MP3-Controller-On-Chip (MP3CoC)

Håkan Kvist, d00hk@efd.lth.se
Linus Walleij B.S.Sc., d00lw@efd.lth.se

December 8, 2004

Abstract

This implementation of the SoC MP3 controller features a microprocessor based on the Free6502 free IP-core, with 1 KB RAM, and I/O ports to control peripherals. The chip also features such as UART:s, Memory Management Unit and a DMA controller.

Contents

1 Introduction **2**
1.1 Related materials 2

2 Architectural overview **2**
2.1 Clock 3
2.2 Serial UART 3
2.3 DAC UART 5
2.4 MP3 UART 5
2.5 MMU 5

3 On the Free6502 processor **6**

4 Memory layout **7**

5 Firmware **8**

6 Firmware upgrade **8**

7 Synthetization **9**

8 Place and Route **10**

9 Fabrication and testing **10**
9.1 Preparations 11
9.2 First impressions 11
9.3 Fixing the problems 11
9.4 Yield 12
9.5 Current dissipation and critical voltage 12

10 Conclusions **13**

1 Introduction

In recent years several lightweight MP3 players has entered the market. Examples include the USA company Apple Computers' *Ipod*, Singapore-based Creative Technology's *Nomad* series, and the Korean company *i-Bead* players (sold in Sweden under the trademark *Jens of Sweden*).

The goal of this project was to create a complete MP3 player, which could read song data from a flash memory and use an LCD-display for showing information. The decoding of the MP3 stream was to be performed by the MP3 ASIC decoder developed during last year's project,[1] Further, a Texas Instruments TLV320AIC23 A/D converter is used for converting the decoded stream into analog signals.

The optional assignments were to implement write support for the flash ROM and a serial RS323-interface for connection to a host device supporting upload of contents to the flash ROM.

Section 2 through 8 describes the design choices made, the implementation in VHDL and the obstacles encountered during design, synthesis, simulation and layout.

Section 10 draw a few conclusions

1.1 Related materials

In order to evaluate and understand this project, the reader will need the project VHDL files, assembler files, C programs, makefile scripts, synthesis and layout setup for the project. These comprise a quite large UNIX .tar-file, and is available from the design team on request.

A large DEF-file containing the final chip layout suitable for manufacturing should be supplied to the reader along with this report.

2 Architectural overview

This design approach features a controller with a microprocessor-based solution, thus supporting future upgradable firmware.[2]

The processor was chosen as an IP-block[3] which was obtained from the FreeIP-project.[4] The processor Free6502 is based on the MOS 6502 processor from MOS Technologies.[5] One of the project members have significant experience in using and programming this type of processor, so this was a natural choice. The OpenCores IP library was also consulted for possible IP components, but lack of experience with their standard system bus (Wishbone) and the assembler of their processor made them an unattractive choice.

Figure 1 depicts the design, where most components are attached to the processor by way of the memory management unit (MMU). The processor controls the different components by way of memory mapped hardware ports, i.e.: when memory addresses appear on the address bus alongside read/write memory strobe signals will retrieve or write values to wires that are connected to the corresponding units.

In this way, the processor holds absolute control over all parts of the controller. When it comes to streaming MP3 data from the flash memory to the MP3 decoder chip, there also

[1] `http://www.es.lth.se/home/tlt/dicp/2002`

[2] Firmware is a name commonly given to software embedded in an embedded system.

[3] IP-block is "Intellectual Property" block, in practice an VHDL or Verilog model of a digital component.

