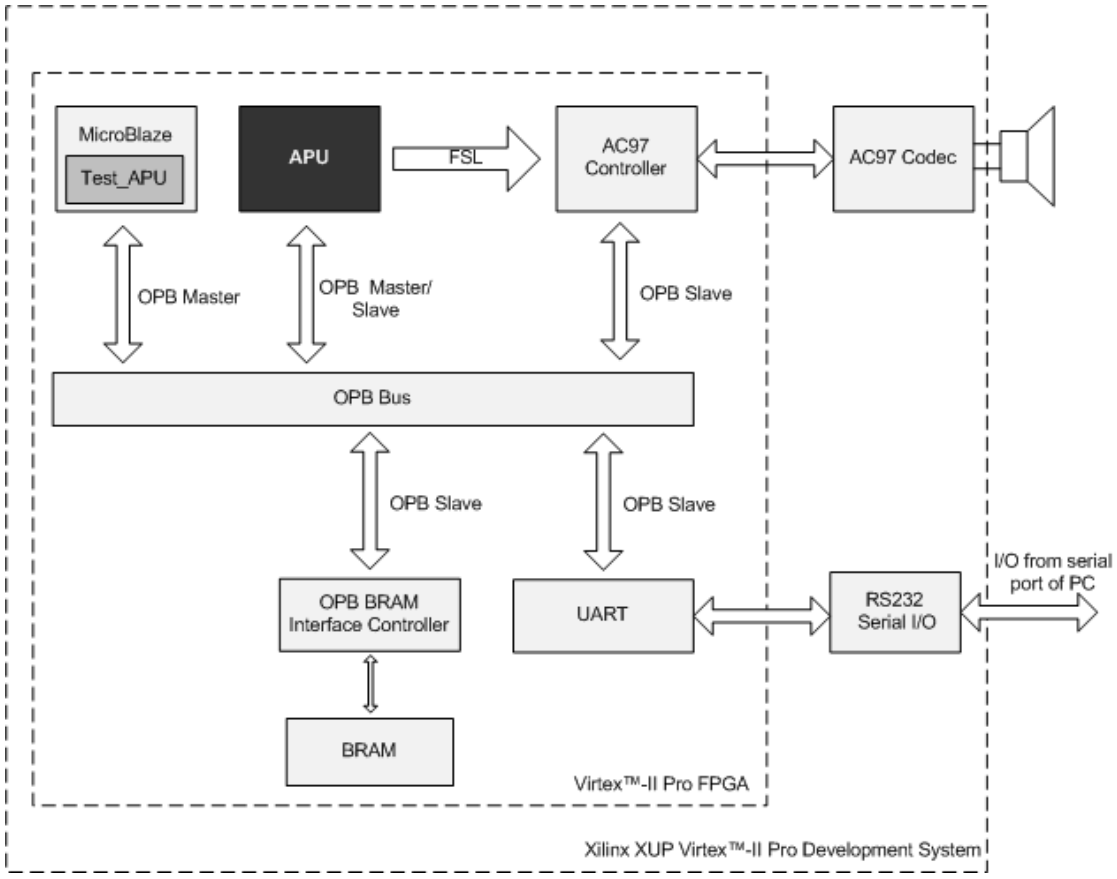


Figure 2: System Block Diagram

The game data and instruction will be stored in the on-chip block RAM (BRAM). The original game sizes are 16KB so this will definitely fit (the XC2VP30 has 2,448Kb BRAM). Although there is a possibility that the final NES system will use the original cartridges that will be connected to the board, for simplicity and testing purposes we have chosen the former design option. The BRAM will be accessed through the OPB BRAM Interface controller. The choice of this design option (instead of using faster LMB to connect the CPU and the BRAM) is described next. Again, there are no bandwidth issues for the reason stated earlier.

Both APU and the PPU should be able to directly access the memory; that is, operate in Direct Memory Access Mode (DMA). Using the OPB bus allows us to (easily) support this operation mode, as OPB bus (as opposite to the LMB bus) allows multiple masters. Having that said, the APU will be connected to the OPB as a master and will be acting as a master only when communicating with the memory.

