Nick Mikstas

ELEN 604

Final Report

Winter 2017

Table of Contents

Introduction……………………………………………………………………………. 1

System Block Diagram……………………………………………………………. 1

Block Descriptions…………………………………………………………………. 1

- NMPSM3………………………………………………………………………………………… 1
  - General Layout…………………………………………………………………………… 2
  - Command Opcodes and Clock Cycles……………………………………………. 2
  - Instruction Descriptions……………………………………………………………… 3
  - Instruction Layout………………………………………………………………………. 5
  - Control Bits……………………………………………………………………………….. 7
  - Processor Internal Layout…………………………………………………………… 10
- VGA Controller (Picture Processing Unit)…………………………………………… 11
  - PPU History……………………………………………………………………………….. 11
  - Background Tile Processing……………………………………………………….. 12
  - PPU Background Tiling Example………………………………………………… 14
  - PPU Sprite Processing………………………………………………………………… 14
  - Memory Addresses of the PPU……………………………………………………. 15
- UART……………………………………………………………………………………………… 15
- Microphone Controller……………………………………………………………………. 16
- Clock Controller………………………………………………………………………………. 16
- Timers 1 and 2………………………………………………………………………………… 17
- ROM Controller and Lookup ROM……………………………………………………. 17
- LED Controllers 1 and 2…………………………………………………………………… 18
- 100KHz Clock Enable………………………………………………………………………. 18
- 7 Segment Controller………………………………………………………………………. 18
- Data MUX………………………………………………………………………………………. 19
- MMCM……………………………………………………………………………………………. 19

Software Functionality……………………………………………………………. 20

NMPSM Assembler…………………………………………………………………. 22

Differences between Old Project and New Project……………………… 23

Meta-Stability……………………………………………………………………….. 24

Things Not Implemented…………………………………………………………. 24

Resource Usage……………………………………………………………………… 25

Timing Report………………………………………………………………………… 25

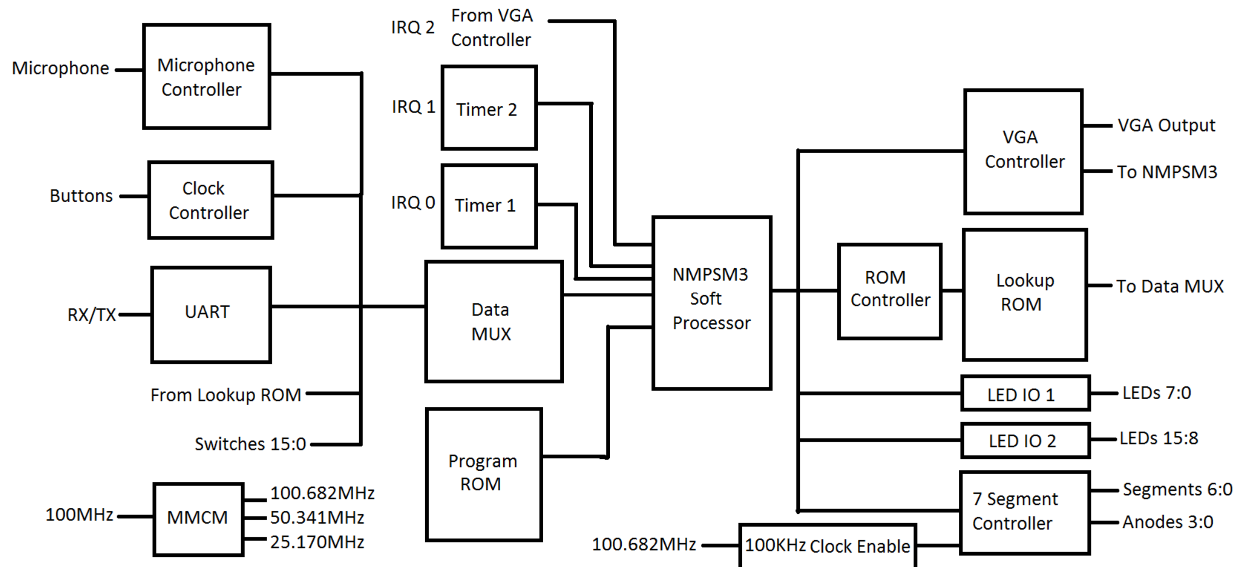
Power Report…………………………………………………………………………. 25

Conclusion…………………………………………………………………………….. 26

Introduction

This project is an update and extension of a previously built project after taking a class from UCSC extension several years back. The old project was based on a Nexys 2 board and a Spartan 3 FPGA. The new project is based on a Basys 3 board and an Artix 7 FPGA. The original design was ported over and then improved on by taking advantage of higher speeds and additional resources. Several issues were encountered in the project and had to be overcome. Those issues will be discussed later in the report.

System Block Diagram



Block Descriptions

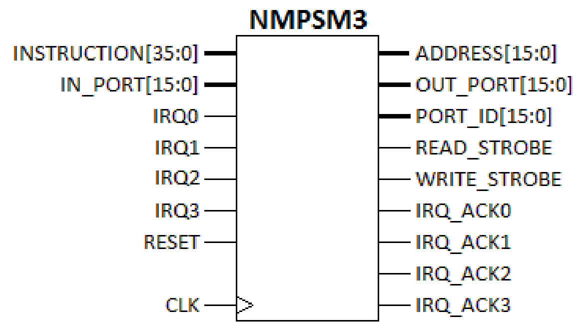
NMPSM3

The NMPSM3 is the heart of the system. It is a 16-bit soft processor inspired by the PicoBlaze design. It has four interrupt lines, flat 16-bit addressing and 16-bit port IDs. The processor is capable of speeds well over 100MHz in the Artix 7 FPGA.

The NMPSM3 uses a block RAM as a micro-coded controller. It uses another block RAM for the stack and another for the registers. The NMPSM3 has 1024 16-bit general purpose registers and a 1024-word stack. At least one more block RAM is required for the program ROM. Each 18K block RAM can hold 512 instructions.