[4] `http://www.free-ip.com/`

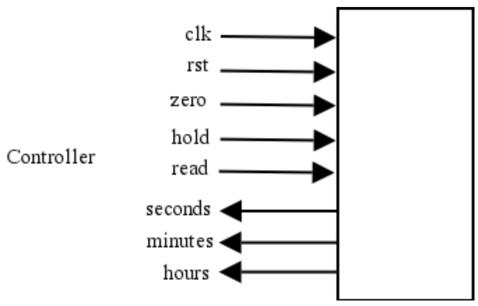
[5] MOS Technologies were for a period of time also known as CSG—Commodore Semiconductor Group. The 6502 processor was used in Apple II, Atari 2600, Commodore VIC-20, the 8-bit Nintendo video game and in a modified variant (with on-chip hardware ports) called 6510 in Commodore 64. Improved 16-bit derivatives were used in Super Nintendo among other embedded systems.

exists a DMA-like feature which will retrieve bytes from the flash memory and send them directly to the MP3 UART that serialise them and forward them, one bit at the time, to the MP3 decoder chip.

If we juxtapose this solution to a simple state-machine solution we notice that whereas a simple state-machine is very (power-) efficient and fast, its configurabilty and upgradability after manufacturing is actually nil. Utilising a microprocessor with a modifiable firmware program makes it possible to add new functionality post facto. It will also be possible to circumvent problems discovered at manufacturing time to a certain extent. Moreover it allows the user to customise and enhance the firmware herself, if desired.

The following sections elaborate on different blocks of the design.

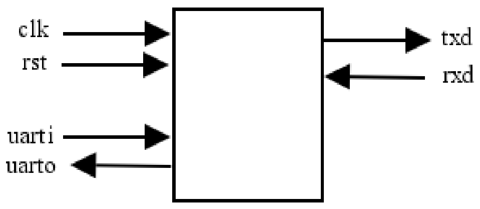
2.1 Clock



The clock keeps track of how long the songs have been played by way of an independent hardware unit that communicate with the processor using memory ports. It counts cycles independently of any processor activity or similar, and have three 8-bit registers that count seconds, minutes and hours in BCD (binary coded decimal) form.

The registers are read by firmware machine-level instructions after requesting a flush of current register contents. This is done to avoid read problems arising from the fact that e.g. the seconds switch from 59 to 00 after reading the seconds register, but before reading the minutes register, so that for example the time 8:00 could be read out as 8:59 because the registers were updated from 7:59 to 8:00 between the read-out of minutes and seconds.

2.2 Serial UART



The serial UART (universal asynchronous receiver transmitter) is also an independent hardware unit. It has two parallel functional units: a ″send″ and a ″recieve″ unit. The testbench tests the UART by connecting the TXD line of the sender unit to the RXD line of the reciever unit, and sending bytes using the sender, while at the same time asynchronously recieveing bytes on the reciever unit. (Of course the baud rate must also be in correspondence for this to work.)

The ″send″ unit of the UART is connected to the LCD at 9600 bps and the ″recieve″ unit is connected to the PC host at 115.2 kbps.

