Coding Demos For

The Atari 2600 Video Computer System

An Introduction To Some Techniques And Suggestions On Solving Commom Problems

by Sven Oliver ('SvOlli') Moll

Revision 3.0 - 2013-03-30 - 17:00

Personal background

Got an Atari 2600 in 1982 for X-mas

Sold it two years later to buy a Commodore C=64

Learned 6502 / 6510 assembler there

Bought a 2600 again in the early 90's for nostalgic reasons

Ported 2600 emulator Stella to Sega Dreamcast in 2002 - 2004

Today most coding time is spent on Qt projects

Motivation for this talk

Addendum to the "Ultimate Atari 2600 Talk", but this talk will work without knowing the other

The most retro hardware you can get without spending big money

Even though it is ancient, there's still much to discover

Since Revision 2.0 only 10 demos were released

Still the most f***ed up hardware I've encountered so far.

Thanks

Thanks to the following sites for providing me with information, supporting me and / or letting me use their content for this talk

http://www.atariage.com/
http://www.biglist.com/lists/stella/
http://www.qotile.net/minidig/
http://www.randomterrain.com/
http://www.console-corner.de/
http://en.wikipedia.org/wiki/Atari_2600

Special big thanks to the folks at AtariAge

Part 1:

What You Need To Start

What You Need To Start

There's no need for hardware, most time of the development runs inside an emulator

All necessary code is available as source

–> platform independent

So far I've met Windows, MacOS and Linux users

Development "Back Then"

Programming in 1977:

Code assembled on a computer running a proprietary OS

Connected to a special cartridge

When the software crashed, stripes top down would be displayed

For debugging a logic analyzer was used, which could display steps leading to a special condition

What You Need: Software (1)

First of all you'll need an emulator

Probably Stella

Most mature, best maintained

Integrated debugger

Several other features for analyzing the behavior of the hardware (CPU, RIOT, TIA)

Emulation is very good, but not perfect

–> what works in Stella might now work on the 2600

What You Need: Software (2)

For coding you'll need an assembler

Basically there are two choices

– dasm is defacto standard of the 2600 scene

– as65 of cc65

Both have advantages and disadvantages

If you don't have reason to pick as65 specifically, I suggest to go for dasm

For rapid development, Batari Basic might be worth a look

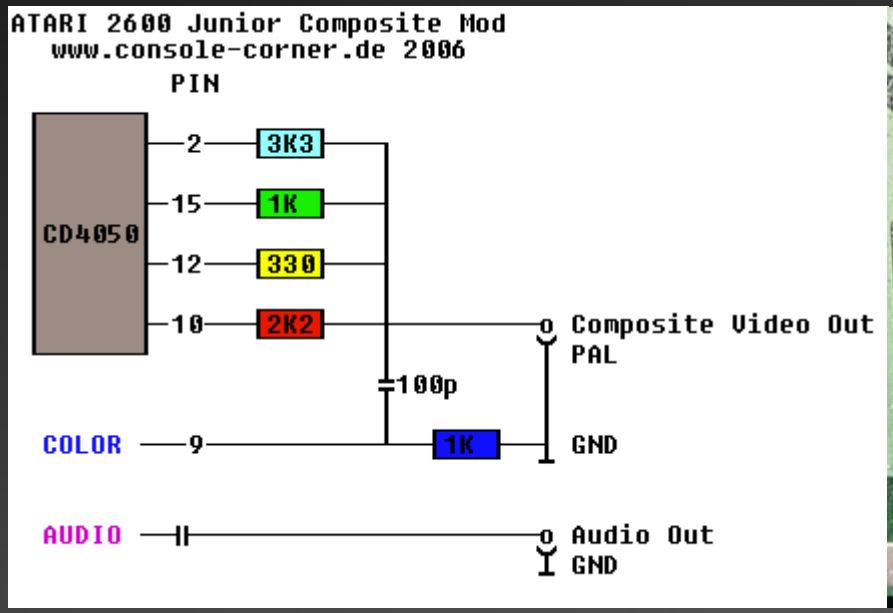
What You Need: Hardware (1)

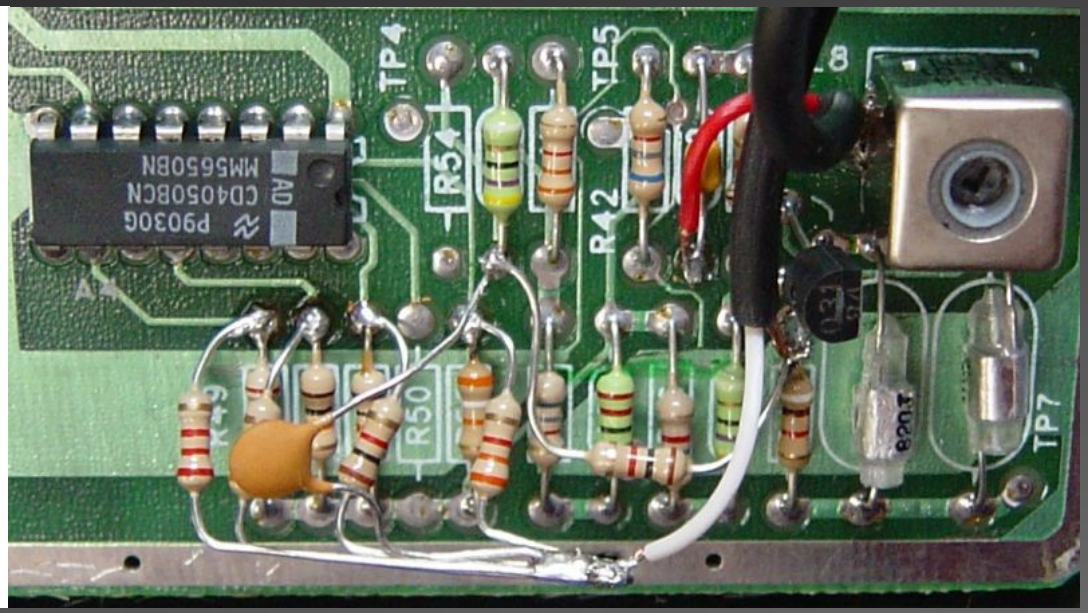
A generic console

For PAL development I suggest looking for a 2600 Jr

It's the easiest to modify with a composite output

http://www.console-corner.de/videomod.html





Images courtesy of www.console-corner.de, used with permission

What You Need: Hardware (2)