The NMPSM3 supports 63 instructions. Most instructions execute in 2 clock cycles but some of indirect instructions can take 3 clock cycles.

General Layout



Command Opcodes and Clock Cycles

Command	Opcode	Clock Cycles	Command	Opcode	Clock Cycles
LOAD rx, #k	$01	2	OR rx, ry	$63	2
LOAD rx, ry	$04	2	XOR rx, #k	$66	2
LOAD rx, (ry)	$07	3	XOR rx, ry	$69	2
STOR rx, ry	$0A	2	ADD rx, #k	$6C	2
STOR rx, (ry)	$0D	3	ADD rx, ry	$70	2
JUMP a	$10	2	ADDC rx, #k	$73	2
JUMP (rx)	$13	2	ADDC rx, ry	$76	2
JPNZ a	$16	2	SUB rx, #k	$79	2
JPZ a	$19	2	SUB rx, ry	$7C	2
JPNC a	$1C	2	SUBC rx, #k	$80	2
JPC a	$20	2	SUBC rx, ry	$83	2
CALL a	$23	2	TEST rx, #k	$86	2
CALL (rx)	$26	2	TEST rx, ry	$89	2
CLNZ a	$29	2	COMP rx, #k	$8C	2
CLZ a	$2C	2	COMP rx, ry	$90	2
CLNC a	$30	2	ASL rx	$93	2
CLC a	$33	2	ROL rx	$96	2
RET	$36	3	LSR rx	$99	2
RTNZ	$39	3	ROR rx	$9C	2
RTZ	$3C	3	SETC	$A0	2
RTNC	$40	3	CLRC	$A3	2
RTC	$43	3	EIN0	$A6	2
RTIE	$46	3	EIN1	$A9	2
RTID	$49	3	EIN2	$AC	2
IN rx, #k	$4C	2	EIN3	$B0	2
IN rx, ry	$50	2	DIN0	$B3	2
OUT rx, #k	$53	2	DIN1	$B6	2
OUT rx, ry	$56	2	DIN2	$B9	2
AND rx, #k	$59	2	DIN3	$BC	2
AND rx, ry	$5C	2	PUSH rx	$C0	2
OR rx, #k	$60	2	POP rx	$C6	3