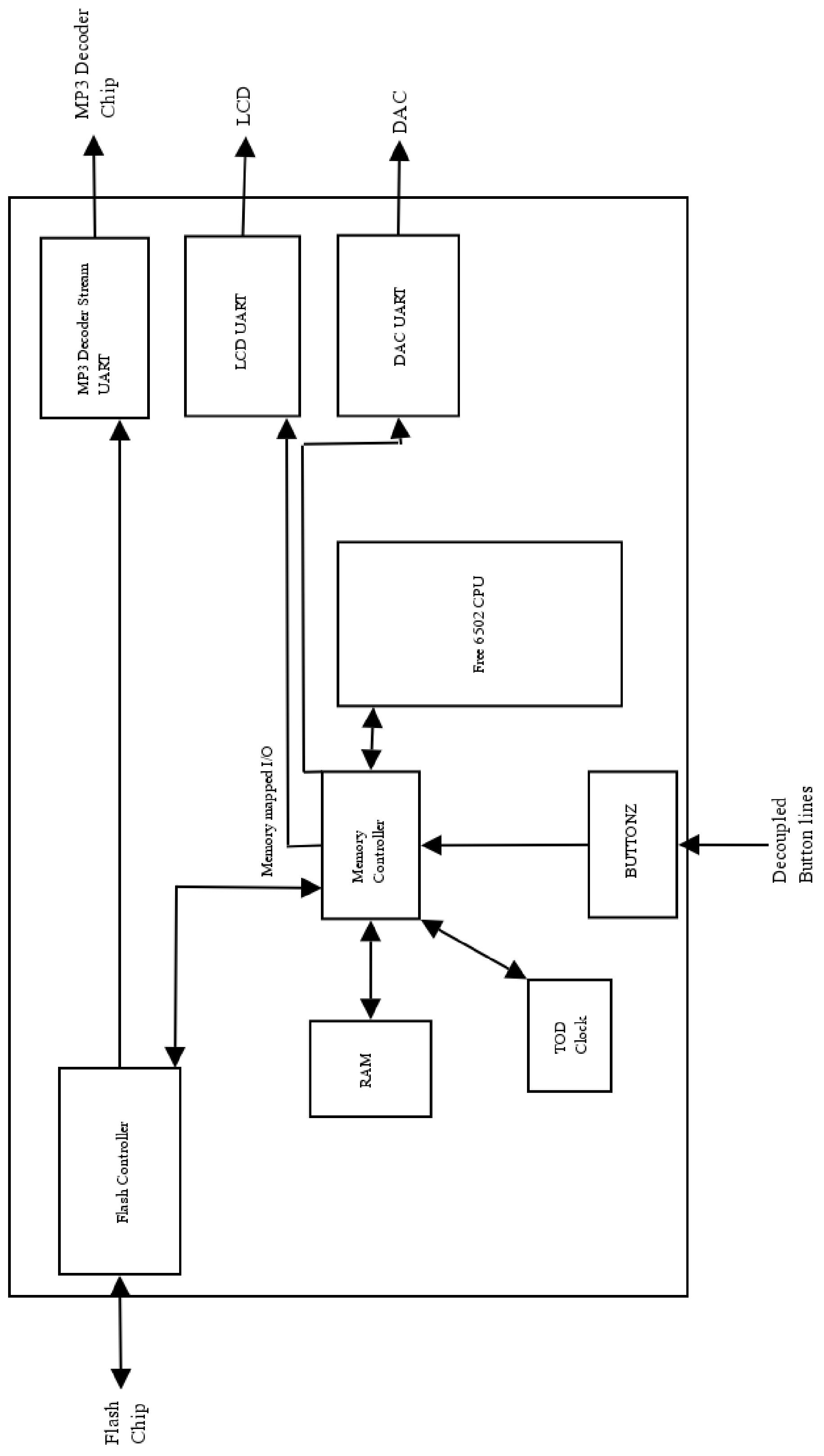
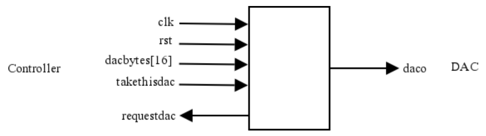


Figure 1: Block diagram of the MP3CoC design.

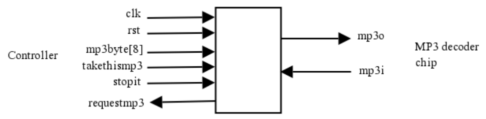
2.3 DAC UART



The DAC UART is another simple memory-mapped I/O component that serialise data and send it to the DAC. It is currently only used for initialisation once during startup, but the dynamic firmware structure makes it possible to add firmware that lets the user customise the DAC settings. (See below on firmware.)

The DAC actually has a 7-bit address and 9-bit data bus combined into a 16 bit word, but during implementation it was see as a single 16-bit word for the sake of simplicity. The firmware is supposed to re-implement and understand the true nature of the 7+9 data format in the 16 bits transfered to the DAC by way of the DAC UART.

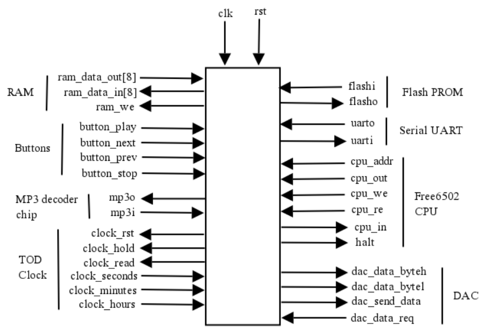
2.4 MP3 UART



This component request bytes from the flash memory (by way of the memory management unit) and serialise them in a format suiting the MP3 chip that was to be used in this project. It is not controlled directly by the processor–rather the memory management unit controls DMA-like registers that will independently stream data to the MP3 UART.