Atari 2600 Jr (1984)

Image courtesy of Ewan-Alan, Wikipedia, public domain

What You Need: Hardware (3)

A module you can upload your code to

The defacto-standard is the Harmony Cartridge

Capable of loading ROM data from USB or SD card

What You Need: Hardware (4)

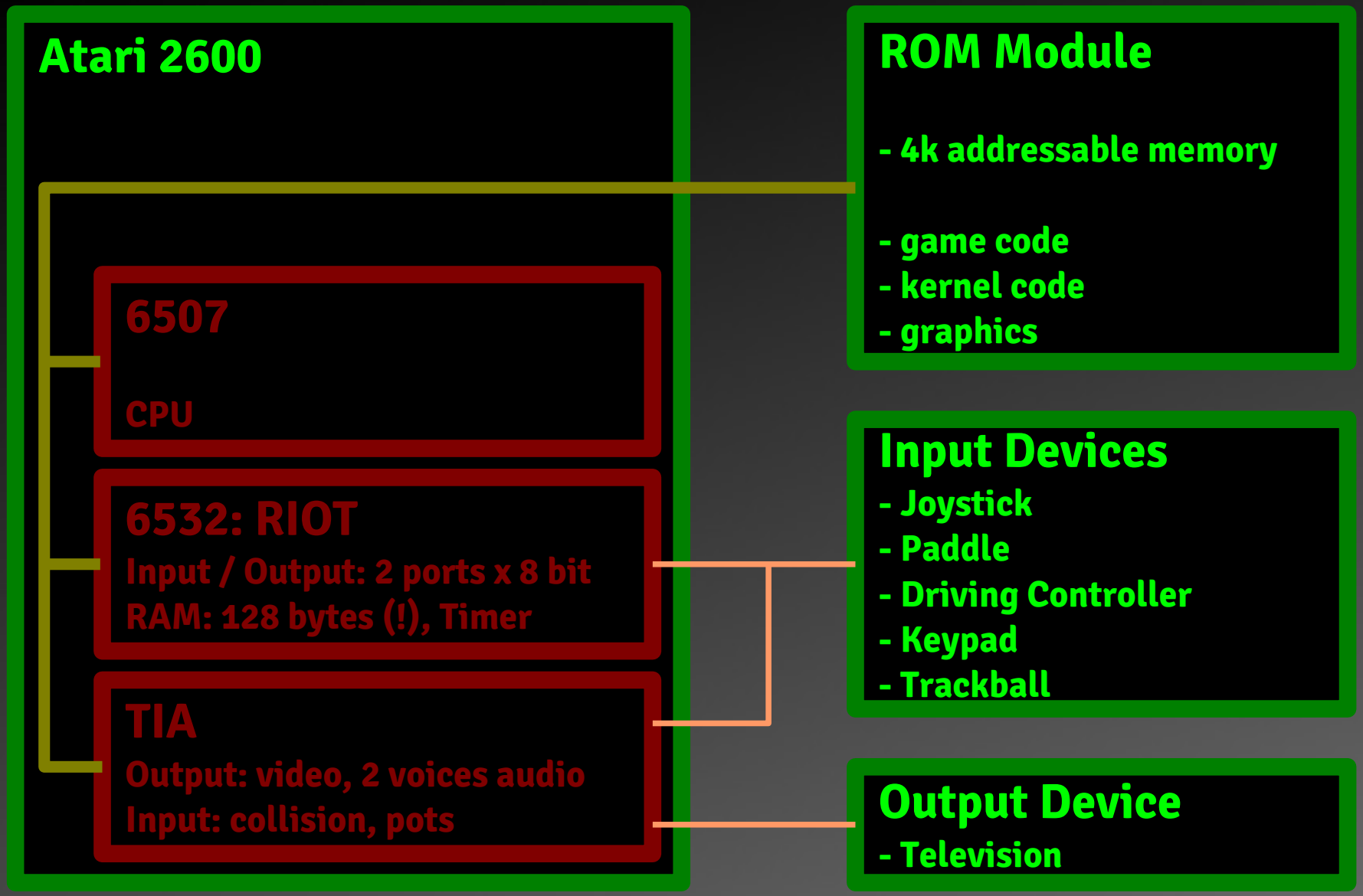
Harmony Cartridge

Image courtesy of Fred Quimby, used with permission

Part 2:

A Look At The Inside

Hardware block diagram



6507: the CPU (1)

The 6507 is a stripped down version of the 6502

Described in depth by Michael Steil on 27c3

Designed by Chuck Peddle, who also worked on the Motorola 6800 team

Clocked at ~1.19MHz

6507: the CPU (3)

Let's compare the 6507 to the 6502:

Smaller chip package (28 pins instead of 40 pins)

What's missing?

3 address lines (64k internal, but only 8k external)

Both interrupt lines are hardwired to +5V internally

1 clock line (phi1), 1 VSS, Sync, S0

3 "n.c." pins ;-)

Even cheaper, popular for embedded applications

6532: RAM, I/O and Timer

Very common companion chip to the 6502 family

128 bytes of RAM

2 I/O ports (8 bit)

- 1 I/O port used for the 5 console switches
- 1 I/O port used for both joysticks
  (only directions, read-write)

Timer that is optionally capable of sending interrupts

(6507 is not capable of receiving interrupts, though)

Memory map (1): overview

External address space of 6507 is 8k

Mirrored 8 times in 64k internal address space

Starting at:

$0000, $2000, $4000, $6000, $8000, $A000, $C000, $E000

$0000 - $0FFF IO, timer and RAM

$1000 - $1FFF ROM (module)

Typically used in two ways:

$0000 - $1FFF

$0000 - $0FFF and $F000 - $FFFF

Memory map (2): ROM

Cartridge port has 24 connectors

Resembling 24 pins of an 32k bit ROM / EPROM

Power: 3 lines: 1x +5V VCC, 2x GND

D0-D7: 8 data lines

A0-A12: 13 address lines

What's missing?

