# Hacking a Sega Whitestar Pinball

**A guided tour through the peculiar design of a pinball machine**

Pierre Surply

EPITA Systems/Security Laboratory (LSE)

## 1 Sega Starship Troopers Pinball Overview

The *Sega Starship Troopers Pinball* is fairly representative of the *WhiteStar Board System* used in several *Sega* pinball games and *Stern Pinball*. This hardware architecture was firstly designed in 1995 for the *Apollo 13* game with the objective to be convenient and extensible in order to be reusable for other playfields. This way, Sega could exploit a large number of licenses without having to design new control circuits for each machine.

This architecture is based on three Motorola `68B09E` clocked at 2MHz and used as main CPU, display controller and sound controller. The two last are mainly dedicated to monitor application-specific processors: for instance, the `6809` used on the display board is charged to interface a `68B45` CRT controller to the main CPU. The sound processing is handled by a `BSMT2000`, a custom masked-rom version of the `TI TMS320C15` DSP.

Sega used this system for 16 other games including *GoldenEye*, *Star Wars* and *Starship Troopers*.

### 1.1 Playfield's wiring

The playfield wiring is quite simple: all switches are disposed in a matrix grid. This method provides a simple way to handle a high number of I/O with a reasonable number of connectors. So, in order to read the switches state, the CPU has to scan each raw of the matrix by grounding it and watching in which column the current is flowing.

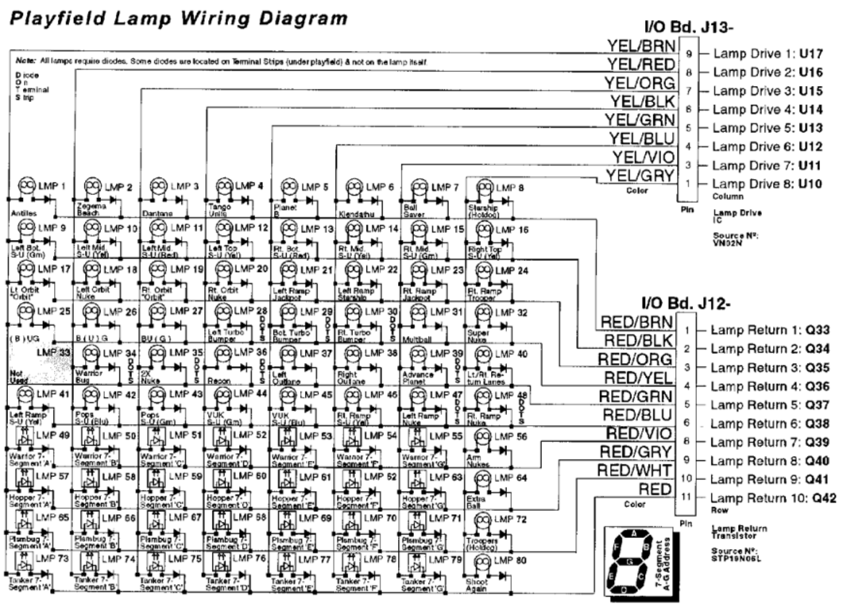

**Fig. 1.** Lamp wiring

A similar circuit is used to control playfield lamps: each raw has to be scanned by grounding it and applying voltage on the column connector according to lamps that have to be switched on the selected raw.

It's truly easy to control a high number of lamps with this layout. The following code switches on the lamp 31 (multiball).

```
lda   #$8
sta   LAMP_ROW   ;; Ground selected row
clra
sta   LAMP_AUX   ;; Clear auxiliary rows
lda   #$40
sta   LAMP_COL   ;; Drive selected column
```

Although playfield switches are handled by the matrix grid, some frequently used buttons are connected to a dedicated connector. This allows the CPU to directly address this input without having to scan the entire input matrix. These switches are user buttons and End-Of-Stroke.

The E.O.S switch prevents foldback when the player has the flipper energized to capture balls. When the Game CPU detects that this switch is open, it stabilizes the position of the selected flip by reducing the pulse applied to the coil.

## 1.2 The Backbox

The Backbox contains all the electronic circuits controlling playfield's behaviour. We will focus on this very part throughout this paper.

### CPU/Sound Board

The main board contains the Game CPU and the Sound circuit. The switches are directly connected to this board so that it is really simple for the CPU to fetch their values.

One of the main problems of this board is the battery location. Populated with a 3xAA battery holder to keep the RAM content alive, alkaline batteries are located on top of the CPU, ROM and RAM chip, which is critical when they will start to leak on this components. Before I started playing with this machine, I spend hours restoring and cleaning the PCB because of the corrosive leaking. To avoid deterioration, relocating this battery could be a smart idea.

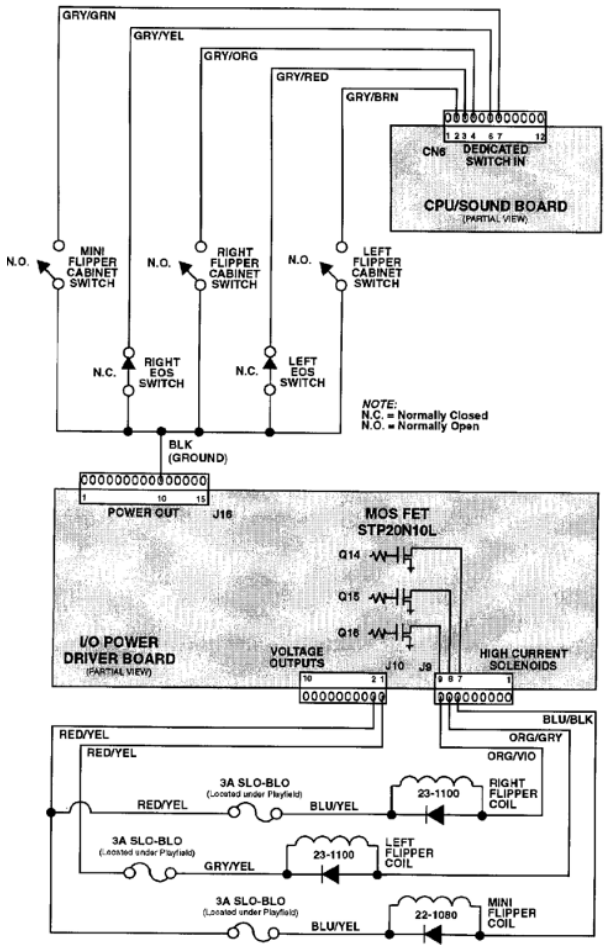

**Fig. 2.** Flippers wiring

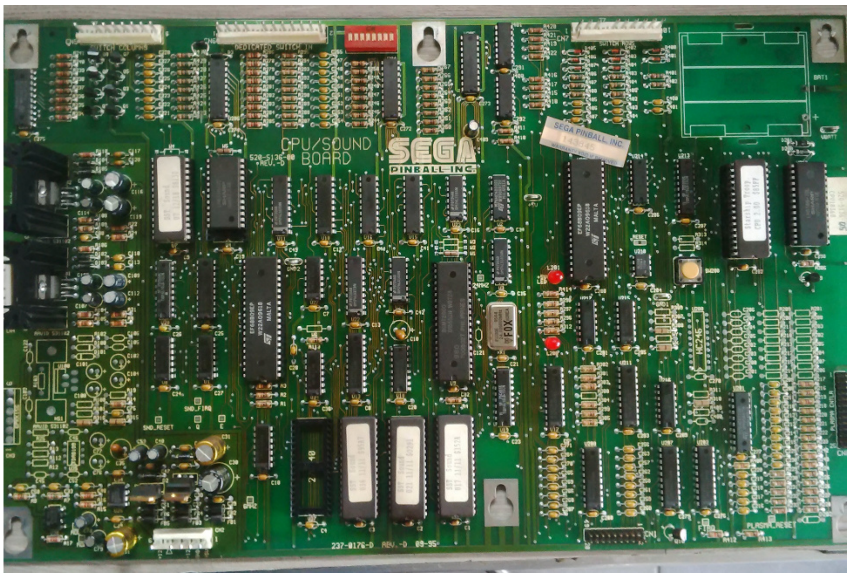
**Fig. 3.** CPU/Sound Board

### Display Controller Board

Like many pinball machines from the 90s, the backbox is equipped with an old school dot matrix display.

As the CPU Board, it is based on a Motorola `68B09E` with a dedicated 512MB UVPROM which contains the dot matrix display driver code and images that can be displayed on it. It communicates with the main board via a specific protocol.

To interface the raster display, the board uses a Motorola `68B45` (`68B45 CRTC` for 'cathode ray tube controller'). Although this chip was primarily designed to control the CRT display, it can also be used to generate correctly timed signal for a raster dot matrix display like in this case.

### I/O Power Driver Board

The IO Power Driver Board is an interface between the low current logic circuit and the high current playfield circuit.

The first part of this circuit consists of converting the alternative current provided by the transformer into exploitable direct current thanks to 5 bridges rectifiers.

The only electromagnetic relay is dedicated to the general illumination and is not controllable via the main CPU. The rest is driven by MOSFET power transistors which are designed to be able to handle high current in order to power playfield coils. Moreover, fuses are placed before each bridges rectifiers in order to easily help identifying where the problem comes from in case of failure.