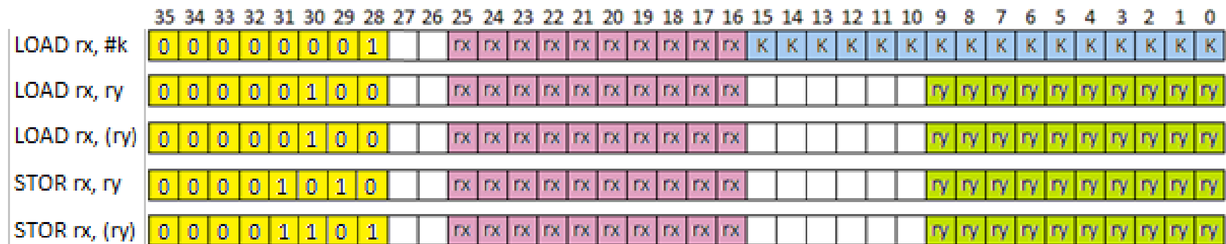
Instruction Descriptions

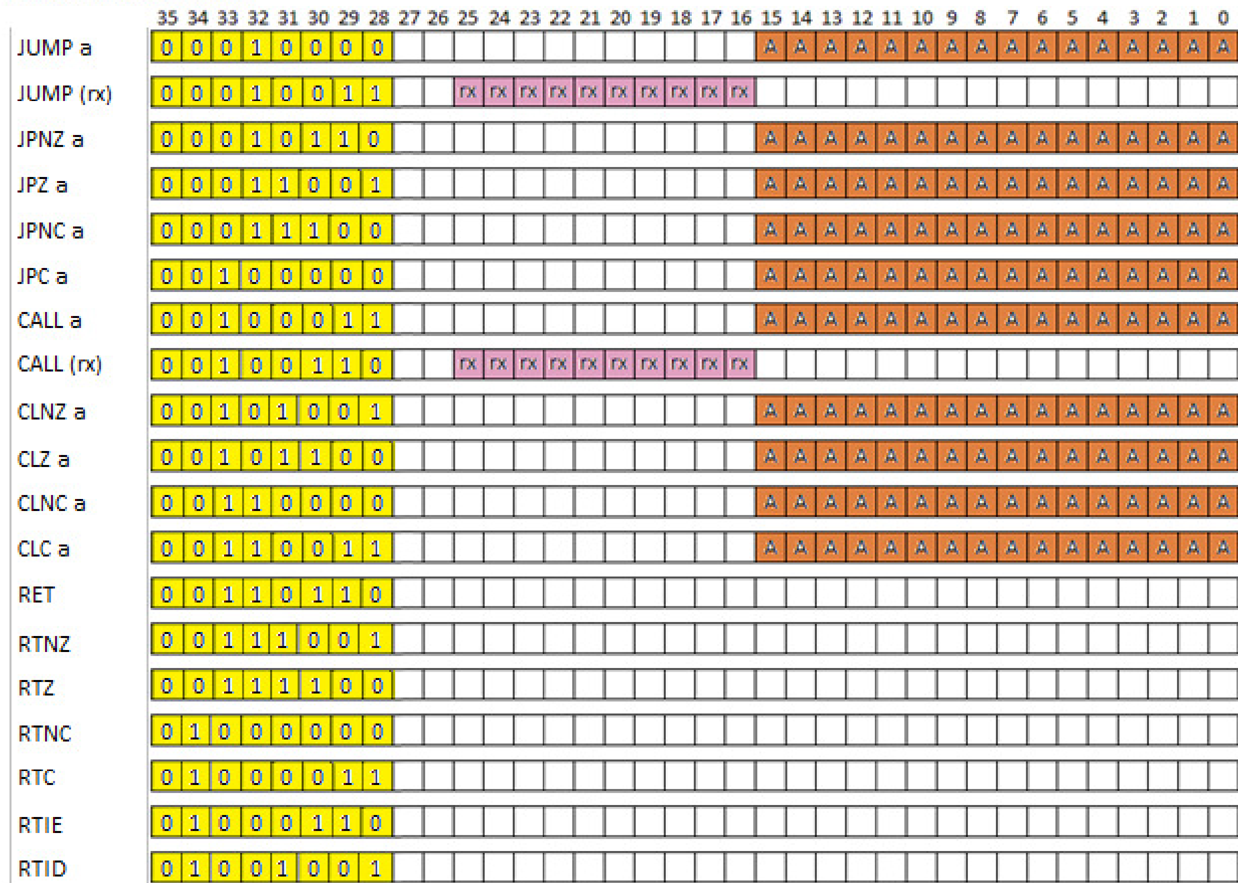
Instruction	Description	Carry Status	Zero Status
LOAD rx, #k	Load register rx with constant k	No change	No change
LOAD rx, ry	Load register rx with contents of register ry	No change	No change
LOAD rx, (ry)	Load register rx with contents of register whose address is stored in register ry	No change	No change
STOR rx, ry	Store contents of register rx in register ry	No change	No change
STOR rx, (ry)	Store register rx in register whose address is stored in register ry	No change	No change
JUMP a	Jump to address a	No change	No change
JUMP (rx)	Jump to address stored in register rx	No change	No change
JPNZ a	If zero flag not set, jump to address a, else increment to next instruction	No change	No change
JPZ a	If zero flag is set, jump to address a, else increment to next instruction	No change	No change
JPNC a	If carry flag is not set, jump to address a, else increment to next instruction	No change	No change
JPC a	If carry flag is set, jump to address a, else increment to next instruction	No change	No change
CALL a	Call subroutine at address a	No change	No change
CALL (rx)	Call subroutine whose address is stored in register rx	No change	No change
CLNZ a	If zero flag not set, call subroutine at address a, else increment to next instruction	No change	No change
CLZ a	If zero flag set, call subroutine at address a, else increment to next instruction	No change	No change
CLNC a	If carry flag not set, call subroutine at address a, else increment to next instruction	No change	No change
CLC a	If carry flag set, call subroutine at address a, else increment to next instruction	No change	No change
RET	Return from subroutine	No change	No change
RTNZ	If zero flag not set return from subroutine, else increment to next instruction	No change	No change
RTZ	If zero flag set return from subroutine, else increment to next instruction	No change	No change
RTNC	If carry flag not set return from subroutine, else increment to next instruction	No change	No change
RTC	If carry flag set return from subroutine, else increment to next instruction	No change	No change
RTIE	Return from interrupt and enable interrupts	No change	No change
RTID	Return from interrupt and disable interrupts	No change	No change
IN rx, #k	Save data on input port k in RAM address rx	No change	No change
IN rx, ry	Save data on input port described in register ry in register rx	No change	No change
OUT rx, #k	Send data stored in register rx to output port k	No change	No change
OUT rx, ry	Send data stored in register rx to output port described in register ry	No change	No change

AND rx, #k	AND register rx with k. Store results in register rx	0	Varies
AND rx, ry	AND register rx with register ry. Store results in register rx	0	Varies
OR rx, #k	OR register rx with k. Store results in register rx	0	Varies
OR rx, ry	OR register rx with register ry. Store results in register rx	0	Varies
XOR rx, #k	XOR register rx with k. Store results in register rx	0	Varies
XOR rx, ry	XOR register rx with register ry. Store results in register rx	0	Varies
ADD rx, #k	Add register rx with k. Store results in register rx	Varies	Varies
ADD rx, ry	Add register rx with register ry. Store results in register rx	Varies	Varies
ADDC rx, #k	Add with carry register rx with k. Store results in register rx	Varies	Varies
ADDC rx, ry	Add with carry register rx with register ry. Store results in register rx	Varies	Varies
SUB rx, #k	Subtract register rx with k. Store results in register rx	Varies	Varies
SUB rx, ry	Subtract register rx with register ry. Store results in register rx	Varies	Varies
SUBC rx,#k	Subtract with carry register rx with k. Store results in register rx	Varies	Varies
SUBC rx, ry	Subtract with carry register rx with register ry. Store results in register rx	Varies	Varies
TEST rx, #k	Bit test register rx with constant k	If rx and k = 0, zero = 1	Carry = odd parity of rx AND k
TEST rx, ry	Bit test register rx with register ry	If rx and ry = 0, zero = 1	Carry = odd parity of rx AND ry
COMP rx, #k	Compare register rx with constant k	If rx < k, carry = 1	If rx = k, zero = 1
COMP rx, ry	Compare register rx with register ry	If rx < ry, carry = 1	If rx = ry, zero = 1
ASL rx	Shift register rx left into carry. Backfill with zero	Varies	Varies
ROL rx	Rotate register rx left into carry. Backfill with carry	Varies	Varies
LSR rx	Shift register rx right into carry. Backfill with zero	Varies	Varies
ROR rx	Rotate register rx right into carry. Backfill with carry	Varies	Varies
SETC	Set carry bit	1	No change
CLRC	Clear carry bit	0	No change
EIN0	Enable interrupt 0	No change	No change
EIN1	Enable interrupt 1	No change	No change
EIN2	Enable interrupt 2	No change	No change
EIN3	Enable interrupt 3	No change	No change
DIN0	Disable interrupt 0	No change	No change
DIN1	Disable interrupt 1	No change	No change

DIN2	Disable interrupt 2	No change	No change
DIN3	Disable interrupt 3	No change	No change
PUSH rx	Push contents of register rx on to stack	No change	No change
POP rx	Pop top of stack and store contents in rx	No change	No change