- Chip select: per definition CS is high active A12
- Read / Write: only defined as ROM port (design fail)

Memory map (3): RIOT (1)

Exact mapping: xxx0 xxMx 1NNN NNNN
M: mode (0: RAM 1: I/O+Timer)

RAM: usually accessed at $0080 - $00FF

IO and TIMER: usually accessed at $0280 - $029F

Available 8 times in 8k space, alternating

RAM: $0080, $0180, $0480, $0580, ..., $0C80, $0D80
IO: $0280, $0380, $0680, $0780, ..., $0E80, $0F80

Memory map (4): RIOT (2)

IO-Ports: $0280 (DRA), $0281 (DDRA)
Switches: $0282 (DRB), $0283 (DDRB)

Timer status registers: $0284 - $028C
$0284: read timer (disable interrupt), $028C (enable int.)
$0285: read interrupt flag register (bit 7: timer interrupt)

Interrupt disabled:
$0294 write timer div by 1
$0295 write timer div by 8
$0296 write timer div by 64
$0297 write timer div by 1024

Interrupt enabled:
$029C write timer div by 1
$029D write timer div by 8
$029E write timer div by 64
$029F write timer div by 1024

Memory map (5): RIOT (3)

RAM: 128 bytes

Needed at two locations

– $0080 - $00FF: "variables"
– $0180 - $01FF: stack

Keep in mind that the stack uses a mirror

Quote from development manual:

"The microprocessor stack is normally located from FF on down, and variables are normally located from 80 on up (**hoping the two never meet**)."

Memory map (6): TIA (1)

Exact mapping: xxx0 xxxx 0xNN NNNN

Usually accessed at $0000 - $003F

Available at 32 different positions inside 8k area:

- $0000, $0040, $0100, $0140, ..., $0F00, $0F40

"Space" for 64 registers

14 "read only" registers

- Mirrored 4 times inside the 64 bytes address space

45 "write only" registers

Memory map (7): TIA (2)

Read registers of the TIA:

CXM0P	CXM1P	CXP0FB	CXP1FB	CXM0FB	CXM1FB
CXBLPF	CXPPMM	INPT0	INPT1	INPT2	INPT3
INPT4	INPT5				

Collision Input

8 registers for collision detection, 6 for input

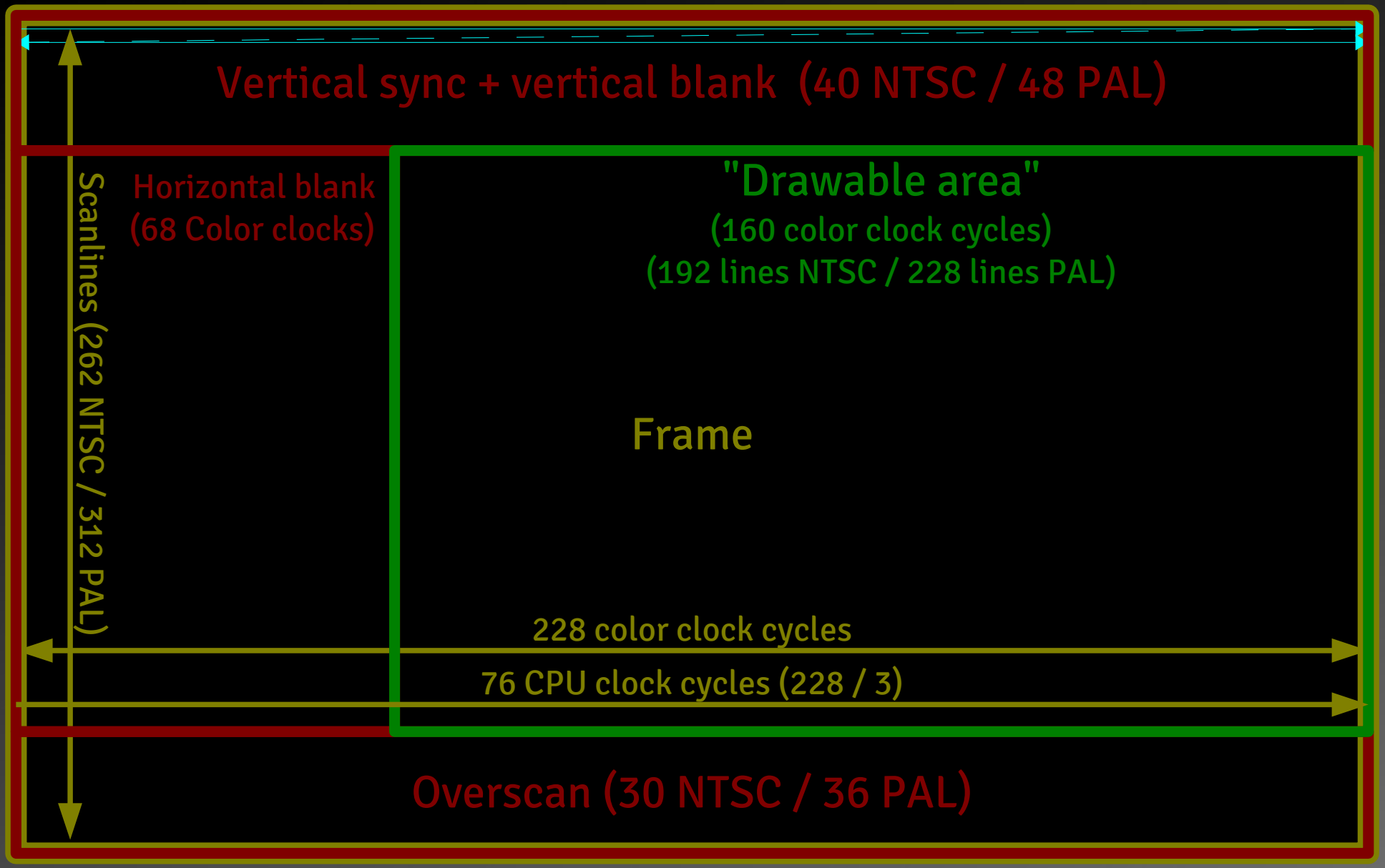
Memory map (8): TIA (3)