The MP3 UART is actually a subcomponent of the MMU (see below) and is thus not visible directly inside the topmost controller component.

2.5 MMU



The MMU (memory management unit) has a central role and is, apart from the Free6502 processor core, the most advanced component in our construction.

This unit maps request for addresseses both in Flash ROM, RAM and the memory port area. It handles CPU stall cycles introduced due to the slowness of the Flash, RAM synchronisation, and mapping of hardware ports into processor address space.

Further, the MMU contains a small DMA controller that will stream data from the Flash ROM to the MP3 chip, introducing stall cycles to the CPU if both try to access the Flash at the same time. The UART serialisation to the MP3 chip is carried out by a subcomponent called MP3UART (see above).

The MMU also controls the reset of the MP3 chip when ″stop″ or ″pause″ states are active. Such requests will generate a reset signal to the MP3 chip in order to silence any PCM remnants in the ring buffers.

It also presents registers to control this DMA transfer to the processor, and registers which make it possible to write bytes to the Flash PROM on user request. However, they user must first erase the chip with a special erase command. When this command has commenced, bytes may be written to the PROM.

The erase command will erase the entire PROM, so it is necessary to download both firmware and all MP3 songs after erasing the memory.

During Flash memory writing, the processor will of course be stalled from reading the Flash memory so that two accesses does not collide. However it is assumed that when writing to the Flash, no MP3 playing is taking place at the same time. As firmware upload can only commence at system startup, this should not be a problem.

This write functionality was however removed during final place and route. Also see the subsection on memory layout below.

3 On the Free6502 processor

Free6502 is (like its parent) an 8-bit CPU. It features only arithmetic: operations like multiplication and division must be programmatically constructed. This has the downside of complicating program development (though stock subroutines for e.g. floating point may be easily employed) but on the other hand, the strictly arithmetic nature of the unit ensures a very simple and compact piece of VHDL code, which will synthesize to form a small physical footprint.

The CPU has one general-purpose 8-bit register (the accumulator), and four special-purpose registers: two index registers named X and Y (both 8-bit), an 8-bit stack pointer (producing a stack of 256 bytes) and a 16-bit program counter. There is also a state register for things like carry, overflow, zero result e.t.c., no more than 8 additional bits.

The Free6502 processor was easy to use and add to the project. It consists of two VHDL files named *free6502.vhd* and *microcode.vhd*. The latter file, as the name suggests, contains the microcode that make up the actual instruction set of the processor. The processor compiled fine immediately after adding these two files to the project VHDL repository.

The 6502 will assert memory addresses on the address bus and expects the contents of the memory address to be available (or written to) that memory address in one clock cycle. In difference from the quite common physical 6502 component, the Free6502 features separate input- and output- data buses. Commonplace microprocessors use bidirectional buses to decrease pin count, but inside a SoC this is not an issue.

At power-on, a RESET signal is asserted to the Free6502 component. The component will then assert the memory addresses $FFFC[6] and $FFFD for reading, yielding a 16-bit memory address from these two bytes. The program counter (PC) will then be set to this address, where a bootstrap startup sequence begins.

Certain difficulties arise when devising the memory management unit, but this will be treated in more depth in the following subsection.

[6]The addresses in this paper are given in hexadecimal notation.

4 Memory layout

The first construction used 32 KB on-chip SRAM memory and a 4KB PROM for the processor. This enabled the processor to fetch data from the SRAM memory at each rising clock cycle. The interface to the Flash memory was asynchronous, so the processor would request a location in Flash through an I/O memory port and then wait for it to appear on another port. The memory layout was this:

$0000-$00FF	Zeropage
$0100-$01FF	Stack
$0200-$7FFF	Work RAM
$8000-$80FF	I/O Ports
$F000-$FFFF	PROM

This design was doomed when it was discovered that the on-chip SRAM memories available to us were limited to 1 KB (i.e. $0400 bytes).