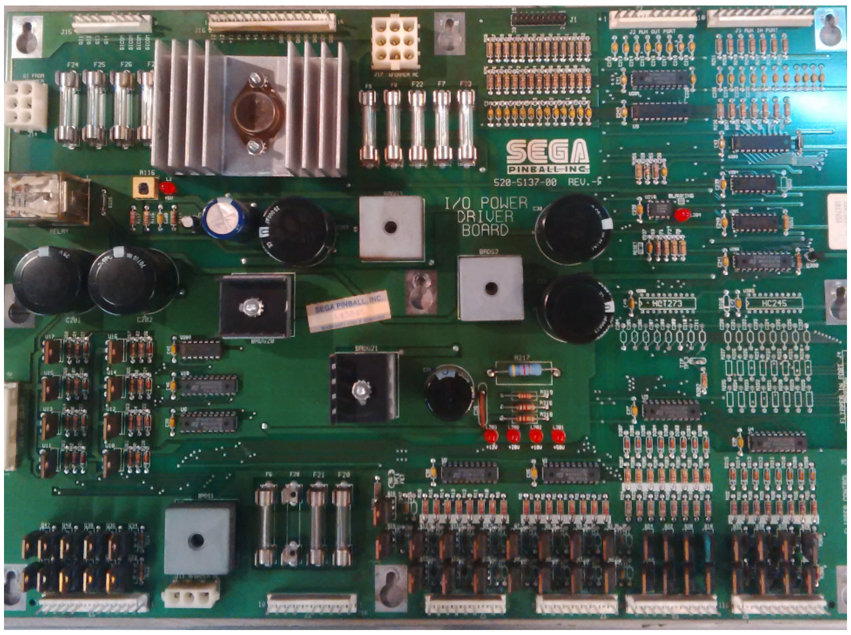

**Fig. 4.** IO Board

## 2 Upgrading the firmware

The title screen displayed in the dot matrix plasma display indicates that the firmware's version is `2.00`. However, an up-to-date image of this ROM exists in Internet Pinball Database which seems to be on version `2.01` according to the ascii string located at offset `$66D7`. Let's try to upgrade the pinball!

An almost suitable flash memory to replace the original UVPROM is the `A29040C`. The only mismatches on the pinout are the `A18` and `WE` pins. This is a minor problem since I fixed the PCB to match the `A29040C` layout.

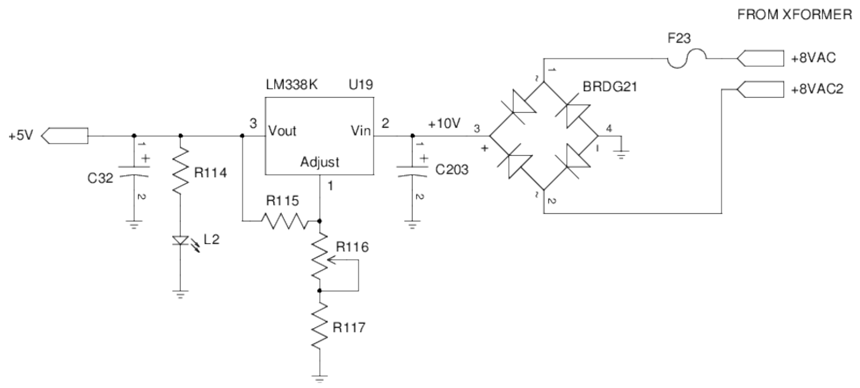

**Fig. 5.** IO Board Power supply

| IC Name | Type | Board Name | Loc. |
|---|---|---|---|
| Game ROM | 1MB | CPU / Sound Board | U210 |
| Voice ROM 1 | 4MB | CPU / Sound Board | U17 |
| Voice ROM 2 | 4MB | CPU / Sound Board | U21 |
| Voice ROM 3 | 4MB | CPU / Sound Board | U36 |
| Voice ROM 4 | Not Used | CPU / Sound Board | U37 |
| Sound EPROM | 512K | CPU / Sound Board | U7 |
| Display EPROM | 4MB | Display Ctrl Board | ROM 0 |
| Display EPROM | Not Used | Display Ctrl Board | ROM 3 |

**Table 1.** ROM Summary

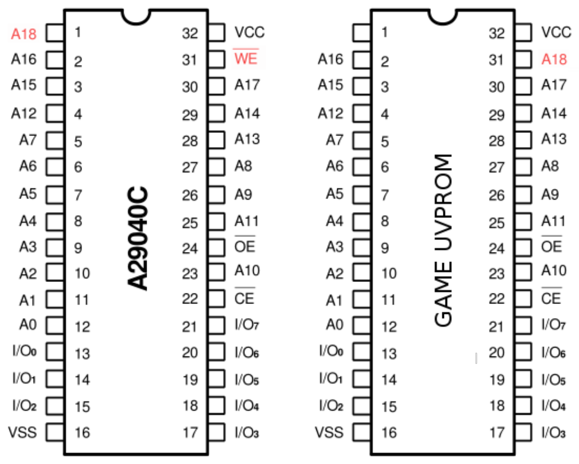

**Fig. 6.** Pinout mismatch

Burning the `A29040C` with the new firmware requires a flash memory programmer. I decided to craft one with an `Arduino mega 1280` based on an `AVR Atmega 1280` microcontroller. The large number of IO of this chip is essential to complete the programming protocol of the `A29040C`.

After successfully programming the flash memory, I was pretty disappointed when I noticed that the new ROM chip was still not working.

I thought that this UVPROM was able to store 512KB of data, just like `A29040C`. It took me a while to realise that the game is a 128KB ROM although the chip is designed to be connected to a 19 bit address bus. This means that the game's ROM simply ignores the value of `A17` and `A18` signals, which means that the game code is mirrored 4 times in the whole ROM address space.

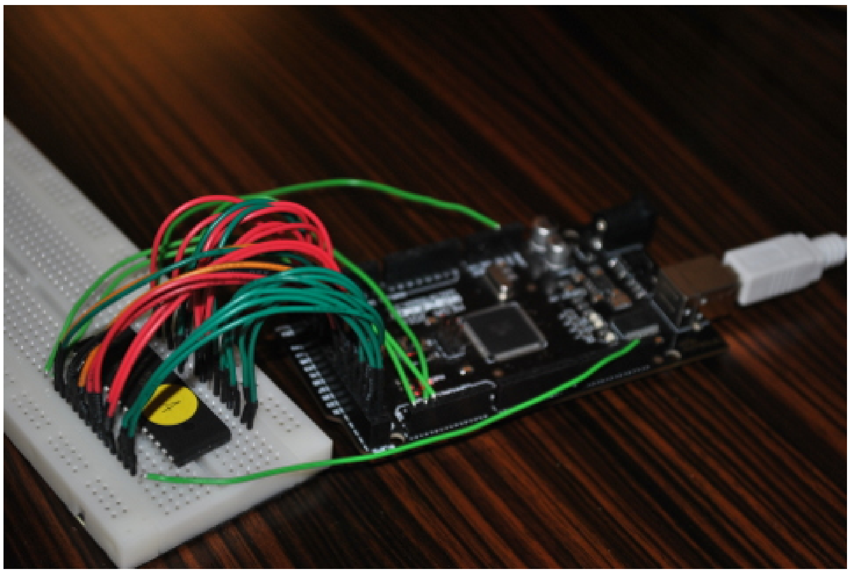

**Fig. 7.** Homemade Flash Programmer

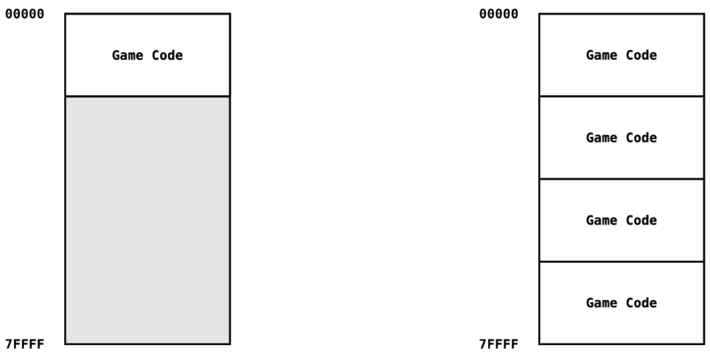

**Fig. 8.** Mirroring

## 3 Building a custom ROM

Now that we are able to substitute the original ROM with a custom flash memory, let's try to run our own code on this machine.

The first thing that we have to do in this case is to determine where the CPU will fetch its first instruction after a reset. According to the `6809` datasheet, the interrupt vector table (which contents the address of the **`reset`** event handler) is located at `0xFFFE`. However, this offset refers to the CPU address space, not that of the ROM chip. So, after a reset, which part of this memory is mapped at `0xFFFE`?

To answer this, it's essential to follow the address bus of the UVPROM. We then easily see that bits 14 to 18 of this bus are connected to 5-bit register (`U211`) while bits 13 to 0 are directly bound to CPU address bus.

This is a typical configuration to implement a bank system since the CPU address space is too narrow to map the entire ROM. That's why only one part of it (also called a *bank*) is mapped at a given time. The mapped bank is chosen by the `U211` register, called `XA`, and can be easily wrote by the CPU when a bank switching is needed.

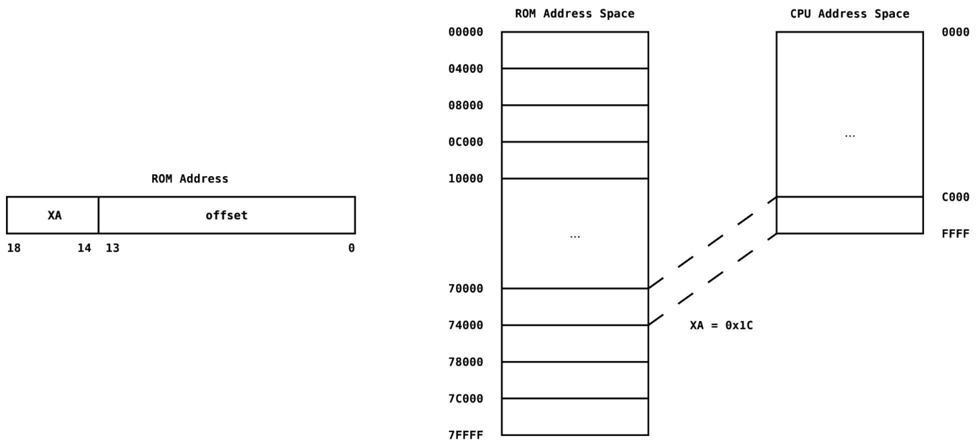

**Fig. 9.** Bank system

## 3.1 Finding address space

On this kind of device, it's always painful to debug the code running directly on the board. The only way to achieve it here is to trigger some visual element of the playfield in order to get a basic tracing of the execution flow.

As there is no IO port on the `6809`, all devices are memory-mapped. The question now is: where are they located?

First, let's focus on the address decoding circuit of the IO Board.