Write registers of the TIA:

VSYNC	VBLANK	WSYNC	RSYNC	NUSIZ0	NUSIZ1
COLUP0	COLUP1	COLUPF	COLUBK	CTRLPF	REFP0
REFP1	PF0	PF1	PF2	RESP0	RESP1
RESM0	RESM1	RESBL	AUDC0	AUDC1	AUDF0
AUDF1	AUDV0	AUDV1	GRP0	GRP1	ENAM0
ENAM1	ENABL	HMP0	HMP1	HMM0	HMM1
HMBL	VDELP0	VDELP1	VDELBL	RESMP0	RESMP1
HMOVE	HMCLR	CXCLR		Sync	Graphics

4 registers for syncing, 34 for graphics display

Display


Vertical sync + vertical blank (40 NTSC / 48 PAL)
Horizontal blank
(68 Color clocks)
"Drawable area"
(160 color clock cycles)
(192 lines NTSC / 228 lines PAL)
Scanlines (262 NTSC / 312 PAL)
Frame
228 color clock cycles
76 CPU clock cycles (228 / 3)
Overscan (30 NTSC / 36 PAL)

No Framebuffer

When the Atari 2600 was designed in 1975, RAM was very expensive

To convert the graphics capabilities to a dumb framebuffer you'll need about 30k of 7-bit words

Not only too expensive, but also not addressable by 6507 (8k)

A completely different approach: program the video chip while the image is displayed

Advantage: cheap and very flexible

Disadvantage: CPU is "occupied" during display

"Racing the beam"

Instead of "running" the graphics frame by frame, the image is drawn line by line

If nothing is changed, the next line is drawn like the one before

There are no registers for Y-components

Example: sprite size is 8 bit wide and as high as the screen

You need to tell the TIA what to paint while it is painting! This is called "Racing the beam"

Playfield graphics (1)

Resolution: 40 bits – 4 color clock cycles per bit

Registers responsible for playfield generation:

COLUPF, COLUBK: color

PF0, PF1, PF2: data

How to squeeze this 40 bit resolution into 3 bytes?

CTRLPF: control register

- Bit 0: 1=reflect playfield, 0=repeat playfield
- Bit 1: 1=use player colors, 0=use playfield color
- Bit 2: 1=playfield over sprites, 0=sprites over playfield

Playfield graphics (2)

The data registers in depth:

- PF0: `ABCD ----`
- PF1: `EFGH IJKL`
- PF2: `MNOP QRST`

So the playfield data are only 20 bits that can be

Mirrored: `DCBAEFGHIJKLTSRQPONMMNOPQRSTLKJIHGFEABCD`

Repeated: `DCBAEFGHIJKLTSRQPONMDCBAEFGHIJKLTSRQPONM`

Changed: `DCBAEFGHIJKLTSRQPONMdcbaefghijkltsrqponm`

Note: Intuitive and straight forward to code for, well this isn't

Sprites

The TIA has 5 sprites:

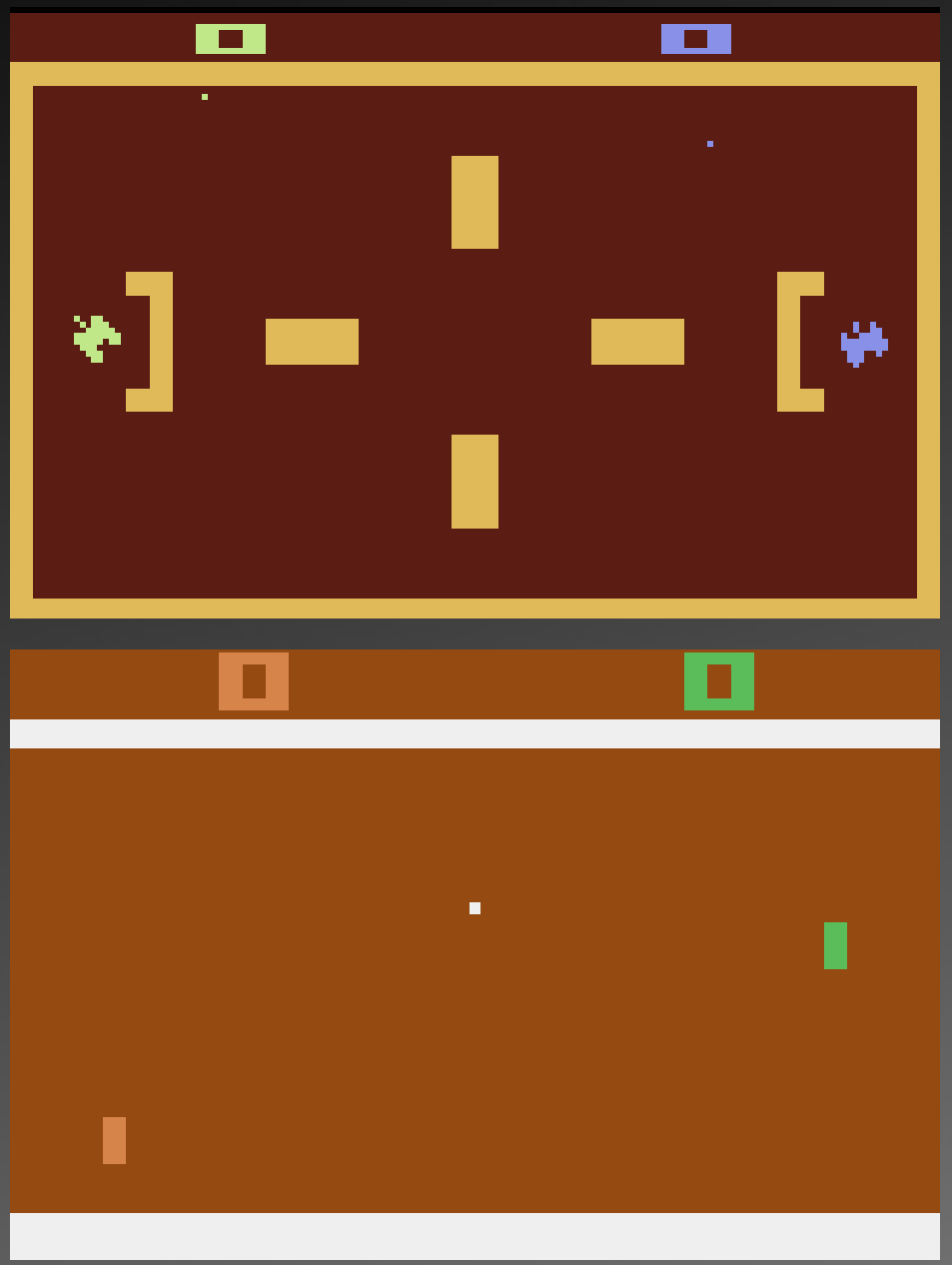
- 2 player sprites (8 bit data)
- 2 missile sprites (1 bit on/off)
- 1 ball sprite (1 bit on/off)