The memory design was then refactored by connecting the Flash RAM to the system address bus, and allowing the first 64 KB of the Flash to be utilised as processor ROM. This also simplified the handling of firmware upgrades (which were previously planned to be accomplished by uploading chunks of memory from flash to the on-chip SRAM, and starting the program there) as everything could be read on-the-fly from the Flash ROM.

The new memory layout was the following:

$0000-$00FF	Zeropage
$0100-$01FF	Processor stack
$0200-$03FF	Work RAM
$0400-$7FFF	PROM
$8000-$83FF	I/O Ports
$8400-$FFFF	PROM

There was however a problem: accessing the Flash RAM was slower than one clock cycle (at which time the CPU expected requests on the address bus to be valid on it's data in lines).

This problem was solved by lining the CPU clock line through the MMU unit, so that the MMU would force the CPU clock line low when it was busy accessing requested data. This would inhibit the CPU when accesses were going on, and was much better than simply reducing the CPU frequency: the CPU speed would only be reduced when absolutely necessary.

It was however indicated that this clock-inhibiting approach could have a negative impact on the routing of the clock nets of the finished circuit, so as a further design iteration step, a true "hold" signal was added to the Free6502 component during synthesis work. This worked out fine.

The first 64 KB of the Flash memory were used, but the SRAM at $0000-$03FF and the I/O ports at $8000-$83FF disabled 2 KB of that memory. The SRAM and the ports "overlay" the RAM memory, so that the CPU can't "see it". Of course the C programs supplied as input to the project and intended to generate the contents of the flash memory also had to be rewritten accordingly.

Further, the RAM and ROM memory type expected from the CPU was to be asynchronous, i.e. putting an address value on it's address pins would immediately read (or write, if the WE pin was high) that memory address, and not wait for a clock pulse to pass. The example ROM that came with the model was coded in this manner:

```
architecture FOO of rom is
begin
  process (addr)
  begin
    case addr(11 downto 0) is
      when "000000000000" =>  data <= "10100010";  -- 0 = A2
      when "000000000001" =>  data <= "11111111";  -- 1 = FF
      when "000000000010" =>  data <= "10011010";  -- 2 = 9A
      when "000000000011" =>  data <= "00100000";  -- 3 = 20
      ....
```

When it came to generating memories for inclusion in the project, only synchronous RAM was available, resulting in two possible scenarios for interfacing to the CPU: generate an asynchronous RAM memory or hold the CPU for one cycle, so that requested addresses will be clocked in to the synchronous RAM. The latter approach was chosen, so that the CPU is inhibited for one clock cycle when reading from RAM, and for several clock cycles when reading from the Flash.

5 Firmware

The firmware for the CPU was written in pure 6502 assembler using the DASM assembler,[7] originally intended for writing programs for the Atari 2600 computer.

Getting this assembler to compile on Solaris was troublesome but it proved possible with some C hacking. After that it would compile programs without any problems.

The *asm* subdirectory of the project contain raw assembler code (*mp3rom.asm*) as used by the processor. Here, one can also find the Free6502 authors idea of a ROM VHDL representation of the same program (mp3rom.vhd). However the design do not use this VHDL-file: instead the object code is linked by a modified version of the TOC building C program *build-toc.c*, which is found in the *c* subdirectory.

The assembler file contains all port definitions and similar things needed to write firmware for the controller, and may be compiled by issuing a "gmake" command in the directory, which will compile the code by way of GNU Make. Subsequently entering the *c* subdirectory and executing "gmake toc" will create the raw Flash image to be programmed to the Flash PROM.

6 Firmware upgrade

If both buttons "play" and "stop" are held down when the controller is powered up (at reset), the controller will enter firmware upload mode and start a receiving program that listens for a 115.2 kbps connection on the PC I/O port.