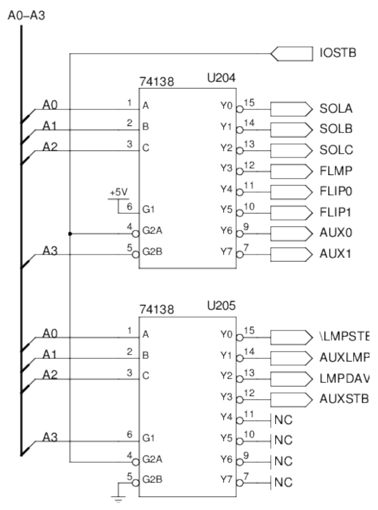

**Fig. 10.** IO Addressing

In order to simplify cascading, the `74138` multiplexer generates output only if the Boolean expression `G1 && !G2A && !G2B` is true. So, in this circuit, `U204` covers IO addresses from `0x0` to `0x7` and `U205` handles from `0x8` to `0xF`.

As we can see on this schematic, the question is: where does the `IOSTB` signal come from?

Following the wire, we can see that this control signal is generated by the CPU Board. It actually acts as a *chip select*: it means that this signal is used to indicates to the IO Board that we are addressing it.

To be more precise, the `IOSTB` is driven by the `U213` chip, a `PAL16L8` (Programmable Array Logic). This kind of integrated circuit is used to implement combinatoric logic expressions. This is widely used for address decoding.

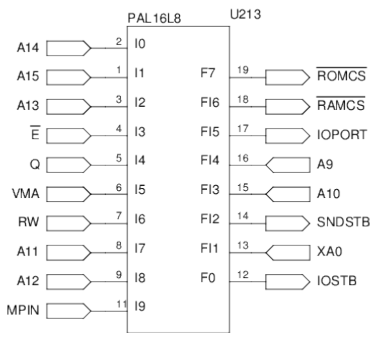

**Fig. 11.** PAL16L8

Dumping the logical expression programmed on this chip is essential to determine the actual CPU address space. One way to do it is to basically test all possible inputs and watch how outputs evolves according to input values. However, some of the `PAL16L8` pins can be considered as inputs as well as outputs. In this case, we can guess that `XA0`, `A9` and `A10` are used as input pins according to the rest of the circuit.

I desoldered the PAL, in order to prevent undesired side effect on the rest of the circuit, and used a simple *Arduino Uno* to generate the truth tables of all outputs.

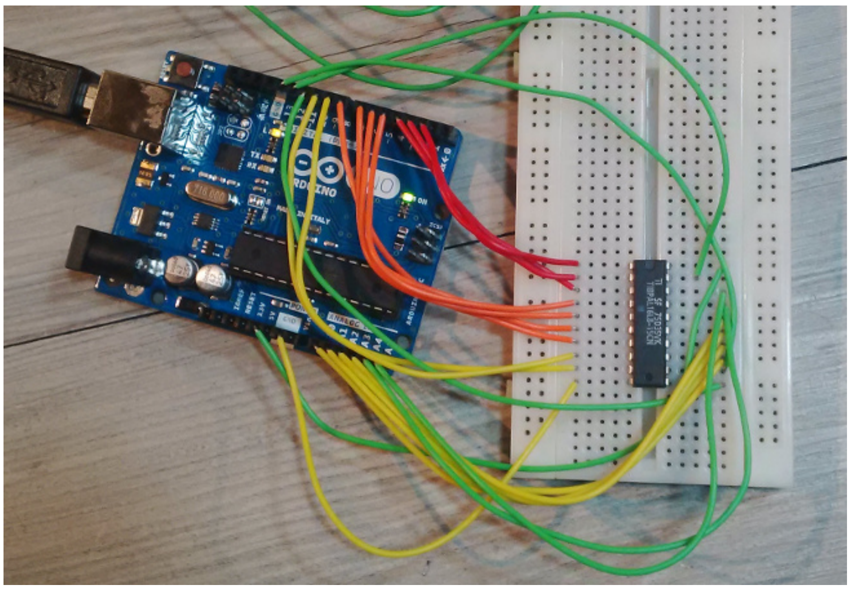

**Fig. 12.** Dumping the PAL16L8

Now, let's extract irreducible logical expressions from the recorded truth tables. As a matter of fact, these truth tables are significantly too large to apply the well-known Karnaugh map method to simplify the extended logical expression. This problem can be solved by using the https://pypi.python.org/pypi/electruth. It fully implements the Quine-McCluskey method which is perfectly suitable in this situation.

After a few hours of computation, I got these expressions, which are truly helpful in the address space determination process:

```
GAL16V8
U213
```

```
A15  A14   A13 /E     Q   VMA RW     A11    A12    GND
MPIN IOSTB XA0 SNDSTB A10 A9  IOPORT /RAMCS /ROMCS VCC

/ROMCS.T = A15 + A14 + IOPORT
/ROMCS.E = /E
RAMCS.T = A15 + A14 + A13 + A12 * A11 * A10 * A9 * /RW * /MPIN
/RAMCS.E = /E
IOPORT.T = A15 + A14 + /A13 + A12 + A11 + XA0
IOPORT.E = /E
IOSTB.T = /A15 * /A14 * A13 * /A11
IOSTB.E = /E

DESCRIPTION:
    Sega Whitestar Pinball
    U213 (Address space decoding)
```

Notice the `MPIN` input which is a signal generated by the cabinet door when it's open. So, the `PAL` restricts the access to a small part of the RAM when the coin door is closed. This section is actually used to store game settings that are only editable for maintenance purpose.

Here is the address space that I was finally able to discover according to the actual wiring:

- `0000-1FFF` : RAM
  - `0000-1DFF` : Read/Write Area
  - `1E00-1FFF` : Write Protected Area
- `2000-27FF` : IO (IOBOARD)
  - `2000` : HIGH CURRENT SOLENOIDS A
    - bit 0 : Left Turbo Bumper
    - bit 1 : Bottom Turbo Bumper
    - bit 2 : Right Turbo Bumper
    - bit 3 : Left Slingshot
    - bit 4 : Right Singshot
    - bit 5 : Mini Flipper
    - bit 6 : Left Flipper
    - bit 7 : Right Flipper
  - `2001` : HIGH CURRENT SOLENOIDS B
    - bit 0 : Trough Up-Kicker
    - bit 1 : Auto Launch
    - bit 2 : Vertical Up-Kicker
    - bit 3 : Super Vertical Up-Kicker
    - bit 4 : Left Magnet
    - bit 5 : Right Magnet
    - bit 6 : Brain Bug
    - bit 7 : European Token Dispenser (*not used*)
  - `2002` : LOW CURRENT SOLENOIDS
    - bit 0 : Stepper Motor #1
    - bit 1 : Stepper Motor #2
    - bit 2 : Stepper Motor #3
    - bit 3 : Stepper Motor #4
    - bit 4 : *not used*
    - bit 5 : *not used*
    - bit 6 : Flash Brain Bug
    - bit 7 : Option Coin Meter

        - `2003` : FLASH LAMPS DRIVERS
            * bit 0 : Flash Red
            * bit 1 : Flash Yellow
            * bit 2 : Flash Green
            * bit 3 : Flash Blue
            * bit 4 : Flash Multiball
            * bit 5 : Flash Lt. Ramp
            * bit 6 : Flash Rt. Ramp
            * bit 7 : Flash Pops
        - `2004` : *N/A*
        - `2005` : *N/A*
        - `2006` : AUX. OUT PORT (*not used*)
        - `2007` : AUX. IN PORT (*not used*)
        - `2008` : LAMP RETURNS
        - `2009` : AUX. LAMPS
        - `200A` : LAMP DRIVERS
    - `3000-37FF` : IO (CPU/SOUND BOARD)
        - `3000` : DEDICATED SWITCH IN
            * bit 0 : Left Flipper Button
            * bit 1 : Left Flipper End-of-Stroke
            * bit 2 : Right Flipper Button
            * bit 3 : Right Flipper End-of-Stroke
            * bit 4 : Mini Flipper Button
            * bit 5 : Red Button
            * bit 6 : Green Button
            * bit 7 : Black Button
        - `3100` : DIP SWITCH
        - `3200` : BANK SELECT
        - `3300` : SWITCH MATRIX COLUMNS
        - `3400` : SWITCH MATRIX ROWS
        - `3500` : PLASMA IN
        - `3600` : PLASMA OUT
        - `3700` : PLASMA STATUS
    - `4000-7FFF` : ROM
    - `8000-BFFF` : ROM (Mirror)
    - `C000-FFFF` : ROM (Mirror)

## 3.2 Handling reset circuitry

In this kind of real-time application, where a huge number of unpredictable events have to be handled, the risk of race condition cannot be fully faded.

Although the software is designed to be able to face any situations, the hardware has to be prepared to a faulty program. One of the simplest and more robust method is to use a *watchdog timer*. This consists of an autonomous timer charged to trigger a reset signal to the system if it reaches its initial point. The main idea here is to force the circuitry to be stopped if it does not correctly respond in order to prevent any damage from uncontrolled behaviour.

In most cases, the timer has to be fed by the software running on the CPU. So, if we want to run our own code on that machine, it's essential to implement as a subroutine the reset of the watchdog in order to stay alive.

In the *Whitestar* pinball, two distinct watchdogs have to be correctly handled. The first one is located on the CPU/Sound Board and is directly connected to the reset pin of the `6809`. *SEGA* engineers chose to use a `DS1232` chip (`U210`) which integrates all the features that are commonly used to monitor a CPU. So, in addition to a regular watchdog timer, this chip also provides a power monitoring and an external override which is actually designed to allow the use of a push button to force the CPU reset (`SW200`).

As the `TOL` pin of this chip is grounded, the `DS1232` continually watches the voltage applied on `Vcc` pin and triggers a reset signal if its value is under 4.7V. From a software engineer point of view, the important pin in that case is the *strobe input* (`ST`): it is used to reset the watchdog timer when a falling edge is applied to it.

On the CPU/Sound Board, this pin is connected to either clock signal (generated by `U2`) or `BSEL` signal according to the location of the jumper (`Wx` or `Wy`). As `Wx` was jumpered on my board, we can assume that the configuration in which `Wy` is fit was used during firmware development. So programmers were able to test their code without having to mind about the watchdog reset: this was automatically done by the clock signal. When the pinball was about to be released, calls to the watchdog reset subroutine were injected in appropriate parts of the firmware and the jumper was moved from `Wy` to `Wx`.