Instruction Layout





Port Commands

	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IN rx, #k	0	1	0	0	1	1	0	0			rx	rx	rx	rx	rx	rx	rx	rx	rx	rx	K	K	K	K	K	K	K	K	K	K	K	K	K	K	K	K
IN rx, ry	0	1	0	1	0	0	0	0			rx	rx	rx	rx	rx	rx	rx	rx	rx	rx							ry	ry	ry	ry	ry	ry	ry	ry	ry	ry
OUT rx, #k	0	1	0	1	0	0	1	1			rx	rx	rx	rx	rx	rx	rx	rx	rx	rx	K	K	K	K	K	K	K	K	K	K	K	K	K	K	K	K
OUT rx, ry	0	1	0	1	0	1	1	0			rx	rx	rx	rx	rx	rx	rx	rx	rx	rx							ry	ry	ry	ry	ry	ry	ry	ry	ry	ry

ALU Commands

	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AND rx, #k	0	1	0	1	1	0	0	1			rx	rx	rx	rx	rx	rx	rx	rx	rx	rx	K	K	K	K	K	K	K	K	K	K	K	K	K	K	K	K
AND rx, ry	0	1	0	1	1	1	0	0			rx	rx	rx	rx	rx	rx	rx	rx	rx	rx							ry	ry	ry	ry	ry	ry	ry	ry	ry	ry
OR rx, #k	0	1	1	0	0	0	0	0			rx	rx	rx	rx	rx	rx	rx	rx	rx	rx	K	K	K	K	K	K	K	K	K	K	K	K	K	K	K	K
OR rx, ry	0	1	1	0	0	0	1	1			rx	rx	rx	rx	rx	rx	rx	rx	rx	rx							ry	ry	ry	ry	ry	ry	ry	ry	ry	ry
XOR rx, #k	0	1	1	0	0	1	1	0			rx	rx	rx	rx	rx	rx	rx	rx	rx	rx	K	K	K	K	K	K	K	K	K	K	K	K	K	K	K	K
XOR rx, ry	0	1	1	0	1	0	0	1			rx	rx	rx	rx	rx	rx	rx	rx	rx	rx							ry	ry	ry	ry	ry	ry	ry	ry	ry	ry
ADD rx, #k	0	1	1	0	1	1	0	0			rx	rx	rx	rx	rx	rx	rx	rx	rx	rx	K	K	K	K	K	K	K	K	K	K	K	K	K	K	K	K
ADD rx, ry	0	1	1	1	0	0	0	0			rx	rx	rx	rx	rx	rx	rx	rx	rx	rx							ry	ry	ry	ry	ry	ry	ry	ry	ry	ry
ADDC rx, #k	0	1	1	1	0	0	1	1			rx	rx	rx	rx	rx	rx	rx	rx	rx	rx	K	K	K	K	K	K	K	K	K	K	K	K	K	K	K	K
ADDC rx, ry	0	1	1	1	0	1	1	0			rx	rx	rx	rx	rx	rx	rx	rx	rx	rx							ry	ry	ry	ry	ry	ry	ry	ry	ry	ry
SUB rx, #k	0	1	1	1	1	0	0	1			rx	rx	rx	rx	rx	rx	rx	rx	rx	rx	K	K	K	K	K	K	K	K	K	K	K	K	K	K	K	K
SUB rx, ry	0	1	1	1	1	1	0	0			rx	rx	rx	rx	rx	rx	rx	rx	rx	rx							ry	ry	ry	ry	ry	ry	ry	ry	ry	ry
SUBC rx, #k	1	0	0	0	0	0	0	0			rx	rx	rx	rx	rx	rx	rx	rx	rx	rx	K	K	K	K	K	K	K	K	K	K	K	K	K	K	K	K
SUBC rx, ry	1	0	0	0	0	0	1	1			rx	rx	rx	rx	rx	rx	rx	rx	rx	rx							ry	ry	ry	ry	ry	ry	ry	ry	ry	ry
TEST rx, #k	1	0	0	0	0	1	1	0			rx	rx	rx	rx	rx	rx	rx	rx	rx	rx	K	K	K	K	K	K	K	K	K	K	K	K	K	K	K	K
TEST rx, ry	1	0	0	0	1	0	0	1			rx	rx	rx	rx	rx	rx	rx	rx	rx	rx							ry	ry	ry	ry	ry	ry	ry	ry	ry	ry
COMP rx, #k	1	0	0	0	1	1	0	0			rx	rx	rx	rx	rx	rx	rx	rx	rx	rx	K	K	K	K	K	K	K	K	K	K	K	K	K	K	K	K
COMP rx, ry	1	0	0	1	0	0	0	0			rx	rx	rx	rx	rx	rx	rx	rx	rx	rx							ry	ry	ry	ry	ry	ry	ry	ry	ry	ry
ASL rx	1	0	0	1	0	0	1	1			rx	rx	rx	rx	rx	rx	rx	rx	rx	rx																
ROL rx	1	0	0	1	0	1	1	0			rx	rx	rx	rx	rx	rx	rx	rx	rx	rx																
LSR rx	1	0	0	1	1	0	0	1			rx	rx	rx	rx	rx	rx	rx	rx	rx	rx																
ROR rx	1	0	0	1	1	1	0	0			rx	rx	rx	rx	rx	rx	rx	rx	rx	rx																
SETC	1	0	1	0	0	0	0	0																												
CLRC	1	0	1	0	0	0	1	1																												

Interrupt Commands

	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EIN0	1	0	1	0	0	1	1	0																												
EIN1	1	0	1	0	1	0	0	1																												
EIN2	1	0	1	0	1	1	0	0																												
EIN3	1	0	1	1	0	0	0	0																												
DIN0	1	0	1	1	0	0	1	1																												
DIN1	1	0	1	1	0	1	1	0																												
DIN2	1	0	1	1	1	0	0	1																												
DIN3	1	0	1	1	1	1	0	0																												