Missile sprite positions can be linked to player positions or positioned independently

Hardware was designed for running

Combat

Pong (Video Olympics)



Sprites placement (1)

How are sprites placed on the screen?

Y: enable before beam reaches position

X: more complicated, though

RESP0, RESP1, RESM0, RESM1, RESBL

Reset the sprite position, no value taken

"Reset" has a slightly different interpretation here:

Not reset to position 0, but to current X position of beam

Sprites placement (2)

TIA clock 3 times as fast as CPU clock

Fine-tuning the position:

HMP0, HMP1, HMM0, HMM1, HMBL:

- 4 bit signed motion register
- can move -8 to +7 color clock cycles
- negative moves right, positive left

HMOVE:

- apply motion register settings

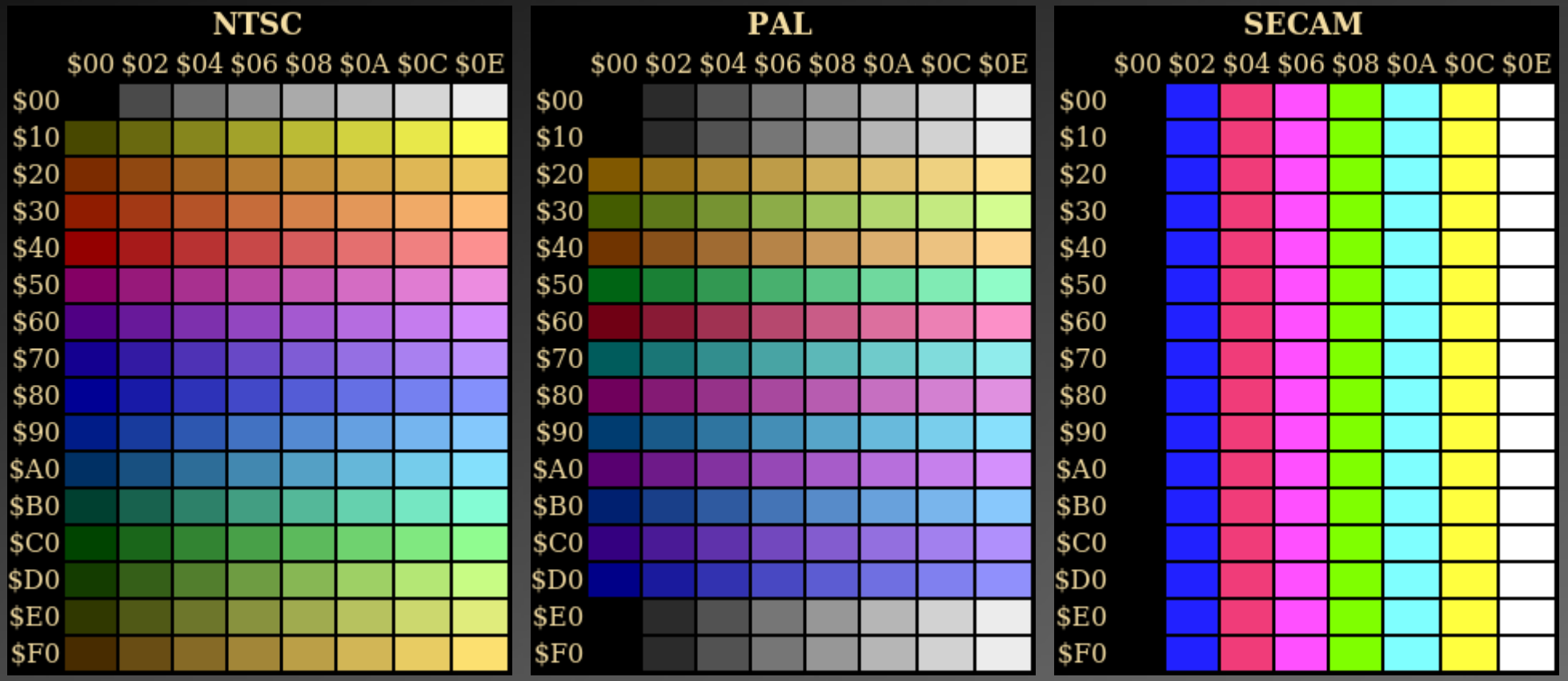
HMCLR:

- clear all HMxx registers at once

Colors (1)

4 Color registers: background, playfield, 2 players

Each color can be picked out of a palette of 128



Colors (2)

COLUBK
– background

COLUPF
– playfield, ball

COLUP0
– player 0, missile 0
– playfield left half (CTRLPF bit 1)

COLUP1
– player 1, missile 1
– playfield right half (CTRLPF bit 1)

NTSC Or PAL?