In my opinion, modifying the hardware by desoldering the jumper and resoldering it on `Wy` is a little bit too easy to solve this kind of problem. So, let's try to handle the watchdog timer with a suitable software subroutine.

The `BSEL` signal is generated when writing at address `0x3200` and is actually used as clock signal for the bank selection (`U211`). This is a clever way to get a nonintrusive watchdog reset subroutine: it's, in fact, hooked on the bank switching mechanism. The hardware designers probably thought it was a good idea to check the regularity of the code execution only by testing a periodic bank switching...

In our case, we do not need to switch from initial bank. The trick I used here is to write `0` in the `XA` register, so the bank is unchanged but the watchdog is fed anyway.

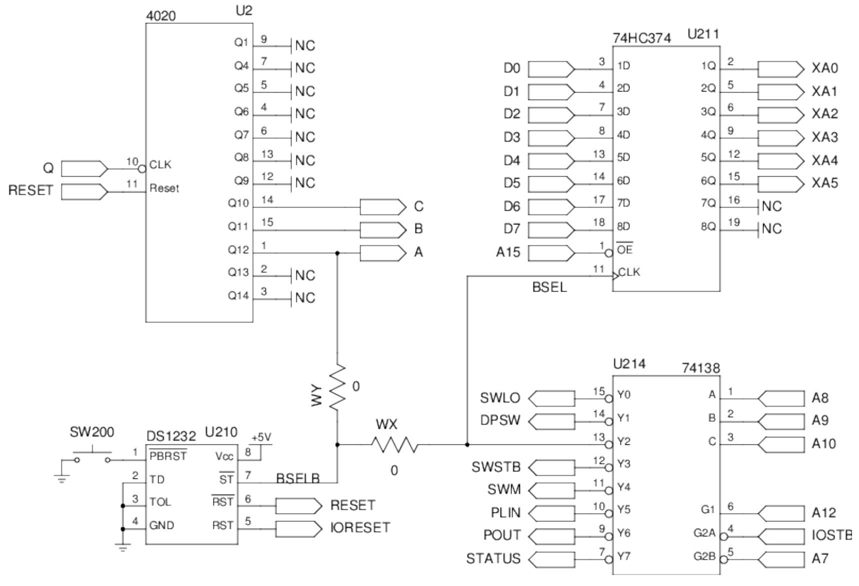

**Fig. 13.** CPU Board watchdog

The second watchdog is located on the IO Board. The chip used is still a `DS1232` (`U210`) but the wiring is a little bit different. Firstly, since there is no code running on that board, the reset pin of the `U210` is not connected to a CPU but to all registers (8-bit D flip-flop) which drive power transistors.

Secondly, there is no reset pushbutton on the IO Board. The `PBRESET` pin is connected to the `BRESET` signal coming directly from the CPU/Sound board. So, if the first `DS1231` triggers a reset signal, it automatically overrides the second watchdog timer and forward the signal to all IO Board components. However, this is not reciprocal: the IO Board cannot stops the CPU/Sound Board.

The *strobe input* of this watchdog is directly connected to the `DAV0` signal which is used to ground the first raw of the lamp matrix. This means that the firmware has to frequently scan it to keep the IO Board alive. Tricky, but not fully irrelevant since the lights are still blinking on this kind of arcade machine in order to keep the game catchy.

All of this reset circuitry have to be kept in mind when developing a firmware for this kind of platform.

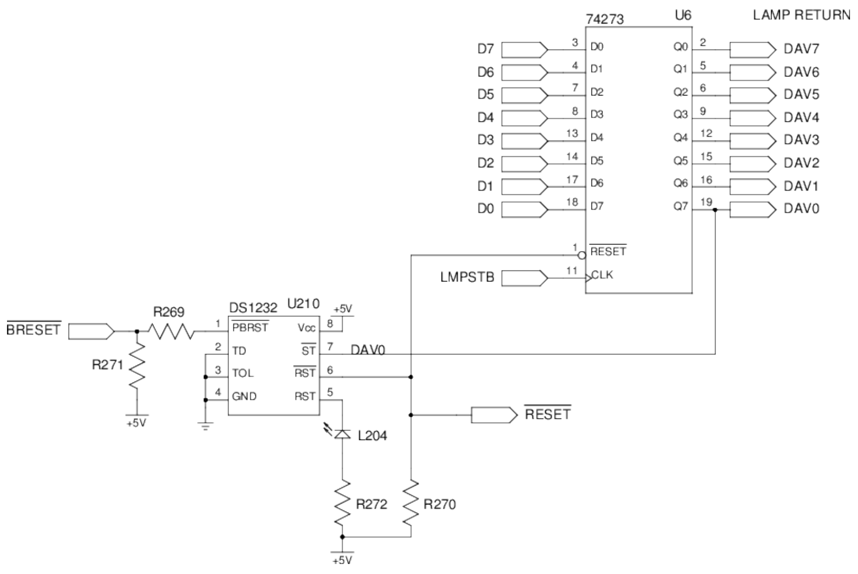

**Fig. 14.** IO Board watchdog

## 3.3 Firmware example

After many hours spent to reverse engineer the hardware part of this machine, I was finally able to print *LSE* on the 7-segment display of the playfield thanks to the code fetched from a custom flash ROM.

Here is the assembly code of my own basic firmware:

```
LAMP_ROW EQU $2008
LAMP_AUX EQU $2009
LAMP_COL EQU $200A
BANK_SELECT EQU $3200

;; CPU/Board Watchdog reset
wdr             .MACRO
                 clra
                 sta BANK_SELECT
                 .ENDM

;; Dummy delay subroutine
delay            .MACRO i
                 lda i
@1:              deca
```

```
                bne @l
                .ENDM

;; Entry point
                .ORG    0xC000
main:           ldx #lamps
                clrb
                stb LAMP_AUX    ;; Clear auxiliary rows
                incb            ;; Select first row

loop:           clra
                sta LAMP_ROW
                sta LAMP_COL    ;; Clear rows and colunms
                delay #$1F      ;; Dummy delay

                lda ,x+         ;; Fetch columns value
                sta LAMP_COL    ;; Set columns
                stb LAMP_ROW    ;; Ground selected row

                delay #$1F      ;; Dummy delay
                wdr             ;; Watchdog reset

                lslb            ;; Select next row

                bne loop        ;; Branch if the first 8 rows are not updated
                bcc main        ;; Branch if the 9th row is updated

                rolb
                stb LAMP_AUX    ;; Select the 9th row
                clrb
                bra loop

;; Lamp matrix values
lamps:
                DB $01, $00, $00, $00, $00
                DB $00, $1C, $B6, $9F, $00

;; Interrupt vector table
                .ORG    0xFFFE
reset:          DW main
```

tpasm is needed to assemble the preceding code and turn it into an Intel hex file using the following commands:

```
$ tpasm -P 6809 -o intel cpu.hex cpu.s
$ hex2bin ./cpu.hex
$ dd if=/dev/zero of=cpu.rom bs=16K count=32
$ dd if=cpu.bin of=cpu.rom bs=16K seek=31
```

## 4 Sound board

**Abstract.** A reverse engineering of a BSMT2000 DSP used on the audio circuit of an old-school pinball. An overview of the electronic design of this uncommon and discontinued machine will be presented before focussing on the peculiar conception of its sound board.

**Keywords:** Reverse Engineering, Hardware, Pinball Machine, Audio, DSP

### 4.1 Sound board overview

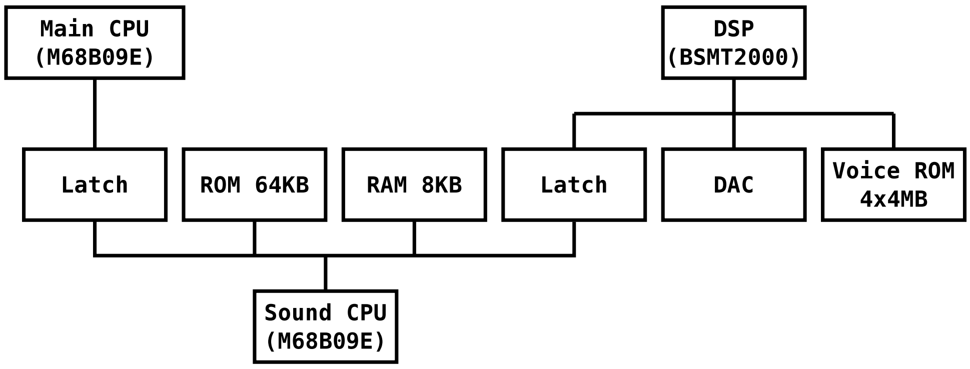

**Fig. 15.** Block diagram

The audio section consists of a Motorola `68B09E` CPU and a `BSMT2000` DSP. The sound playing is controlled by the main CPU which latches data to a dedicated register. The Sound CPU reads in this buffer commands and handles the interfacing to the `BSMT`. The DSP can read audio samples stored in the four dedicated 4MB EEPROMS and mixes it to a background melody. The data stream outputted by the DSP is then serially shifted into a stereo 16-bit Digital to Analog Converter (DAC). Finally, the analog signal is filtered and amplified before being applied to the speakers.

#### First look at sound CPU wiring

The sound board is entirely driven by the `68B09E` CPU. In order to reverse the behaviour of this circuit, it is a good idea to see how componants are exposed from the point of view of the sound CPU.

As the main CPU, address decoding is achieved using a `PAL16L8` as shown on Figure 16. Dumping `U26` configuration could be performed as explained on section 3.1. However, this trick implies desoldering chip and could be very long to compute. Reversing the code stored on sound CPU ROM in order to guess the address space is a better idea due to the reasonable size and the simplicity of the sound card firmware.