For digital to analog conversion, we will be using the AC97 codec which is the device placed on the Xilinx XUP Virtex™-II Pro Development System. Codec is controlled with the AC97 controller. The APU communicates with the AC97 controller through the FSL. For configuration and debugging purposes, the AC97 controller is also connected to the OPB bus, which allows communicating with the CPU. Furthermore, for the debugging purposes we will use serial port to display messages on the monitor. The RS232 is controlled by the UART.

The only software component is the Test_APU, which is the simple application for testing the APU.

IP Cores

The following cores will be reused:

- MicroBlaze (Xilinx IP)
- OPB Bus (Xilinx IP)
- AC97 Controller (Xilinx IP, but also using documentation (and maybe code) of past projects)
- OPB BRAM Interface Controller (Xilinx IP)
- BRAM (Xilinx IP)
- UART (Xilinx IP)

We will build the APU as our custom IP core. Each channel will be implemented as separate component, and the outputs from the all channels will be "added" together in mixer component (see Figure 3.). Functionality of each channel is described in Appendix A.

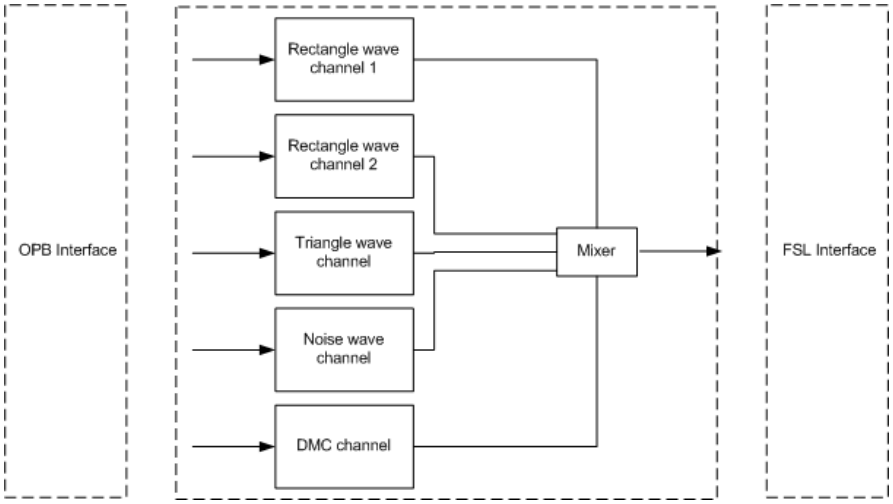


Figure 3: APU IP Core

Milestones

The main things that we have to do/learn:

- Detail knowledge of behavior of each APU channel (this is already covered to some extent)
- Implementing each APU channel functionality
- Understanding AC97 Controller
- Developing a user-design peripheral

We propose the following implementation plan:

February 13	Model and simulate one channel (for example rectangular wave channel) of the APU. Use Xilinx demos (or past projects) to design a MicroBlaze system which uses AC97 controller to play (any) sound.
February 27	Develop user-design OPB peripheral (APU) and connect it to the proposed system
March 5	Implement other channels. The DMC channel is potentially the trickiest as it request communication with memory.
March 12	Testing. Building OPB-6502 Bridge[2]
March 19	Testing.
March 26	Final demo