This question divides into two topics: color and frequency (50 vs 60Hz)

Looking at the colors, NTSC is better

The frequency of 50Hz is imho better than 60Hz because you've got more rastertime to do stuff

I've also seen using the difficulty switches for adjusting color map and frequency

Keeping In Sync (1)

Since the timing of writing to the registers is essential, it is crucial to know where the beam is

To accomplish this, there are three rules:

1) Count the cycles: of every opcode the time it takes to execute is known
2) Use a write to WSYNC to stop the CPU until the start of a new scanline is reached
3) If you can't predict how long some code will take, start the timer and wait for it to timeout after the work is done

Keeping In Sync (2): The Timer

An example of using the timer (close to real life):

```
; straight forward
 lda #(DELAY / 8)
 sta $0295
; [ …doing stuff… ]
@loop:
 lda $0284
 bne @loop
```

```
; with interrupt
 lda #(DELAY / 8)
 sta $029D
; [ …doing stuff… ]
@loop:
 bit $0285
 bmi @loop
```

When timeout already occurred, left code will block

-> even though there's no interrupt line on the CPU, the interrupt of the timer is not useless

Audio (1)

The TIA has 2 voices each having 3 registers

AUDV0, AUDV1: Volume 4 bit

AUDF0, AUDF1: Frequency 5 bit

- Base frequency divided by (AUDFx + 1)

AUDC0, AUDC1: Control 4 bit

- 11 unique settings
- Most of the settings can not be used for music,
  but for sound effects like motor noise, shots, ufos...

Audio (2)

Sound generation can be looked at in two steps:

Step 1: basic signal is generated by setting the audio line high or low: basic output is a rectangle

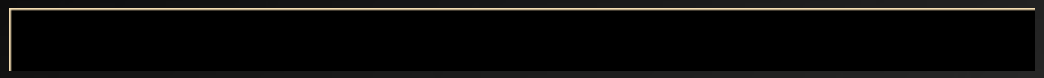
AUDC0, AUDC1 define the bit pattern

Base frequency = color clock / 114
NTSC: 3579575 Hz / 114 = 31399.78 Hz
PAL: 3546894 Hz / 114 = 31113.10 Hz

Sound generated by shifting out by the bit pattern

Possible basic waveforms

AUDCx = 0 & 11



AUDCx = 1

AUDCx = 2

AUDCx = 3

AUDCx = 4 & 5

AUDCx = 6 & 10

AUDCx = 7 & 9

AUDCx = 8

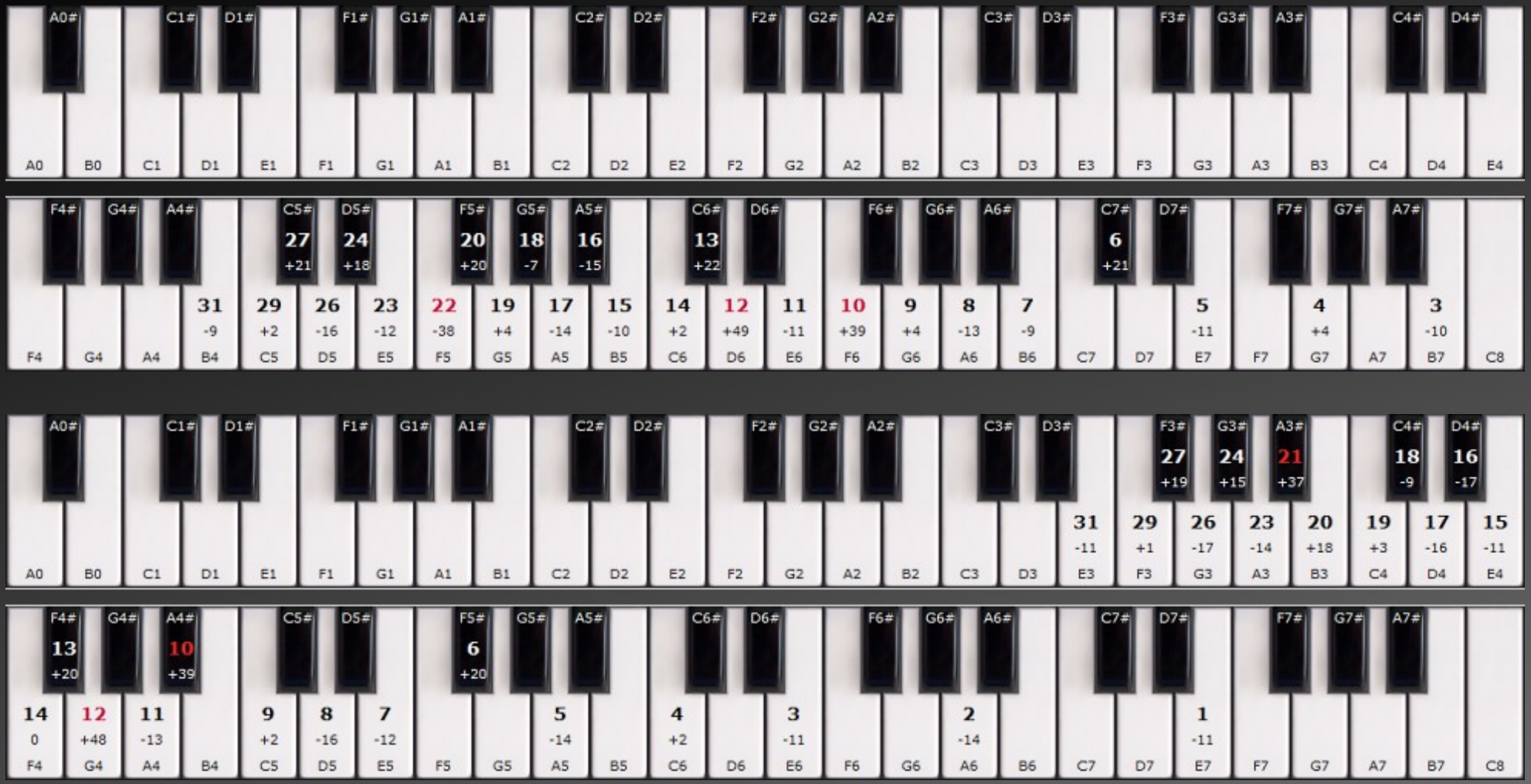
AUDCx = 12 & 13

AUDCx = 14

AUDCx = 15

Audio (3)

AUDCx: keys for settings 4 and 12



Images courtesy of www.randomterrain.com, used by permission

Audio (4)

Step 2: basic signal is multiplied with AUDVx

AUDCx bit pattern "0" is useful: when activated 4 bit digital audio can be played by writing data to the corresponding volume register.

Most impressive example for this kind of sound generation is Berzerk VE (Voice Enhanced), a hack which features the voice of the arcade version!