To handle firmware upgrades in this architecture, the PROM sets up the transfer and then copies a small recieveing program to RAM address $0200 and starts it there. This enables the processor to overwrite $0400-$7FFF and $8400-$FFFF without interfering with it's own code.

[7] `http://www.atari2600.org/dasm`

The entire PROM is then erased and the memory must be fully reprogrammed from the serial bus, including both firmware and all MP3 files.

After this upgrade, the controller program is "rebooted" with a jump to the reset vector.

The memory upload protocol is as follows:

BNF grammar

<transfer>	::=	<chunk>+ <eofmark>
<chunk>	::=	<header> <bytes>
<header>	::=	$aaaaaaaa $llllllll
<eofmark>	::=	$FFFFFFFF $FFFFFFFF

Where $aaaaaaaa is the destination address of the chunk, and $llllllll is the length of the chunk.

The firmware upload functionality also makes it necessary to have separate hardware ports for storing bytes in the Flash. When writing bytes to the memory, the CPU will inhibit during the writing process so that writing may commence.

All this firmware upload and flash write functionality was added to the design, but at the place and route stage it was eventually removed from the topmost component as it introduced project hazards. The logic is in there, but it is not used.

7 Synthetization

Synthetization of the design was performed using the Synopsys tool from Mentor Graphics. At first synthesis try, many parts of the design didn't work. This was due to several facts:

First, a faulty toolkit from the vendor was used. This resulted in flip-flop arrays being replaced by erroneously mapped scan-path flip-flops. This was solved by downgrading to a previous toolkit for an older version of the Synopsys compiler.

Second, we had some apparent glitches in our design. This was solved in most cases by moving the signal-driving logic to a clocked process. In some cases asynchronous signals had to be used (as for the "halt" signal to the CPU) but this was apparently OK.

Third, the generated RAM memory would not work. During post-synthesis simulation, it appeared as if the RAM memory did not even exist: no signals would come out of it. This problem was a lot trickier to solve.

At first consultation with development engineers the team was informed that the RAM memory could not be added to the interior of the controller so the signals for it had to be moved to the top of the controller component, so that the post-synthesis simulation would be able to use the generated memory. Acting on this advice, the connections to the RAM memory were moved to the top component, and the generated simulation model for the RAM was then added to the testbench.

This line of thinking is not unreasonable, but was in fact incorrect in this case: RAM:s may in fact be instantiated at any point in the design. The problem was of a considerably more subtle nature: a library part of the RAM had somehow been created in the "WORK" library, taking precedence over the generated post-simulation RAM library. By removing (rm -rf work), recreating (vlib work) and recompiling the "WORK" library ("make clean"), this ghost-component was removed and all things started to work as expected.

From this it was learned the hard way that these tools are never intuitive and may not be trusted. In fact, they do not abstract away any part of the underlying interiors of the design tools, so a technical documentation of how Modelsim use the underlying "library folders" would have been handy. However, designers tend to develop an intuitive feeling for "how Modelsim thinks" that is of that typical tacit knowledge type that engineers are so well known for possessing. (Knowledge that "reside in the fingers of the tool operator" and does not appear in the tool manuals.)

The project would then successfully run through an exhaustive post-simulation test suite, testing all buttons and text output in a very long, automated script that would run for about 5 hours on an ordinary Sun workstation.

8 Place and Route

During place and route the design was imported into the Cadence chip design tool called *Silicon Ensemble*. The Verilog description file generated from the synthesis step was used at this final design.

The chip floor area had to be extended from the preset defaults to 2100 $\times$ 2100 square microns in order to fit the RAM memory. The suggested values from the scripts delivered with our SE setup would not fit it. The RAM memory proved complicated to place using a script and had to be placed by hand. The block halo had to be increased to 55 microns in order to avoid power net routing constraint violations.

One of the first problems during place and route was that signals had been added to the top component in order to support writing of flash memory contents. An 8-bit write data bus and three chip control signals had been added. This path was incorrect, since the data bus for the flash memory is bidirectional (tristate) and thus it would have been necessary to introduce logic to switch the control pads for the data bus from *read* to *write*.