Table 1: Milestones

References

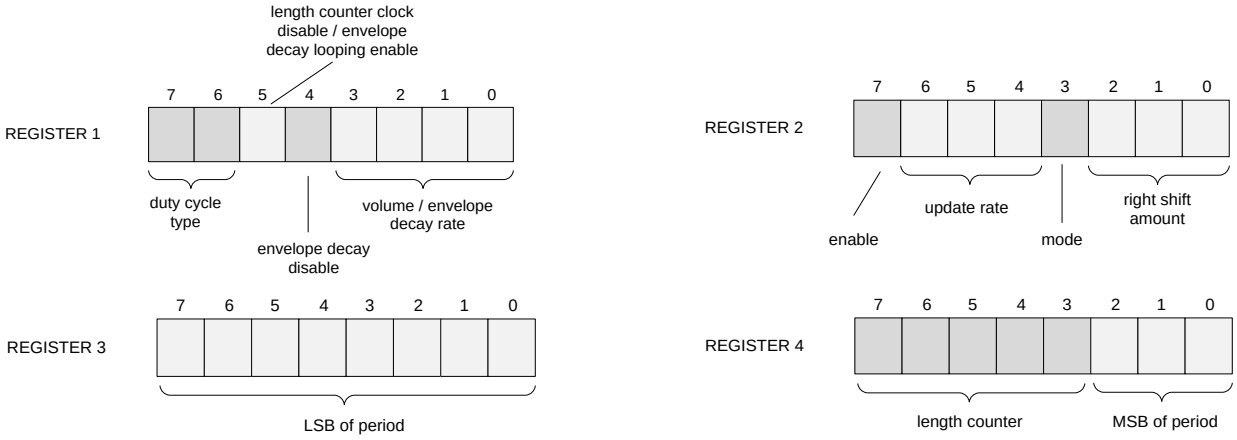
1. Patrick Diskin, Nintendo Entertainment System Documentation
2. Brad Taylor, 2A03 technical reference
3. Brad Taylor, NES APU Sound Hardware Reference
4. Brad Taylor, Delta modulation channel tutorial 1.0
5. Brad Taylor, The NES sound channel guide 1.8

[2] OPB-6502 Bridge will be provided so that developed APU can be integrated into final NES design, which will be using 6502 bus.

Appendix A

Rectangle Channel

Registers



Basic Operation

Rectangle wave channel generates rectangle wave forms. Frequency of a generated waveform is defined by $\frac{CPU\_clock}{16 * (period+1)}$, where *period* is defined in REGISTER 3[7:0] and REGISTER 4[2:0]. Duty cycle of the waveform is defined by *duty cycle type* (REGISTER 1[7:6]) as shown in Table 2. Volume of positive level is defined by *Envelope Decay Unit*. Negative is zero.

Duty cycle type	Output (positive/negative)
0	2/14
1	4/12
2	8/8
3	12/4

Table 2: Duty cycle types

Length Counter

Length counter is used to set the total duration of the waveform. Duration (8 bit value) is defined by encoding the value from REGISTER 4[7:3]. This relation is shown in Table 3. When the length counter arrives at a count of 0, the counter will be stopped, and the channel is silenced. By writing 1 to REGISTER 1[5] counting is paused. It operates on the frequency of 60Hz.

Envelope Decay

Has effect only if the bit REGISTER 1[4] is 0. If this is the case, the output volume is determined by internal register (ENVELOPE_REGISTER), which is decreased by 1 with the frequency $\frac{240Hz}{DECAY\_RATE+1}$, where the *DECAY_RATE* is defined in REGISTER 1[3:0]. When it reaches zero it will (1) stop (channel muted) or (2) wrap-around, depending on the bit REGISTER 1[5]. It is always counting, no mater of the REGISTER 1[4]!

Frequency Sweep

Has effect only if the bit REGISTER 2[7] is 1. In this case, the frequency is changed in real-time. It can be both incremented and decremented, depending on the value REGISTER 2[3]. This relation is shown in Table 4. Refresh rate is defined by $\frac{120Hz}{REFRESH\_RATE+1}$, where *REFRESH_RATE* is defined in REGISTER 2[6:4].

REGISTER 4[7:3]	Duration (Hex)	REGISTER 4[7:3]	Duration (Hex)
00000	05	10000	06
00001	7F	10001	08
00010	0A	10010	0C
00011	01	10011	09
00100	14	10100	18
00101	02	10101	0A
00110	28	10110	30
00111	03	10111	0B
01000	50	11000	60
01001	04	11001	0C
01010	1E	11010	24
01011	05	11011	0D
01100	07	11100	08
01101	06	11101	0E
01110	03	11110	10
01111	07	11111	0F