Audio (5)

For providing your demo with exciting music, I suggest to take a look at Paul Slocum's "Music Kit 2"

It's kind of a tracker playroutine without an editor

Music is set up by creating tables with the assembler

The source code and the format is well documented

Part 3:

How To Get Things Done

The First Program

Use some existing code to start your project

– There's a lot of code that works fine as a template

You need:

– A reset routine

– Some display code (called kernel)

– Probably some logic on what to display next

Asymmetrical Playfield (1)

Writing some kind of framebuffer using the asymmerical playfield is a good start

The simplest form of "racing the beam"

Short recap: playfield data looks like this

Changed: DCBAEFGHIJKLTSRQPONMdcbaefghijkltsrqponm

Using the registers

- PF0: ABCD ----
- PF1: EFGH IJKL
- PF2: MNOP QRST

Sprites

Two are available: player 0 and player 1 (P0, P1)

Sprites are 8 bits = 1 byte wide

As high as you want to

Remember: the 2600 works on a line-by-line basis

Also available are two missile and a ball sprite

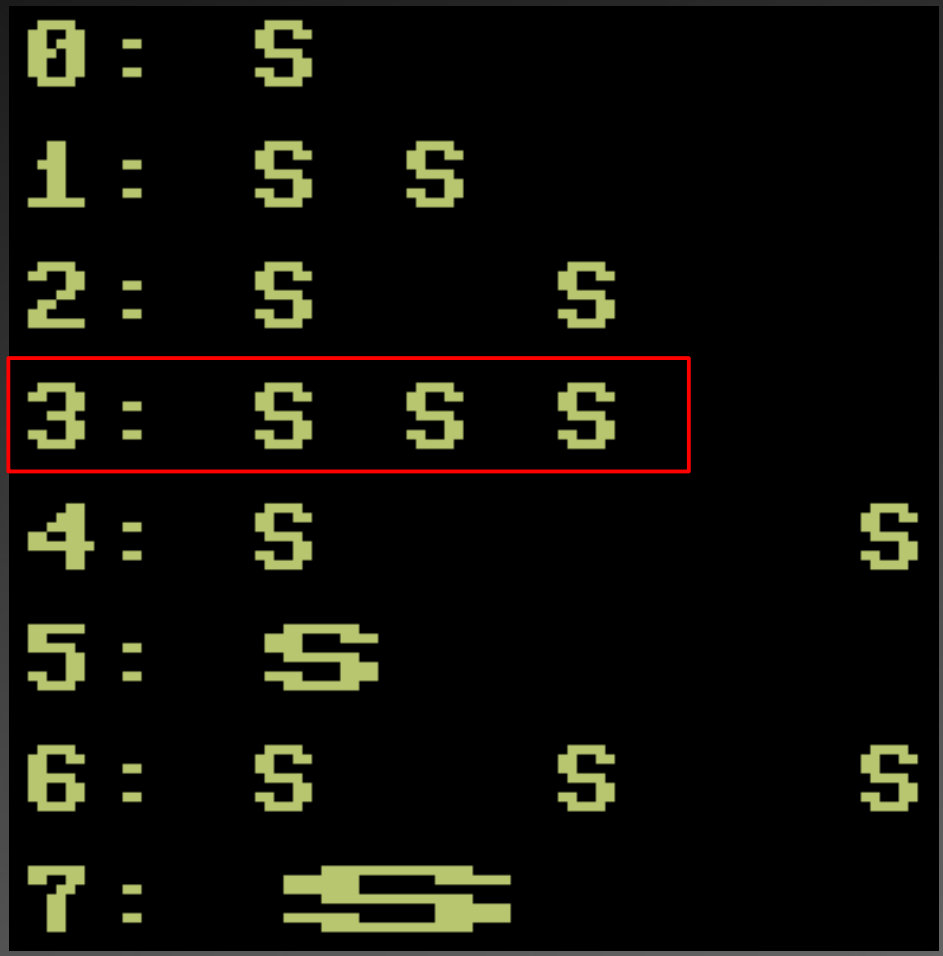
These can be only turned on or off (1 bit)

Sprites: size and repetition

The player sprites can be repeated or stretched in 7 different ways

Mirroring of player sprites is also possible

Ball and missile sprites can be defined being in size of 1, 2, 4 or 8 clock cycles



48 Pixel Sprite (1)

We have two player sprites, each 8 pixels wide

Each can be repeated three times with an 8 pixel gap

They can be positioned to form one big 48 pixel sprite

The biggest problem is to change the graphics data at the correct time

48 Pixel Sprite (2): VDEL

VDEL enables a TIA internal graphics buffer

Originally intended to delay graphics change to the next scanline

Advantage: only one player sprite needs to be updated per scanline

Disadvantage: now every "pixel" is twice as high, the programmer looses resolution

But: this also works in the same scanline

48 Pixel Sprite (3): Code

```
@loop:
 inc $002e       ;+6= 6
 ldy dataheight;+3= 9
 lda (s0),y      ;+5=14
 sta GRP0        ;+3=17
 lda (s1),y      ;+5=22
 sta GRP1        ;+3=25
 lda (s2),y      ;+5=30
 sta GRP0        ;+3=33
 lda (s5),y      ;+5=38
 sta temp        ;+3=41
```

```
;(continued)
 lda (s4),y      ;+5=46
 tax             ;+2=48
 lda (s3),y      ;+5=53
 ldy temp        ;+3=56
 sta GRP1        ;+3=59
 stx GRP0        ;+3=62
 sty GRP1        ;+3=65
 stx GRP0        ;+3=68
 dec dataheight;+5=73
 bpl @loop       ;+3=76
```

Cycle Exact Positioning

Each CPU clock the beam travels three "pixels"

For exact positioning two components are needed:

X = (CPUclock / 3) + (remainder of 0,1 or 2)