Stack Commands

	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PUSH rx	1	1	0	0	0	0	0	0			rx	rx	rx	rx	rx	rx	rx	rx	rx	rx																
POP rx	1	1	0	0	0	1	1	0			rx	rx	rx	rx	rx	rx	rx	rx	rx	rx																

Control Bits

Section #	Bit #(s)	Description
A	0	1 = Load next instruction address from decoder ROM. 0 = Load next instruction from IRQ or program ROM.
B	1	PC register control bit. 1 = Load program counter.
C	4,3,2	Address MUX control bits in program counter. 000 = addr. 001 = addrp1. 010 = INSTRUCTION. 011 = Port A. 100 = IRQload. 101 = portB.
D	8,7,6,5	IRQ register control bits in decoder. 0001 = Enable IRQ0. 0010 = Enable IRQ1. 0011 = Enable IRQ2. 0100 = Enable IRQ3. 0101 = Disable IRQ0. 0110 = Disable IRQ1. 0111 = Disable IRQ2. 1000 = Disable IRQ3. 1001 = Enable IRQ master control. (RTIE) 1010 = Enable IRQ master control, disable all IRQs. (RTID) 1011 = Disable IRQ master control. (IRQ called)
E	10,9	Port ID MUX control bits in port controller. 00 = 16'b0. 01 = Port B. 10 = INSTRUCTION.

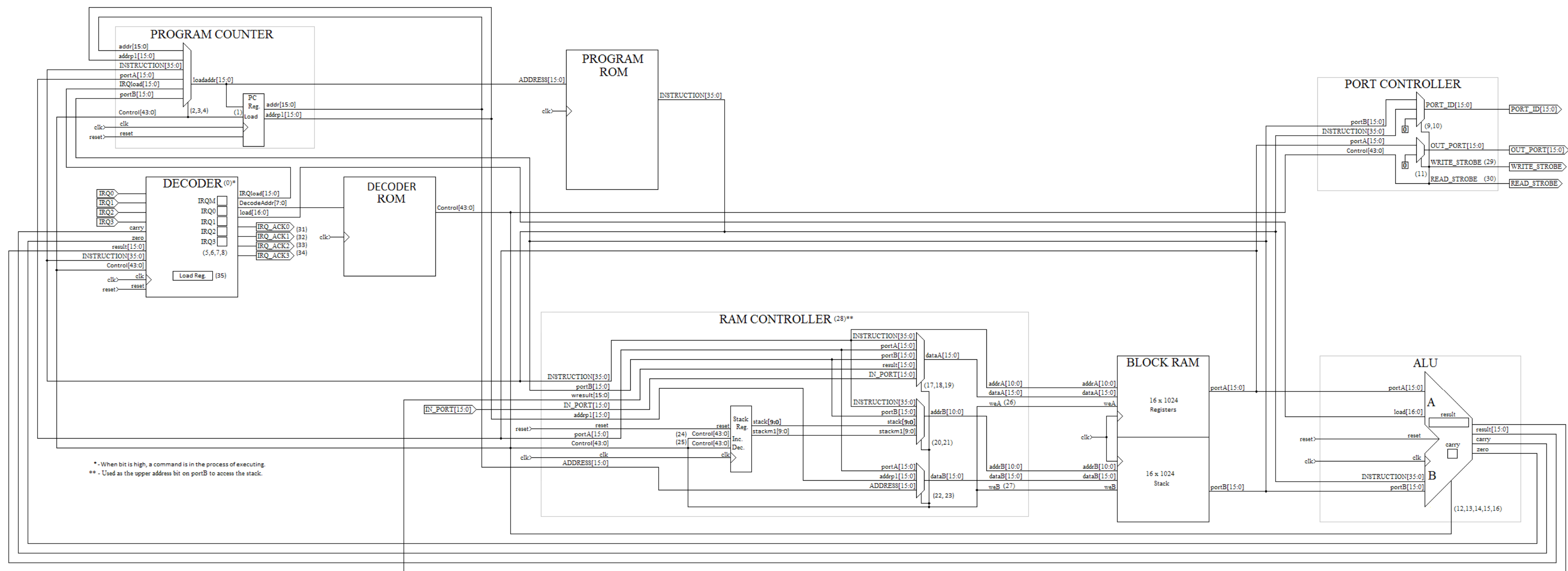
F	11	Out port MUX control bits in port controller. 0 = 16’b0. 1 = Port A.

G	16,15,14,13,12	ALU control bits.
		00000 = No change.
		00001 = LOAD
		00010 = OR rx, #k
		00011 = OR rx, (ry)
		00100 = AND rx, #k
		00101 = AND rx, (ry)
		00110 = XOR rx, #k
		00111 = XOR rx, (ry)
		01000 = ADD rx, (ry)
		01001 = ADDC rx, (ry)
		01010 = SUB rx, (ry)
		01011 = SUBC rx, (ry)
		01100 = ADD rx, #k
		01101 = ADDC rx, #k
		01110 = SUB rx, #k
		01111 = SUBC rx, #k
		10000 = TEST rx, #k
		10001 = TEST rx, ry
		10010 = COMP rx, #k
		10011 = COMP rx, (ry)
		10100 = ASL rx
		10101 = ROL rx
		10110 = LSR rx
		10111 = ROR rx
		11000 = CLRC
		11001 = SETC

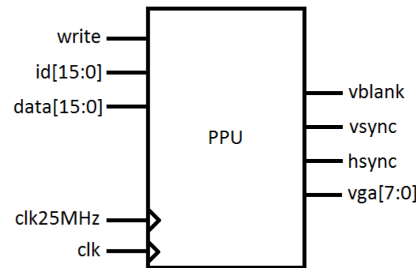
H	19,18,17	Data A MUX control bits in RAM controller. 000 = INSTRUCTION. 001 = Port A. 010 = Port B. 011 = wresult. 100 = IN_PORT.
I	21,20	Address B MUX control bits in RAM controller. 00 = INSTRUCTION. 01 = Stack register. 10 = Port B. 11 = Stackm1.
J	23, 22	Data B MUX control bits in RAM controller. 00 = Port A. 01 = Addrp1. 10 = ADDRESS.
K	25,24	01 = Increment stack register. 10 = Decrement stack register.