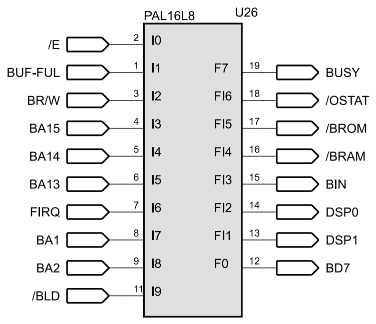

**Fig. 16.** Sound CPU address decoding

Unlike the main CPU ROM, the sound CPU ROM is not banked. Taking into account the fact that ROM size is 64KB, this memory perfectly fit the address bus width. This would mean that no spaces are left for other devices on the address space which is not conceivable. Opening the ROM content on an hexadecimal editor shows that the first 16KB are blank and some valid `6809` instructions can be disassembled above `0x4000`. This proves that this memory is actually not fully mapped to the sound CPU address space and so, some part of it will never be accessible. We guess that this design has been chosen to simplify relocation of addresses referenced on sound CPU code since the ROM is identity mapped. According to the disassembled code, the following lower mapping can be easily deducted:

– `0x0000-0x1FFF`: RAM
– `0x2000`: Status Register (`OSTAT` signal)
– `0x2002`: Main CPU / Sound CPU Command Register (`BIN` signal)
– `0x2006`: DSP Status (`/BLD` signal)

Things seem to get a bit more tricky for addresses above 16KB. Useful data can be found on the ROM from `0x4000` to `0xFFFF` such as code, read-only data structures and interrupt vector. However, it seems that sound CPU stores data on `U16` and `U11` by writing from `0xA000` to `0xA0FF` and on `U15` by writing at `0x6000`. Since the `PAL16L8` is taking `BR/W` signal used to indicate the opration type (read/write), it's perfectly possible to admit a different address space depending on the CPU operation. It is here used to overlap ROM space and DSP control space on this relatively restrained address space. The higher mapping can then be defined as:

– During `read` operation:
  • `0x4000 - 0xFFFF`: ROM
– During `write` operation:
  • `0x6000`: DSP Command (MSB)
  • `0xA000-0xA0FF`: DSP Command (LSB)

## 4.2 Interfacing sound board to main CPU

### Hardware interface

In order to indicate basic status information for the rest of the board, the sound CPU can write on a *status register* defined by two D flip-flop as shown on Figure 17. The first bit, is used to indicate to the main CPU that audio card successfully finished his initialization phase and is ready to process some commands. The second bit, mapped on the bit 7 (BD7), is wired to `RESET` pin of DSP and is triggered during the initialization or when `BSMT` is not responding.

```
        lda     #$80
        sta     IO_STATUS ;; Reset DSP
        cla
        anda    #1
        sta     IO_STATUS ;; Indicate to Main CPU that audio card is ready
```

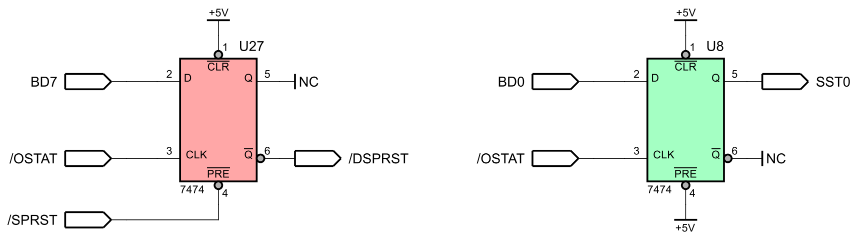

**Fig. 17.** Sound CPU status register

The sound calls are made by the main CPU by writing on the `U5` register (Figure 18). In order to inform the sound CPU that data is available, the circuitry defines the `BUF-FUL` signal which is set when the main CPU is writing on the command register using the `SNDSTB` signal. In the other hand, the sound CPU drives the signal `BIN` when it needs to read the instruction. According to code reversed from the sound CPU ROM, the reading is performed during initialization and during `FIRQ` handler execution: the sound CPU is periodically checking the content of the command register. The reading implies the driving of the `U5` content on the sound CPU data bus by grounding the `/OE` (Output Enable) pin of `U5`. Moreover, some side effects are associated with the reading operation: the `BUF-FUL` and `FIRQ` signals are cleared thanks to `U8` and `U1` latches. This means that command is marked as consumed and a new `FIRQ` can be triggered by the next rising edge of the FIRQ clock. Of course, the `BUF-FUL` is cleared when the sound board is reset using `SNDRST`.

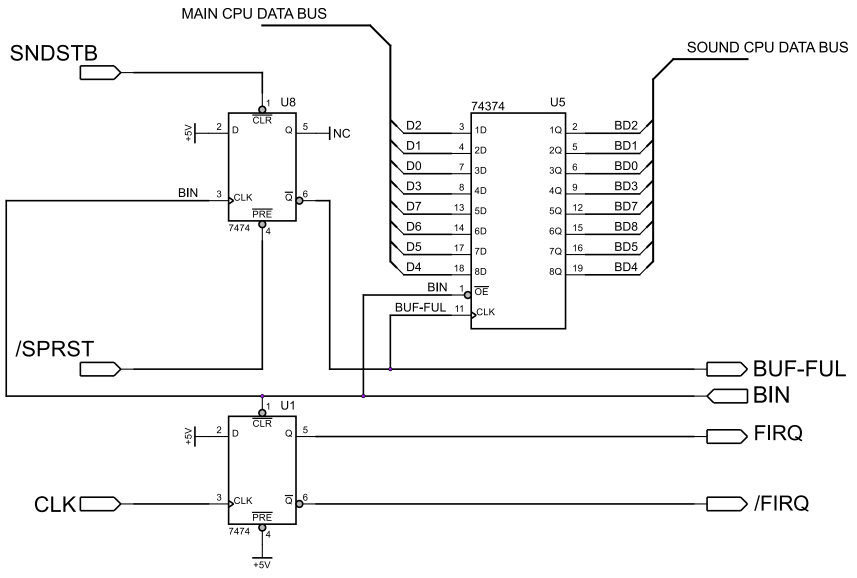

**Fig. 18.** Main CPU / Sound CPU interface

### Handling CPU commands

As we could expect from the wiring, main CPU commands are fetched when the sound CPU receives an FIRQ.

The FIRQ handler is quite simple: it increments timers used by the rest of the application and reads into U5 which clears the FIRQ signal. If the command is different from the previous one, it is enqueued into a dedicated ring buffer located at 0x0067 and defined as:

```
struct cmd_ring_buffer {
    uint8_t begin;
    uint8_t end;
    uint8_t data[16];
};
```

The software's main loop can now dequeues the command and jump to the corresponding handler. Each command is defined by a structure which contains a callback index and pointers to data which are not relevant at this point. Those command descriptors are arranged on two separate arrays and are defined as:

```
struct cpu_cmd {
    uint8_t     callback_idx;
    uint8_t     unk0;
    uint16_t    mask;
    void        **data;
};
```

As imposed by the circuitry, a command is only a 8-bit word which is quite restrictive. That's why the software defines two banks extending the number of commands to 512. The main CPU can then select a bank by sending 0xFD or 0xFE commands.

It is quite hard to deduce the exact behaviour of each command from this point. A good approach to go further is to understand how the DSP operates and then extrapolate CPU commands purposes from it.

## 4.3 Driving the Digital Signal Processor

### Sound CPU/DSP hardware interface

The Figure 19 illustrates the interface between the sound CPU and the DSP. According to this wiring, a DSP command is composed of an 8-bit address stored in U10 and a 16-bit data stored in U15 and U16. The writing of a DSP command cannot be atomic since U15 and U16 are connected to the low byte of the sound CPU data bus. So, when sending a DSP instruction. The first step is to write on U15 using DSP1 signal in order to send command high byte. The second step is to write the command's low byte using DSP0 signal. The least significant byte of data bus and of the address bus will be respectively captured on U16 and U11.

The wiring of U27 shows that an IRQ is sent to the sound CPU when the DSP consumes the command and so is ready to get a new command. Moreover, /BLD signal seems to indicate that a DSP instruction is still pending. With this kind of circuitry, we can suppose that DSP is periodically reading U10 register and then read U16 and U15 registers if U10 is different than 0xFF.

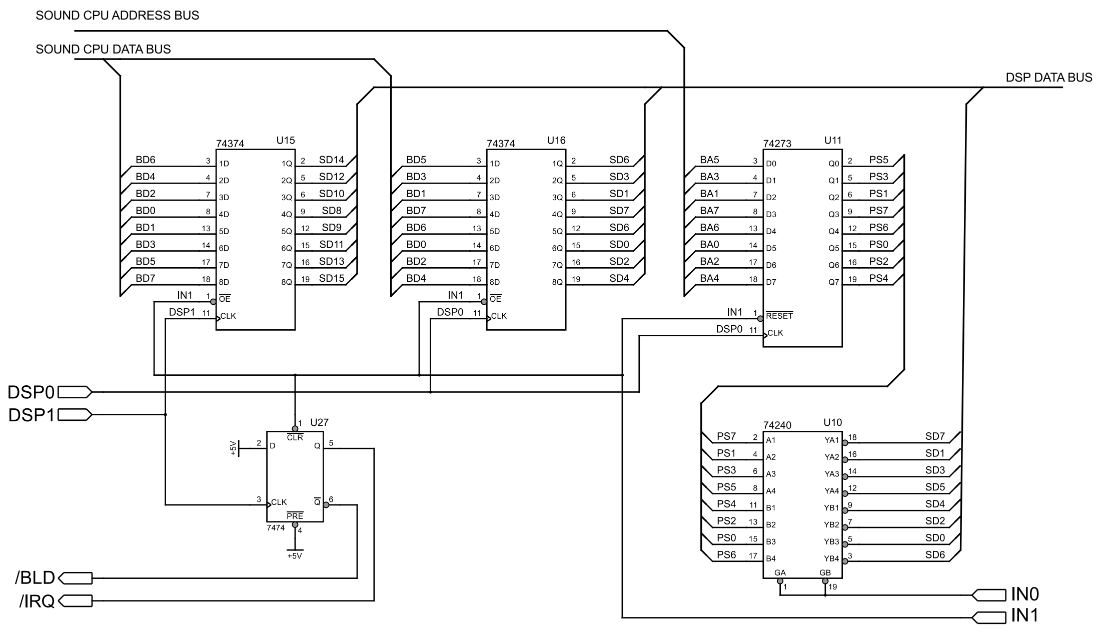
**Fig. 19.** Sound CPU / DSP interface

### BSMT2000 wiring

The digital to analog conversion is performed by a Philips `TDA1543`. This integrated DAC is one of the first which supports $I^2S$ as input format. Figure 20 illustrates the glue logic needed to convert the data stream generated by the DSP to a correct $I^2S$ stream. This is achieved by using two 8-bit shift registers (`U23` and `U24`). Samples are simply written to this registers using `OUT3` signal and are shifted to the DAC using `SCLK` 24MHz clock signal generated by the `BSMT2000`. `WS` signal is toggled to latch the right and left channel sound data into the DAC. Its value is captured during writing operation on shift registers from `SA2` (third bit of DSP address bus).