I used a table of 160-48 = 112 bytes

Binary format: RRDDDDDD

"00RR0000" is written in HMVP0 register

"00(RR+1)0000" is written in HMVP1 register

Clockslide (1): Code

```
delayx:
 lda #>clockslide
 pha
 lda div3table,x
 and #$3f
 clc
 adc #<clockslide
 pha
 sta WSYNC
clockslide:
 rts
```

```
.byte $c9,$c9,$c9,$c9
.byte $c9,$c9,$c9,$c9
.byte $c9,$c9,$c9,$c9
.byte $c9,$c9,$c9,$c9
.byte $c9,$c9,$c9,$c9
.byte $c9,$c9,$c9,$c9
.byte $c9,$c9,$c9,$c9
.byte $c9,$c9,$c9,$c9
.byte $c9,$c9,$c9,$c5
.byte $ea
 rts
```

Clockslide (2): Disassembled

```
; "even calls"
$F0E0: cmp #$c9 ; 17
$F0E2: cmp #$c9 ; 15
$F0E4: cmp #$c9 ; 13
$F0E6: cmp #$c9 ; 11
$F0E8: cmp #$c9 ;  9
$F0EA: cmp #$c9 ;  7
$F0EC: cmp #$c9 ;  5
$F0EE: cmp $ea  ;  3
$F0F0: rts
```

```
; "odd calls"
$F0E1: cmp #$c9 ; 16
$F0E3: cmp #$c9 ; 14
$F0E5: cmp #$c9 ; 12
$F0E7: cmp #$c9 ; 10
$F0E9: cmp #$c9 ;  8
$F0EB: cmp #$c9 ;  6
$F0ED: cmp #$c5 ;  4
$F0EF: nop      ;  2
$F0F0: rts
```

How To Fill 96 Pixels

With interlacing the number of pixels can be doubled

Typically the type of sprite repetition changes from "3 close" (8 pixel gap) to "3 far" (24 pixel gap)

Using a CRT-based monitor will allow to melt two half images as one, won't work with TFT monitors

Basically it gets reduced to the question: "what price you want to pay, 'flickering' or 'gap lines'?"

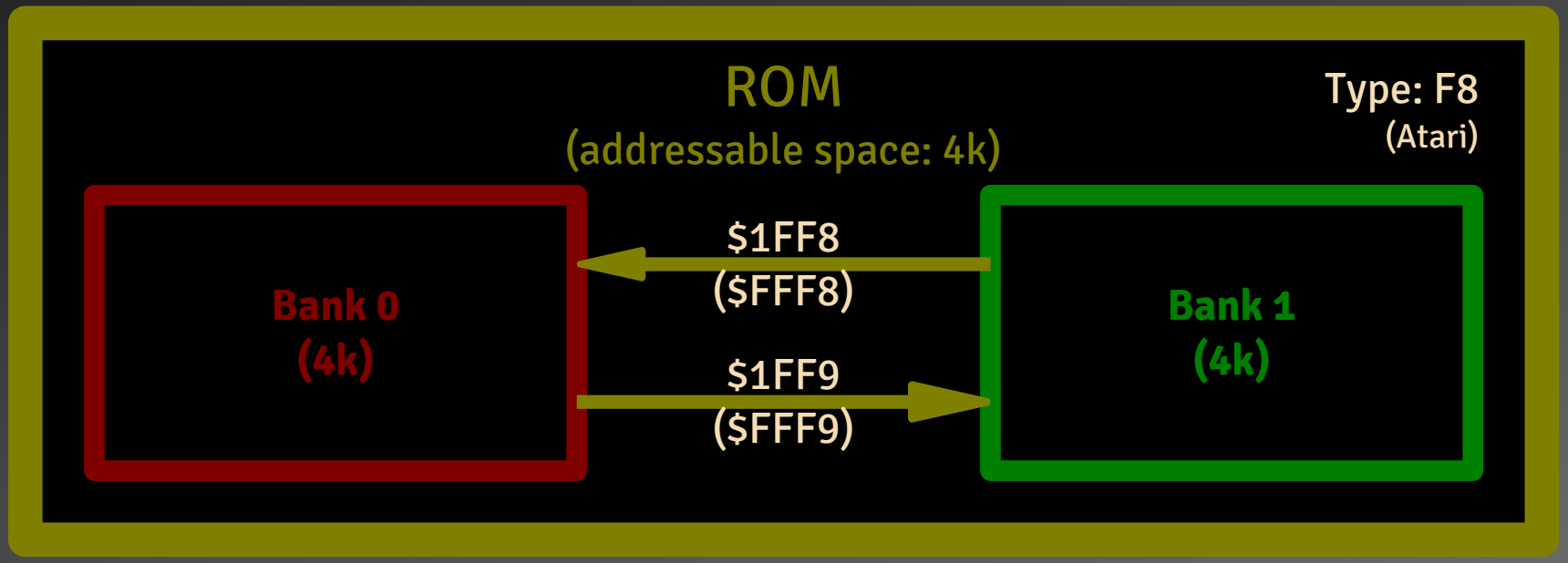
Working around the barriers (1)

At the start (1977) only 2k or 4k ROM modules

At 1981 first 8k ROM modules available

How to fit 8k in a 4k address space?

Bank switching!



Working around the barriers (2)

Now that there's enough ROM, how do we get more RAM?

Remember:
no read / write line available on game module

Solution: use different addresses

Write-port: $1000 - $107F
Read-port: $1080 - $10FF

Read $1080 to get value written to $1000

Variation of F8: F8SC (Atari)

"Cheating": The DPC+ Cartridge

There is even a way to "cheat" to get a cooler demo

The DPC+ cartridge has several extensions implemented on the ARM of the Harmony Cartridge

Based upon Activision's patented DPC cartridge used for Pitfall II

6 x 4k banks (F8/F6/F4-like)

3 channel sound using one of TIAs sound channels

Additional ROM available though "data streams"

DPC+ Cartridge: Data Streams

```
 lda #$00   ; enable
 sta $F058  ; FASTFETCH
 ldy #spriteheight
@loop:
 sta $02    ; WSYNC
 lda #$08   ; this reads first data stream
 sta $1B    ; GRP0
 lda #$09   ; this reads second data stream
 sta $1C    ; GRP1
 dey
 bne @loop
 lda #$FF   ; disable
 sta $F058  ; FASTFETCH
```

How did they do it?

From "The Ultimate Atari 2600 VCS Talk" I still owe an explanation upon the planet of Solaris is done with this limited hardware

Let's start a live Stella hacking session

Thank you for your attention!

Questions?