L	26	1 = Write enable port A.
M	27	1 = Write enable port B.
N	28	Upper address bit of port B. Used to access the stack.
O	30,29	01 = write strobe. 10 = read strobe.
P	34,33,32,31	0001 = IRQ0 ack. 0010 = IRQ1 ack. 0100 = IRQ2 ack. 1000 = IRQ3 ack.
Q	35	1 = Load the load register in decoder.
R	43,42,41,40,39,38,37,36	Address of next decoder state.

Processor Internal Layout



VGA Controller (Picture Processing Unit)



PPU History

The PPU was first created by Nintendo for use in the Nintendo Entertainment System (NES). The NES was an eight-bit gaming console that was popular in the 1980s. The NES used a 6502 processor for its main processor and needed a second processor to handle the game graphics as the 6502 was not powerful enough for both tasks. The picture processing unit was developed for this purpose.

The PPU is a tile based processor and does not have the ability to draw geometric images directly to the screen. All graphics must be reduced to a series of 8 by 8 pixel tiles. The tiles are then arranged into a screen buffer (called a name table) and sent out to the display.

The PPU has two main functions: background tile processing and sprite processing. The background tiles are fairly static and can only start in columns and rows that are a multiple of eight. Because of these limitations, background tiles require only minimal processing to be displayed on the screen. The second function of the PPU is to do sprite processing. Each sprite is also composed of 8 by 8 pixel tiles. Up to 256 sprites can be displayed on the screen at any one time. The original Nintendo could only handle 32 sprites and only 8 could be on the same scanline at a time. Sprites can appear at any location on the display. Because sprites can appear anywhere at any time, the amount of processing required to render the sprites is considerably more than that required to render the background.

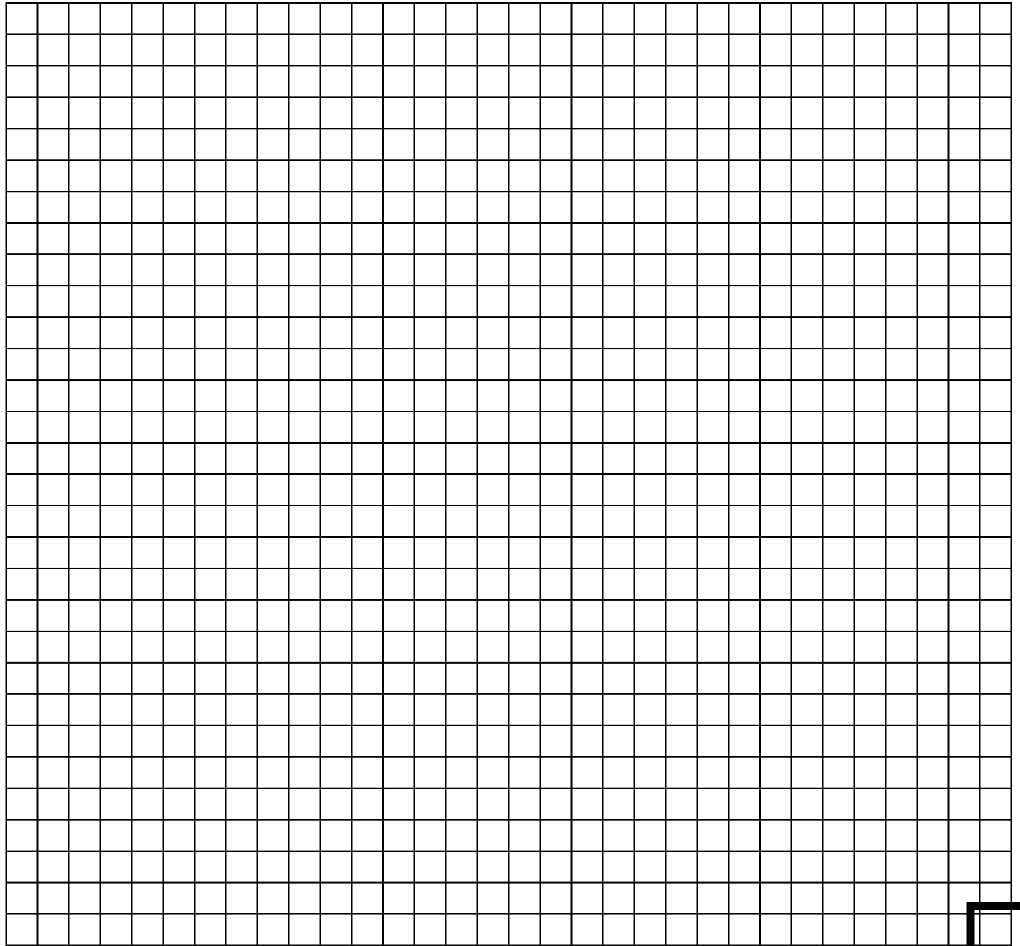
The PPU developed for this project is not an exact recreation of the Nintendo PPU and is missing some of its features, such as multiple name tables and the ability to scroll between those name tables. The PPU in this project also has some improvements over the original, including the ability to handle 256 different colors. The original PPU could only handle 64 colors.