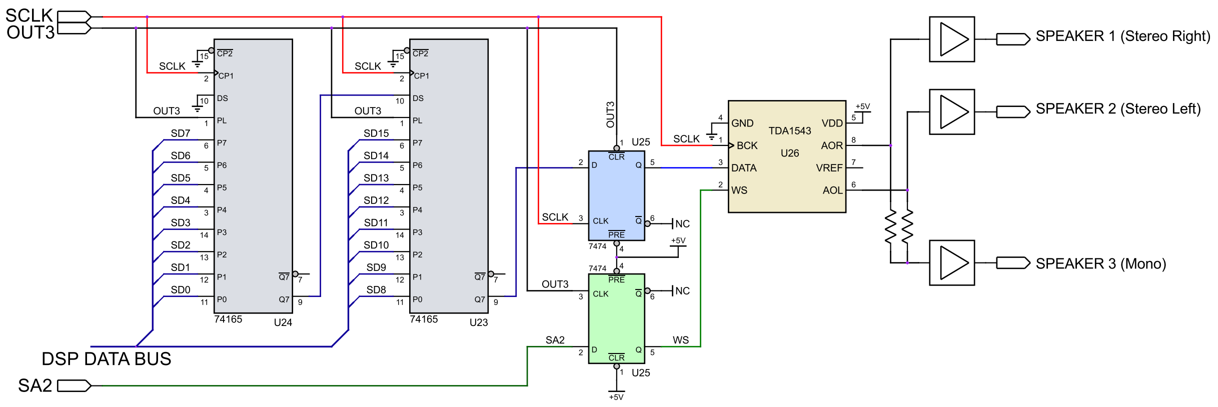
**Fig. 20.** DSP / DAC interface

### Dumping the BSMT2000 Mask ROM

Although the `BSMT2000` is undocumented, it's well-known that this chip is a masked-ROM version of the `TMS320C15` Texas Instruments's DSP from 1987. According to the wiring on the sound board, the pinout is identical to the 40-Pin DIP version of the `TI`'s DSP. In order to focus the reverse engineering on this particular chip, it's essential to isolate it from the rest of the sound board. This way, we avoid all side effects caused by external circuitry which can rise some unwilling behaviour during the test process. Moreover, it can be useful to provoke some unusual and controlled events to the chip in order to deduce design details. The best example is the dumping of the internal program memory.

The chip is clocked at 24 MHz which is actually too high to use a microcontroller to probes or generates the signals needed to correctly operates. The use of an FPGA is the most convenient sane way to simulate the sound card which host this DSP. The testbench is based on a *DE0-Nano*, an *Altera Cyclone IV* developpement board.

A first analysis of signals driven on the address and control busses shows that the DSP is periodically reading on `U11` register (`IN0`) as expected regarding the sound CPU code and the external circuit.

The original `TMS320C15` can be used in two separate modes which define the location of the used program memory. The current mode is selected using the `/MP` pin:

- Microcomputer mode (`/MP = 1`): Fetch instructions from internal program memory
- Microprocessor mode (`/MP = 0`): Fetch instructions from external program memory

When used on the pinball sound card, the `/MP` is connected to 5V, selecting the Mask ROM as program memory. Grounding this pin on the custom testbench allows us to execute basic `TMS320C15` instructions from FPGA internal RAM. This proves that `BSMT200`'s features reside on the program stored on Mask ROM: the glue logic seems to be identical to a real `TMS320C15`.

Of course, there is no programming protocol allowing program memory reading as some microcontrollers feature. However, this kind of DSP are based on a modified Harvard architecture which means that the program can read itself using specific instructions. In our case, `TBLR` instruction is a good candidate.

The trick here is to inject this instruction using external memory in order to read internal ROM. Although `/MP` signal is not designed to be toggle during execution, experimentations seem to show that this pin is simply controlling a multiplexer on data bus selecting the corresponding program memory and can then be switched during execution. As shown on figure 22, toogling this pin to MP mode directly after fetching `TBLR` forces the DSP to switch to mask ROM during execution of this single instruction allowing the reading from the mask ROM to data memory. The `/MP` signal must then be grounding in order to continue fetching from external memory. An `OUT` instruction can then be used to read the mask ROM word from data memory and outputting it to data bus. Although this trick seems to be simple to perform, very tight timing on `/MP` signal has to be respected to allow Mask ROM reading.

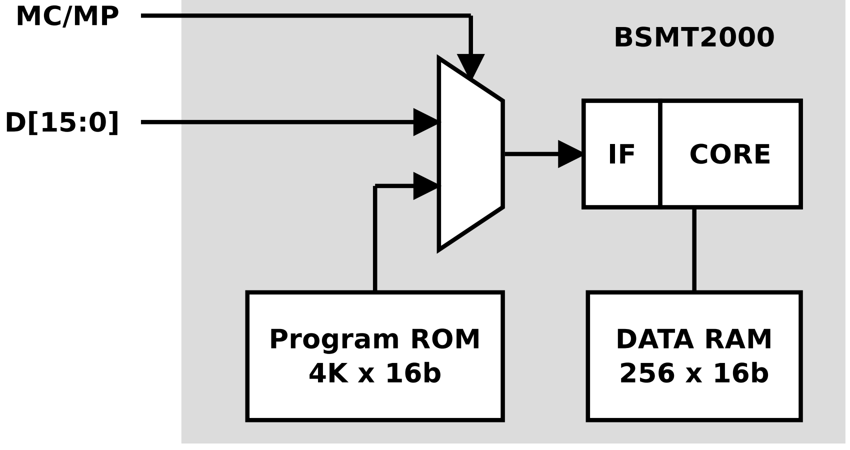

**Fig. 21.** BSMT2000 program bus

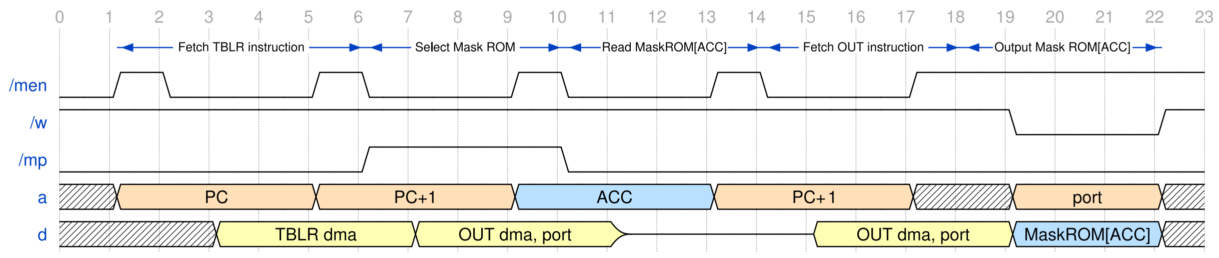

**Fig. 22.** BSMT2000 dump waveform

This behaviour can be implemented using FPGA. *Altera Cyclone IV* provides an efficient way to integrate JTAG-editable memories using *M9K* cells. As shown on figure 23, this is used on this design to define external program memory and a shadow mask-ROM used to store words read from DSP ROM. The program memory content is defined using this Memory Initialization File (`.mif`):

```
WIDTH=16;
DEPTH=64;

ADDRESS_RADIX=HEX;
DATA_RADIX=BIN;

CONTENT BEGIN
    0: 0111111000000001; -- LACK 1      ;; ACC <- 1
    1: 0101000000000000; -- SACL 0      ;; DATA[0] <- ACC
    2: 0110101000000000; -- LT 0        ;; T <- DATA[0]
    3: 1000000000000001; -- MPYK 1      ;; P <- 1 x T
    4: 0111111110001001; -- ZAC         ;; ACC <- 0
    5: 0110011100000000; -- TBLR 0      ;; DATA[0] <- PROG[ACC]
    6: 0101000000000001; -- SACL 1      ;; DATA[1] <- ACC
    7: 0100100100000001; -- OUT 1, 1    ;; IO[1] <- DATA[1]
    8: 0100100000000000; -- OUT 0, 0    ;; IO[0] <- DATA[0]
    9: 0111111110001111; -- APAC        ;; ACC <- ACC + P
    A: 1111100100000000; -- B 5
    B: 0000000000000101;
END;
```