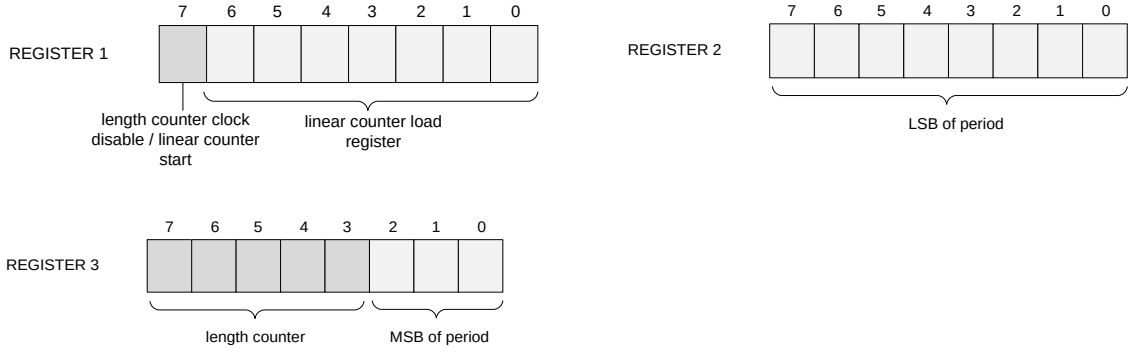
Table 3: Duration of rectangle wave form.

REGISTER 2[3]	*New period*
0	$Current + (Current >> REGISTER\ 2[2:0])$
1	$Current - (Current >> REGISTER\ 2[2:0]) - 1^*$

Table 4: Frequency sweep, calculating new period. *-1 is added only for Channel 1.

Triangle Channel

Registers



Basic Operation

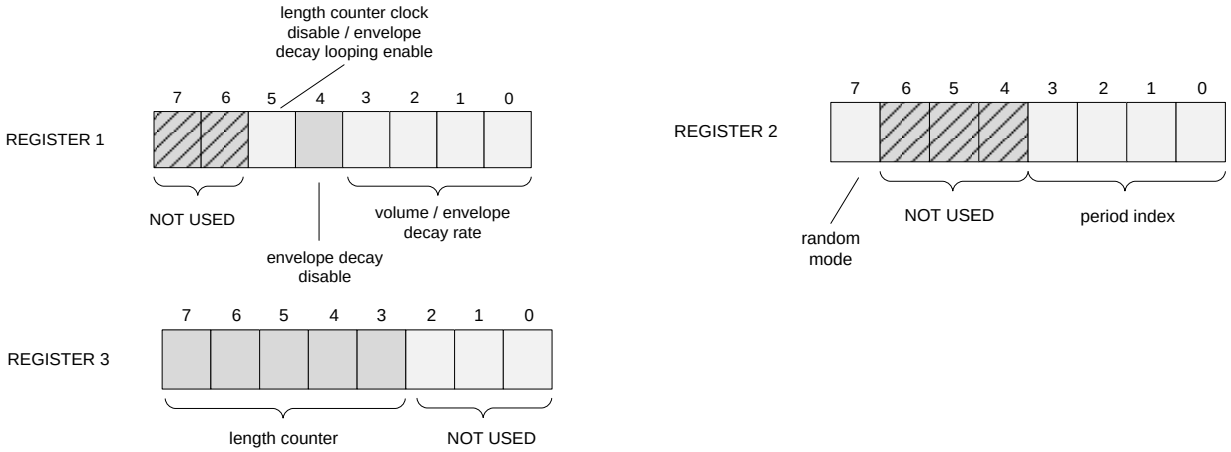
Triangle wave channel generates triangle wave forms with a 4-bit resolution (16 steps). Frequency of a generated waveform is defined by $\frac{CPU\_clock}{32 * (period+1)}$, where *period* is defined in REGISTER 3[7:0] and REGISTER 4[2:0]. It uses the length counter in the same way as the rectangle channels.

Linear Counter

Performs same task as the length counter (muting the channel after certain time period) but just more precise. It operates on 240Hz. The value is stored in REGISTER 1[6:0].