While this was indeed possible, it would have introduced delays in the project and increased the risk of a failed design, so the design team opted to disconnect the flash write functionality, so that the topmost component would resemble that of other project teams, and make it pin-compatible with their designs. This makes it possible, among other things, to use the same physical testbench at chip verification.

The logic for flash write and firmware upload was left in the VHDL model so that it may be reactivated: only the topmost signals (on *controller_top*) were removed.

One small additional problem occurred when performing place and route: the command **ctgentool** complained about a piece of code that inverted the *clk* signal and passed it immediately to the SPI *sclk* signal.

The lesson learned is that one can *never* actually *use* the *clk* signal in constructions; one should only use it for triggering logic functions with **rising_edge(clk)** constructs. This way it proved wise to implement a true *halt* function in the IP-component: had this not been done, the clocknet routing would probably have ended up all wrong.

The final chip design has the look of figure 2.

9 Fabrication and testing

After finalizing design, our chip was manufactured and we ran a test of functionality. A (physical) testbench was built at the ASIC department and used for all chips in this project run. The testbench had the LCD, DAC and MP3 decoder chip in place and interfaced to the controller using a socket which had pins connected according to the pin layout of the project files. The push-buttons on the testbench were decoupled to avoid jitter.

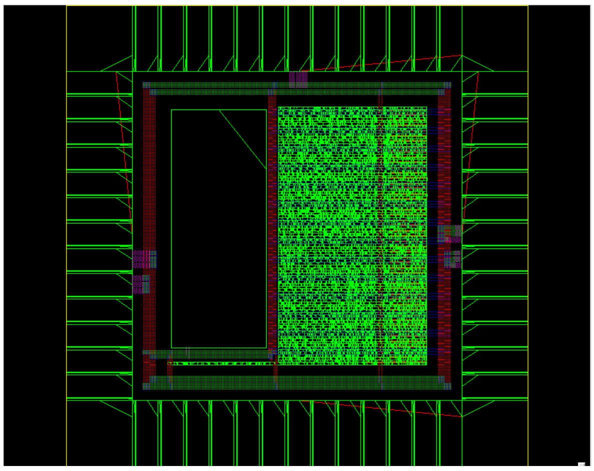


Figure 2: The final controller chip layout in Silicon Ensemble.

9.1 Preparations

Before verifying we had a EEPROM with suitable firmware for our controller CPU. This was the same firmware that had previously been tested in the testbench for the design.

9.2 First impressions

When we connected the circuit to the testbench and sent a RESET signal the processor started running and wrote the correct welcome message to the LCD. From this we concluded that both CPU and MMU was working as intended. The processor continued by reading key presses, and we could advance back and forth among the songs in the EEPROM using the ”backward” and ”forward” buttons.

When we started playing a piece of music using the ”play” button the LCD correctly stated that we had entered playmode, and the TOD-clock started ticking up seconds and print them to the display. However the clock continued ticking and no music came out of the DAC. We could still press ”backward” and ”forward” buttons to change song, but the machinery remained silent and the clock would never stop for a single song, it would run on forever, way past the actual length of the song.

9.3 Fixing the problems

All issues turned out to be firmware-related or caused by other chips and could be fixed, so the controller was essentially 100% working.

We suspected that the set-up sequence for the DAC was incorrect, and that this was the reason to why no sound came out of the circuit. By consulting data sheets and altering the sequence in the firmware, this problem could be fixed: we had neclected the fact that the DAC had to be set up to accept a 12 MHz clock and that the digital interface had to be specified to I^2S. It also turned out that the RESET register of the DAC was dangerous to write, presumably because we wrote it last and the reset would restore the default configuration. For this reason we avoided using DAC RESET altogether.

After this sound came out of the attached speaker, but only noises. By analyzing the memory with a logic analyzer, we could see that the DMA-part of the MMU begun reading MP3 data from the memory correctly and sent it to the decoder chip, so logically speaking everything was working until the MP3 decoder chip suddenly stopped requesting bytes.