Background Tile Processing

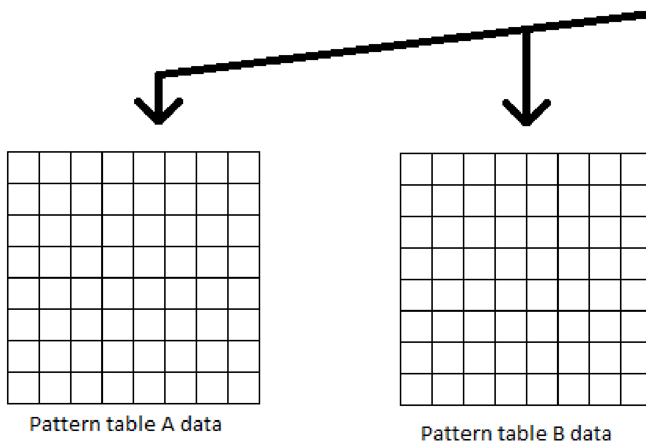
The following graphics describe the general theory behind how the background tiles are drawn on the display:

1. The location of the current tile is accessed in the name table

The name table is a screen buffer that divides the screen into tiles. The screen is divided into 32 by 30 tiles. Each tile is 8 by 8 pixels. This creates a screen resolution of 256 by 240 pixels.



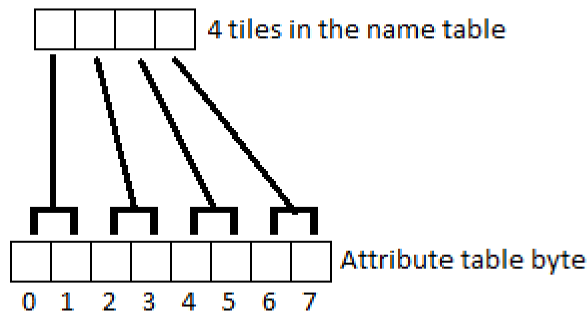
2. pattern table A and B access



An 8 bit value if loaded from an entry in the name table into the A and B pattern tables. That value is an index into the pattern tables which extract 8 bytes of data that represent the tile battern. The two patterns are overlayed to create a single tile.

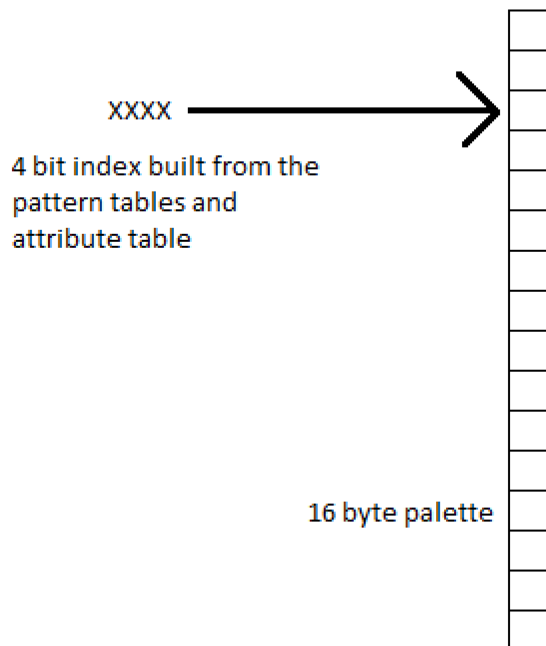
3. Attribute table access

At the same time the pattern tables are being accessed, the attribute table is being accessed. The attribute table is being accessed. Each byte in the attribute table contains the upper two color bits of four tiles in the name table.



4. Palette table access

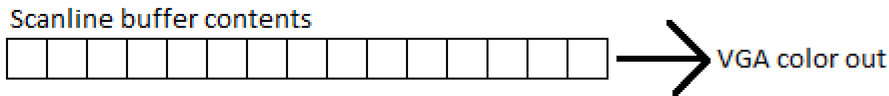
Once the two attribute table bits and the color bits from each of the pattern tables are loaded, they are used as an index into the color palette. Bits 3 and 2 are taken from the attribut table, bit 1 is taken from pattern table B and bit 0 is taken from pattern table A.



5. Scanline buffer write

Once the palette color is selected, the byte is written to the scanline buffer. The scanline buffer is 256 bytes long and buffers the contents of the next row to be drawn on the screen.

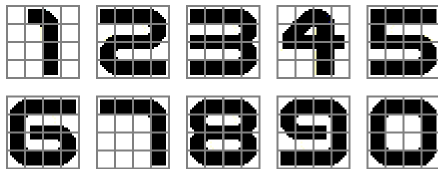
6. VGA out



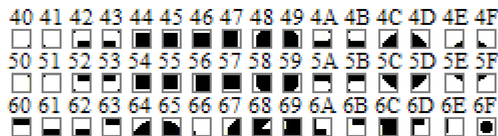
The contents of the scanline buffer are then placed on the output of the VGA port which causes a pixel to be placed on the display.

PPU Background Tiling Example

The following is an example of how the background tiling is done for the numbers on the clock:



First, the numbers are broken down into 4 by 4 tile squares. The individual tiles are loaded into the pattern tables. In this example, the numbers are a single color so only pattern table A is loaded with the number tiles while pattern table B is left blank. Below is what the patterns look like when they are divided into individual tiles:



The 48 tiles above contain all the tiles necessary to draw any of the ten digits into the name table. The numbers above the tiles represent the index numbers for each of the corresponding tiles. The index numbers are the actual data that is written into the name tables.