Noise Channel

Registers



Basic Operation

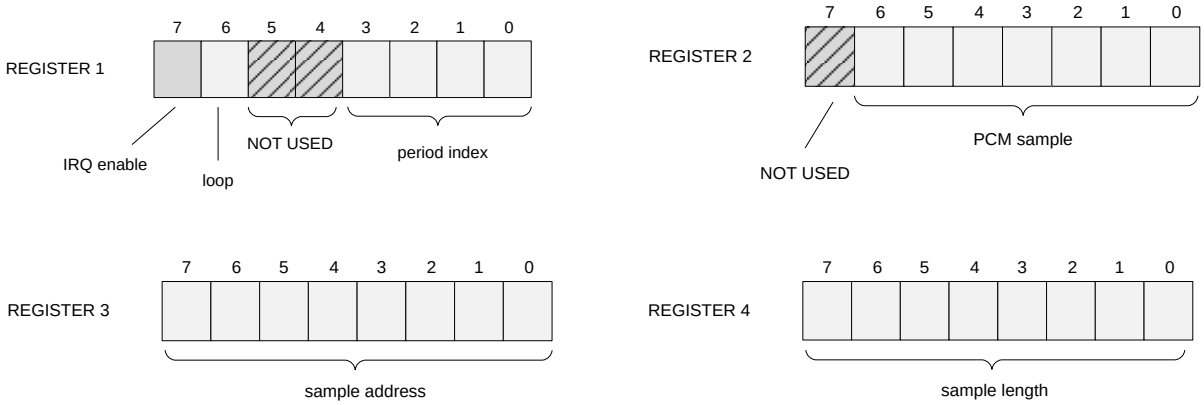
The main component of the noise channel is random number generator, which as output produces pseudo-random binary sequences. It is capable of producing 93-bit (short) and 32,767-bit (long) sequences. The type of sequence is defined in REGISTER 2[7]. The random number generator is implemented as a 15-bit shift register. The frequency is determined by $\frac{CPU\_clock}{2 * period}$, where period depends on REGISTER 2[3:0] as shown in Table 5. The output of random number generator is used an input for *Envelope Decay Unit* (same as for rectangle channels), where final volume level is determined.

REGISTER 2[3:0]	***period* (Hex)**
0000	002
0001	004
0010	008
0011	010
0100	020
0101	030
0110	040
0111	050
1000	065
1001	07F
1010	0BE
1011	0FE
1100	17D
1101	1FC
1110	3F9
1111	7F2

Table 5: Determining *period* parameter for the noise channel

DMC channel

Registers



Basic Operation

DMC channel as an output provides 7-bit PCM sample. For generating PCM samples channel uses delta-modulation and DMA. REGISTER 3 defines starting address for fetching bytes that will be used for generating samples. REGISTER 4 defines how many bytes should be transferred from the memory. DMA frequency is $\frac{CPU\_clock}{period}$, were period is dependent on the REGISTER 1[3:0] as shown in Table 6.

REGISTER 1[3:0]	***period* (Hex)**
0000	D60
0001	BE0
0010	AA0
0011	A00
0100	8F0
0101	7F0
0110	710
0111	6B0
1000	5F0
1001	500
1010	470
1011	400
1100	350
1101	2A0
1110	24)
1111	1B0

Table 6: Determining *period* parameter for DMC channel

When a byte is fetched, it is stored in internal 8-bit shift register. The shift register is clocked at 8x the DMA frequency. The bits shifted out are then used as an input of the 6-bit internal delta counter (with clipping behavior). The counter works on the same frequency as shifter. The final output is defined as OUTPUT [7:0]={DELTA_COUNTER[6:0], REGISTER 2[0]}. Writing to REGISTER 2 has effect of loading that value in the internal delta counter. This can be used for playing PCM samples directly, in which case programmer has to take care of the output frequency.

REGISTER 1[7:6] defines what happens when all samples are played. If REGISTER 1[6] is set then DCM works in loop mode, constantly playing the same samples. If REGISTER 1[6] is 0, then when all samples all played DCM causes interrupt if and only if REGISTER 1[7] is set.

Channel mixing

The final output is determined as:

$$out1 = \frac{95.88}{\frac{8128}{rectangle1 + rectangle2} + 100}$$

$$out2 = \frac{159.79}{\frac{1}{\frac{triangle}{8227} + \frac{noise}{12241} + \frac{dmc}{22638}} + 100}$$

$$output = out1 + out\ 2$$

,where *rectangle1*, *rectangle2*, *triangle*, *noise* and *dmc* are the outputs from corresponding channels.