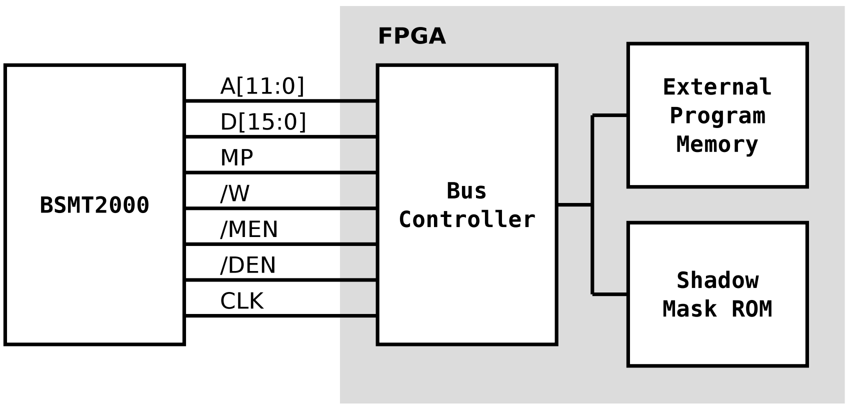

**Fig. 23.** BSMT2000 testbench block diagram

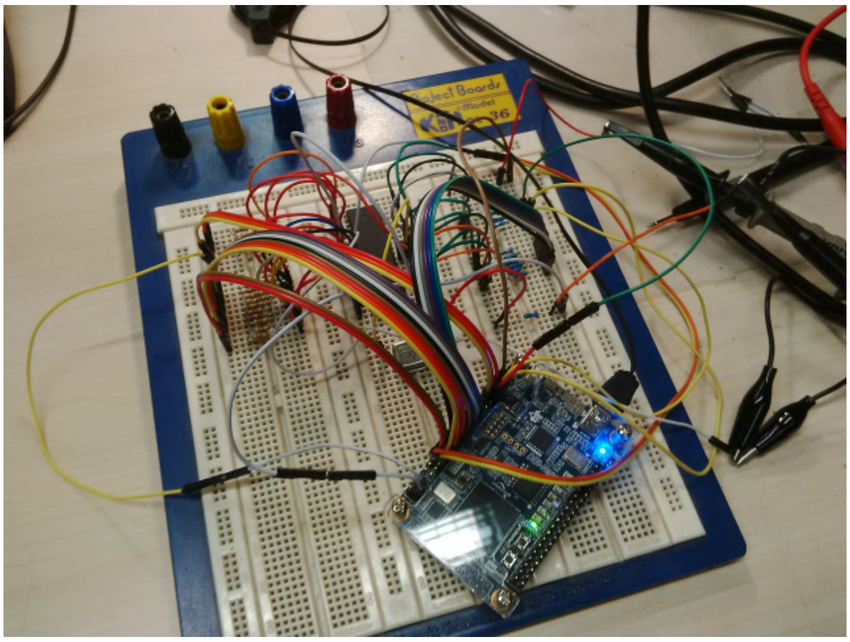
**Fig. 24.** BSMT2000 testbench

**BSMT2000 firmware**

Before reversing BSMT2000's firmware, it is necessary to examine the circuitry in order to see how the DSP can interact with the rest of the board and in particular how it can fetch samples from voices EEPROMs and how it can output signals to the DAC.

As the DSP is configured in *microcontroller* mode, it can only communicate with devices by using IN/OUT instructions. TMS320C1x's instruction encoding limits the IO space width to 6 bits. The address decoding is performed by a 1-of-8 demultiplexer (74ALS138) and it is quite simple to deduce this IO mapping:

- IN 0: Sound CPU command address
- IN 1: Sound CPU command data
- IN 2: EEPROM data
- OUT 0: EEPROM address
- OUT 1: EEPROM bank
- OUT 3: Sample out (Left)
- OUT 7: Sample out (Right)

The first thing the DSP firmware do after resetting is to disable interrupts, read into the first IO port and jump to the corresponding subprogram.

The sound CPU firmware seems to be aware of this behaviour according to this subroutine which reset the DSP and configure it to mode 1:

```
OSTAT   EQU $2000
DSP1    EQU $6000
DSP0    EQU $A000

init_dsp:
            ;; Reset DSP
            lda #$80
            sta OSTAT    ;; Set DSPRST

            ;; Compute command address according to the desired DSP mode
            ldb #$FE     ;; We need to write the complement of the actual value
```

```
                    ;; due to U10 inverting output tri-state buffer
                    ;; Here, we select mode 1
        ldx #DSP0
        abx         ;; x <- b + x

        ;; Select DSP mode by writing 0 at DSP0 + ~mode
        clra
        sta #DSP1   ;; MSB
        sta ,x      ;; LSB

        ;; Start DSP
        sta OSTAT   ;; Clear DSPRST
        rts
```

The rest of this study will exclusively consider the mode 1. Modes 0, 5, 6 and 7 are similar to mode 1 and others are for testing purposes.

The mode 1 main loop is basically composed of four stages. The first one is to fetch samples from the voices EEPROMs. Addressing those memories must be performed by selecting a bank on U22 and latching an offset on U12 and U13. The bank value is composed of two parts: bits 3 and 4 are used to select one of the four ROMs and bits 0 to 2 select a 64KB bank into the corresponding chip. In this case, audio samples are simply encoded using 8-bit mono PCM at 8KHz.

The second stage is about decoding the ADPCM channel. Although eleven channels are working on PCM encoded samples, the twelfth provide an custom ADPCM decoder which enables voice signal compression. However, in the case of *Starship Troopers* pinball, voices EEPROMs only contain PCM samples. We can then suppose that this part of the DSP firmware is not used and we will detail this part here.

The purpose of the third stage is to mix the different channels into one sample that can be outputted to the DAC. The following code snippet exhibits the fact that the TMS320C1x is perfectly designed for this kind of operation:

```
    ZAC                     ;; ACC <- 0
    LT VOLUME1              ;; T <- DATA[VOLUME1]
    MPY SAMPLE1             ;; P <- T * DATA[SAMPLE1]
    LTA VOLUME2             ;; ACC <- ACC + P; T <- DATA[VOLUME2]
    MPY SAMPLE2             ;; P <- T * DATA[SAMPLE2]
    ...
    LTA VOLUME12            ;; ACC <- ACC + P; T <- DATA[VOLUME12]
    MPY SAMPLE12            ;; P <- T * DATA[SAMPLE12]
    APAC                    ;; ACC <- ACC + P
    SACH 0, TMP             ;; DATA[TMP] <- ACC[31:16]
    OUT DAC, TMP            ;; IO[DAC] <- DATA[TMP]
```

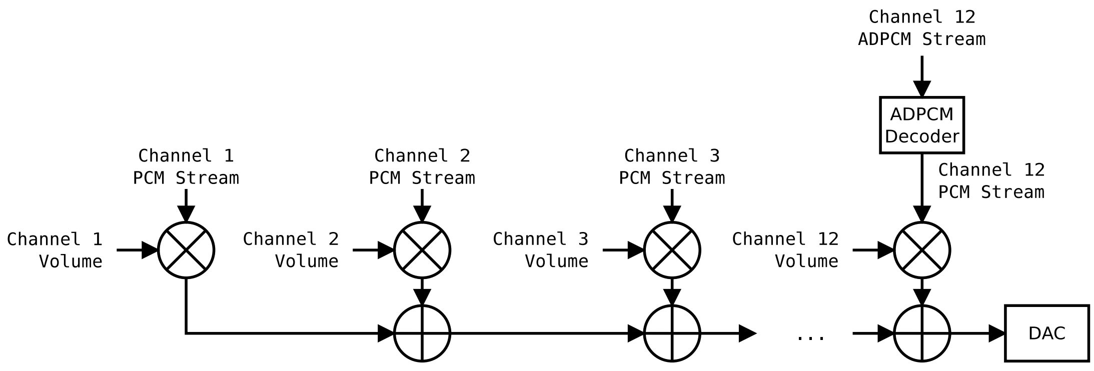

**Fig. 25.** Mode 1 block diagram

The last stage is dedicated to handle commands from sound CPU. This is suprising how this part is simplistically designed. The firmware is periodically fetching the command and then writes its value (`IN 1`) into arbitrary address (`IN 0`) without any verification. It means that the sound CPU can write anywhere in the DSP data memory. Notice that as the writing is unconditionnal and as the reset value of the `U11` register will be read as `0xFF`, the last word of the DSP data memory will perpetually filled with garbage values when no command is pending.

```
        BIOZ fetch      ;; Jump to 'fetch' if TST pin is active

        NOP             ;; Burn CPU cycles
        NOP             ;;
        NOP             ;;
        B next

fetch:  IN 0, 60        ;; DATA[60] <- IO[0]
        LAR AR0, 60     ;; AR0 <- DATA[60]
        IN 1, *         ;; DATA[AR0] <- IO[1]

next:   ...
```

As you might notice, access to command registers is only conditioned by the `BIOZ` instruction. It basically jumps if the `TST` physical of the chip is active. As shown on Figure 26, this pin is wired to the `CLKOUT` signal which is clocked at 1/4 `CLKIN` (main DSP clock) frequency. We can suppose that this mechanism is setup to limit the number of reading on the command registers to only one quarter of the main loop iterations. This requires that each iteration consume exactly the same number of cycles which explains the usage of `NOP` instructions to burn CPU cycles in some cases.

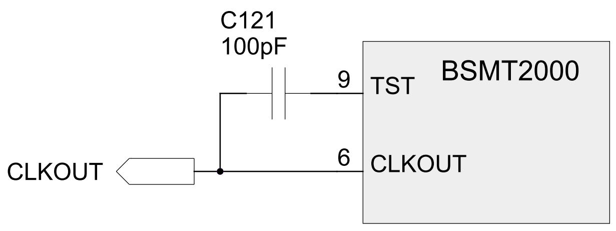

**Fig. 26.** `TST` pin wiring

The DSP firmware seems to segment the data memory into several ranges which defines channels configurations and that can be written by the sound CPU:

- `0x0-0xA`: Channel playback positions
- `0x16-0x20`: Channel rates
- `0x21-0x2B`: Sample limits
- `0x2C-0x36`: Sample loops
- `0x37-0x41`: Sample bank
- `0x42-0x4C`: Channel right volume
- `0x4D-0x57`: Channel left volume
- `0x58-0x62`: Sample data

Although, this kind of command handling may seem risky because of the lack of verification, the harvard architecture adopted by the `TMS320` avoids code rewriting. Moreover, the rest of the address space is simply not interpreted by embedded code which relativize the consequences of corrupted data.

The DSP firmware is actually quite simple. And even a bit too simple. His only purpose here is to fetch samples at a given rate and mix them together. But now that we know how the DSP can be controlled, it could be a good idea to go further into the sound CPU reverse-engineering.

## Back to the Sound CPU firmware

The communication with the DSP is ensured by the subroutine located at `0x55B8`. This function waits until DSP status register (`0x2006`) is clear and then write into DSP command register.

This code is called by several subroutines which are never directly referenced in the rest of the firmware. Instead, they are arranged on a jump table structure as follow:

```
#define MAX_CHAN 12

struct dsp_ops {
    void (*set_fixed_volume[MAX_CHAN])();
    void (*set_rate[MAX_CHAN])();
    void (*set_default_rate[MAX_CHAN])();
    void (*stop_playing[MAX_CHAN])();
    void (*load_pcm_sample[MAX_CHAN])();

    void (*op5[MAX_CHAN])();
    void (*op6[MAX_CHAN])();
    void (*op7[MAX_CHAN])();
    void (*op8[MAX_CHAN])();
    void (*op9[MAX_CHAN])();
};
```

This kind of *object-oriented* pattern is useful to define different behaviours for each channel exposed as a single operation. For instance, the ADPCM channel volume is not configured the same way as the other PCM channels.