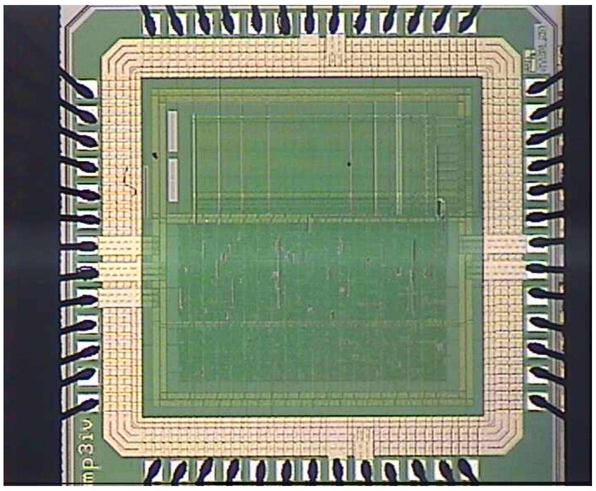


Figure 3: The final physical chip as seen in a microscope.

We suspected that this second problem was cause by bad MP3 files, or rather that the MP3 decoder was quite sensitive in regards to what it would accept. It it was to read a number of bytes and then “crash”, this phenomenon would appear: the circuit would lock up and put the processor on hold so that it seemed like the song would run on forever as it was waiting for the decoder to request new bytes for all eternity.

By replacing our own MP3 files with the test files distributed along with the project, everything suddenly started working. We could not determine for sure what properties of the MP3 files that crashed the decoder, but we had indications that parameters like sample rate, bitrate and number of channels (mono prefered) were sensitive things.

A remaining error was that the last letter in the LCD was not erased, so after switching from “pause” to “stop” or “play”, the display would display “stope” or “playe”. This could easily be fixed by adding a space to the strings in the firmware.

9.4 Yield

We recieved five circuits manufactured with serial number 67000 01/B (there were two designs in each circuit, /A and /B, but in each circuit only one of them was bonded). The Electroscience customer number A37370 was also written on the capsules. Of these five, four proved to work and one was broken. On this circuit we removed the lid and studied the circuit in a microscope, see figure 3.

9.5 Current dissipation and critical voltage

During normal operations at 3V DC the circuit would consume some 1.5 mA current. When we reduced the voltage the current dissipation decreased until we reached a consumption of 0.5 mA at 1.76V. Here the circuit stopped working entirely. It is thus possible to lower the current dissipation somewhat by lowering the voltage.

However it came to our knowledge that other circuits manufactured would only dissipate around 0.5 mA, so what we won in flexibility and configurability, we loose in power dissipation.

10 Conclusions

The construction principle of using a CPU IP-core and construct peripheral components around an MMU proved feasible, though complex. However the final design exhibits a future flexibility that would not have been possible using state machine constructs only. Using the off-the-shelf Free6502 IP-core was simple and the only modification done was to introduce a “halt” signal to the CPU.

Creating firmware for the CPU proved to be very simple. Any experienced assembler programmer can easily set up the development tools. One very good thing about using upgradable firmware is that while the chip has already been sent to production, firmware can still be developed and the final user interface design is not fixed. Adding e.g. graphic equalizer capabilities and volume control to the firmware is still possible.

The PC connection bus was implemented for firmware uploads, but since the memory write capability was removed, the original purpose is no longer feasible. Yet, the port may be utilised by firmware to carry out e.g. remote control or file streaming.

Interfacing the memory-mapped I/O, Flash and RAM memory to the CPU proved very complicated. The lesson learned is that MMU:s are complex creatures.

Using a CPU consumed a lot of silicon space. The on-chip RAM memory for the system stack, zeropage and user RAM also consumed a lot of space. However: using an arithmetic-only CPU simultaneously saved space on the chip as complicated operations such as multiplication could be implemented by way of arithmetic software.

The success with manufacturing of this chip give at hand that the design decisions made were correct, but that running a CPU off flash memory will cost something in terms of power dissipation.