By using this bottom-up approach it is easier to deduce the behaviour of the audio sequencer which is really similar to what was generated by old school music trackers. It is implemented as a simple virtual machine computing audio pattern from a dedicated bytecode. As shown on Figure 27, patterns are referenced into CPU command definition structure. The third field of this structure (previously defined on section 4.2) is a 12bit mask defining on which channels the pattern must be played.

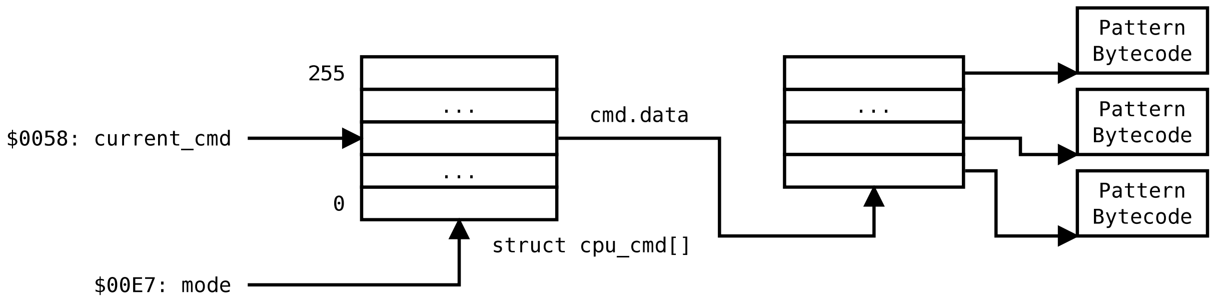

**Fig. 27.** CPU command definition

The bytecode instruction encoding is quite simple: the first byte is an operation code following by a variable number of arguments depending on the operation. For instance, the `05` opcode is used to load a PCM sample into a channel. As argument, it takes the address of a sample descriptor structure described as:

```
struct pcm_sample {
    uint16_t    base;
    uint16_t    limit;
    uint16_t    loop_start;
    uint8_t     unk;
    uint8_t     bank;
};
```

A total of 43 instructions are implemented allowing control of playback rate , volume, timing and bytecode execution. The following listing describes the pattern played on a channel when the ball hits a target on the playfield. It basically load a PCM sample into the DSP and play at fixed volume and rate. The main CPU can play this pattern by sending `0xAD` command when the first sound CPU mode is selected.

```
    ;; PCM sample description
828B: 00 00         ;; pcm.base
                    ;; sample starts at 0x0000

828D: 47 AC         ;; pcm.limit
                    ;; sample finishes at 0x47AC

828F: 47 86         ;; pcm.loop_start
                    ;; sample playing must loop at 0x4786

828F: 3C

828F: 03            ;; pcm.bank
                    ;; sample is located on bank 3 of U17 EEPROM

    ;; Explosion pattern bytecode
91DE: 05 81 8B      ;; load pcm sample described at 0x818B into channel
91E1: 09 01         ;; set channel volume
91E3: 01 1D 01 6D   ;; set channel rate, start sample playing
                    ;; and wait 7425 ticks (0x1D01) => 2.53 seconds
91E7: 0F            ;; free the channel and stop sample playing
```

This mechanism may seem overdesigned for playing single PCM sample but it's really powerful when it is about synthesizing background music or other complex melody which takes too many spaces when encoded in PCM.

When a command is called by the main CPU, a new instance of the virtual machine is started for each channels masked by the corresponding command descriptor. Each execution instance is represented by this structure:

```
struct track {
    struct track    *next;
    struct track    *prev;

    void            *instruction_pointer;   // Address of the next bytecode
                                            // instruction

    uint16_t        counter;                // Used for operation timing
    uint16_t        last_timestamp;

    uint8_t         next_instruction;
    uint8_t         type;                   // 0: Background track
                                            // 1: Foreground track
    uint8_t         channel_id;

    uint16_t        unk0;
    uint16_t        unk1;
    uint8_t         unk2;
};
```

Those track structures are arranged into a double-linked list allocated in-place from 0x03C0 to 0x05B8. As no heap-based memory allocator is provided, list nodes are allocated in-place as represented on Figure 28. Notice that the first node is never allocated and so is used as a sentinel for the free list.

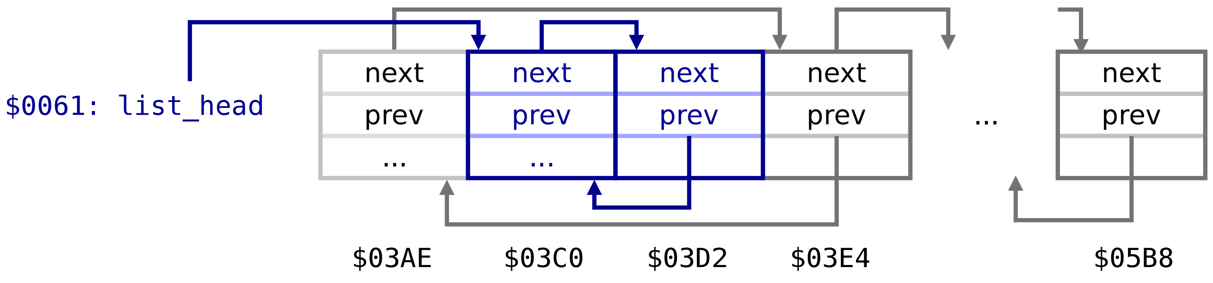

**Fig. 28.** Track allocation

The virtual machine scheduler is simply a round-robin which executes only one bytecode instruction per execution time. As audio patterns need timing between some operations, the counter field of the track structure is used to retard the execution of the next bytecode instruction. It is relative to ticks updated by the FIRQ handler.

The callback index contained on command descriptor structure can refer two types of track playing:

– 0x2: Play foreground track (Remove previous foreground tracks on masked channels)
– 0xB: Play background track (Remove all previous tracks)

Both of them inserts tracks on the playing list with specific pattern bytecode. They only differ by the way they clean the list before inserting new tracks and how they will be interpreted by the virtual machine. As a matter of fact, 0xB commands are always provided with 0x7FF which means that background music is allocated on all PCM channels.

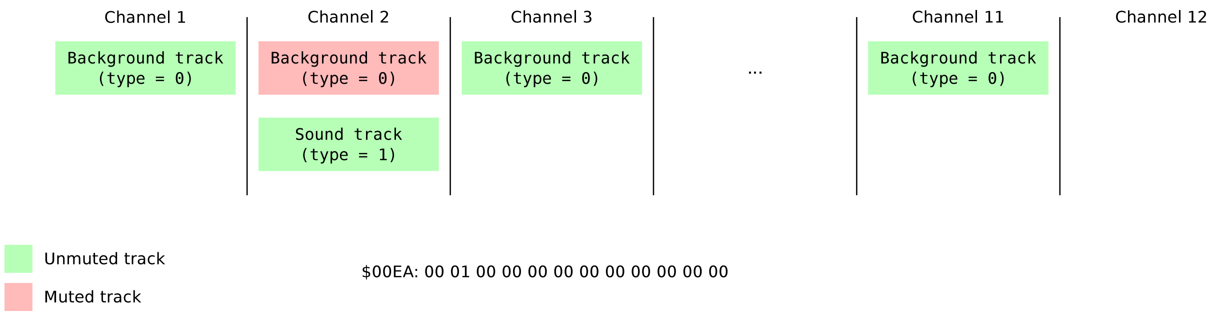
Channel 1
Background track (type = 0)
Channel 2
Background track (type = 0)
Sound track (type = 1)
Channel 3
Background track (type = 0)
...
Channel 11
Background track (type = 0)
Channel 12
Unmuted track
Muted track
$00EA: 00 01 00 00 00 00 00 00 00 00 00 00

**Fig. 29.** Track types

As shown on Figure 29, channels can accept at most one track of each type. When two tracks are sharing the same channel, the foreground one has priority and so the backgound one is muted. The current type of the track played on each channel is reported on an array located at 0xEA. Of course, to avoid desynchronization between background tracks, pattern bytecode is still executed but before actually applying operation to the DSP, the virtual machine compares track type with the one which has the highest priority on the current channel.

```
uint8_t *channels_types = (void *) 0x00EA;
if (track.type == channels_types[current_channel])
    dsp_ops[current_channel]();
```

Indicated by the 0x0F bytecode instruction, the foreground track stop playing by removing itself from the track list and unmutes the background track by clearing the corresponding entry on the 0x00EA array.

## 4.4 Conclusion

Although this sound card hardware design is simple, it is not simplistic and its severals tricks used to implement a custom and reliable hardware interfaces between the CPUs is remarkable.

We can be a little bit more suspicious about the BSMT2000's program which can be seem as a draft of a real audio chip. In fact, it is hard to imagine the need of a mask ROMed version of the TMS320C1x for this kind of simple signal mixing.

On the other hand, the sound CPU firmware seems to perfectly use those hardware ressources in order to get a flexible and fine-grained audio sequencer which is crucial for this kind of arcade machine. This can be heard in particular during the start of multiball sequence in which 10 patterns are played simultaneously.

In conclusion, this sound system is an outstanding sample of the *state of art* audio and electronics engineers from the 80s. However, this pinball was released in 1997. The same result could have been achieved by using more reliable and cheaper componants. Of course, we can consider this outdated design as an essential part of pinball culture's folklore.

## References

1. Sega, Starship Troopers Pinball Manual, 1997, http://mirror2.ipdb.org/files/4341/Sega_1997_Starship_Troopers_Manual.pdf
2. Texas Instruments, 74138 Datasheet, http://www.ti.com/lit/ds/symlink/sn74ls138.pdf
3. Dallas Semiconductor, DS1232 Datasheet, http://datasheets.maximintegrated.com/en/ds/DS1232.pdf
4. Motorola, 68B45 Datasheet, http://www.gbgmv.se/dl/doc/md09/MC6809_DataSheet.pdf
5. Texas Instruments, PAL16L8 Datasheet, http://www.ti.com/lit/gpn/pal16r6am
6. Alliance Memory, A29040C Datasheet, www.farnell.com/datasheets/1770385.pdf
7. Texas Instruments, TMS320C15 Datasheet, http://www.ti.com/lit/gpn/tms320c15