PPU Sprite Processing

Sprite processing is similar to background tile processing in that the sprite patterns come from a pair of pattern tables. The pattern tables used by the sprites are separate from the pattern tables used by the background. Also, the palette used by the sprites is a sixteen-byte palette and is separate from the background palette. Sprites differ from background tiles in that there is no name table that contains

sprite information. Instead of a name table, sprites use their own RAM which is integrated into the PPU. There are 256 bytes of sprite RAM and each sprite requires 4 bytes of RAM each. This allows for a maximum of 64 sprites to be displayed on the screen at any one time. Below is a description of the four bytes used by each sprite:

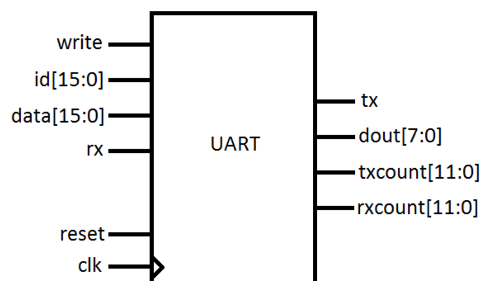
Byte0: YYYYYYYY
Byte1: VHNNNNCC
Byte2: PPPPPPPP
Byte3: XXXXXXXX
Y: Y coordinates of upper left hand corner of sprite.
V: 1 = Flip sprite vertically, 0 = draw sprite normally.
H: 1 = Flip sprite horizontally, 0 = draw sprite normally.
P: 8-bit address for tile pattern.
X: X coordinates of upper left hand corner of sprite.
N: Not used.

Once it is determined that a sprite is within range of the current scan line, the pixels for the sprite are drawn in the scan line buffer. Any background data already in the buffer will be overwritten. Overwriting background tile data in the scan line buffer has the effect of always drawing sprites on top of the background. The color black in the sprite tiles is transparent and does not overwrite background tile color information.

Memory addresses of the PPU

0x8000 - 0x87FF Background pattern table A.
0x8800 - 0x8FFF Background pattern table B.
0x9000 - 0x97FF Sprite pattern table A.
0x9800 - 0x9FFF Sprite pattern table B.
0xA000 - 0xA3FF Name table.
0xA400 - 0xA4FF Unused.
0xA500 - 0xA5FF Background attribute table.
0xA600 - 0xA60F Background palettes 0 through 15.
0xA610 - 0xA61F Sprite palettes 0 through 15.
0xA700   Base color (background color).
0xB000 - 0xB3FF Sprite RAM (256 sprites).

UART



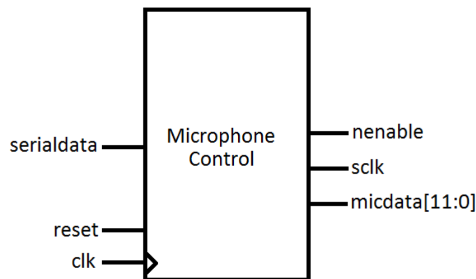
The UART uses two block RAMs. One block RAM for the receive buffer and one for the transmit buffer. Each block RAM can hold 4096 bytes. The UART can handle any baud rate up to half the system clock frequency. The baud rate control register is set by the following equation:

System Clock Frequency / Desired Baud Rate / 2

Control registers for the UART

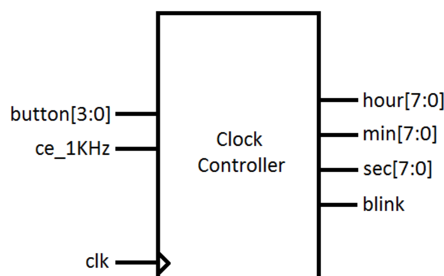
0x0200 Set baud rate.
0x0201 Buffer byte to transmit.
0x0202 Flush transmit buffer.
0x0203 Purge transmit buffer.
0x0204 Get next byte in the receive buffer.
0x0205 Purge the receive buffer.

Microphone Controller



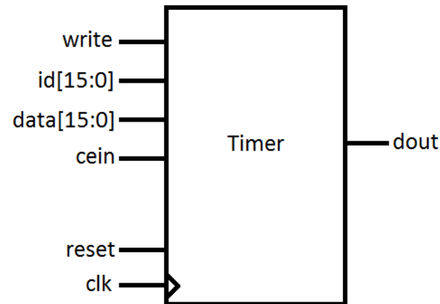
The microphone controller is an SPI based controller. The SPI clock runs at 25 MHz and continuously samples the digital output of the microphone at 1.5 MSPS. The digital output is made available on the micdata port and is automatically written over every time a new sample is available. There are no controls other than reset for this module.

Clock Controller



The clock controller is used to keep track of the time. The output of the controller is visible on the VGA display. A hardware controller was used to keep track of the time as an accurate clock using software would have been more complicated. The input buttons are used to set the time. The base clock frequency is controlled by the 1KHz clock enable pin. The NMPSM3 has no direct control over the clock controller.

Timers 1 and 2

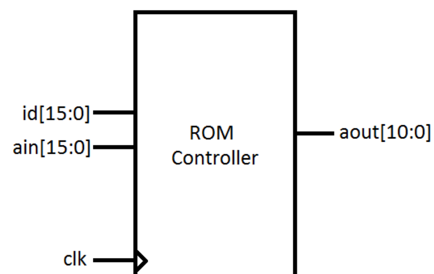


The timers are set by the NMPSM3 through the id and data ports. The timers each have a 16-bit countdown register and a time resolution of 1 ms. The outputs of the timers are tied to the interrupts on the processor and provide the timing pulses used to update the 7-segment display and the LEDs.

Control registers for the timers

0x0010 Set timer 1.
0x0011 Set timer 2.

ROM Controller and Lookup ROM

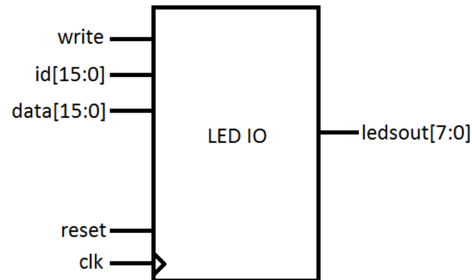


The ROM controller brakes the lookup ROM into eight 256 byte blocks. This allows the address to the lookup ROM to be latched while the processor performs a read.

ID values for the ROM controller

0x0000 ROM bank 0.
0x0001 ROM bank 1.
0x0002 ROM bank 2.
0x0003 ROM bank 3.
0x0004 ROM bank 4.
0x0005 ROM bank 5.
0x0006 ROM bank 6.
0x0007 ROM bank 7.

LED Controllers 1 and 2

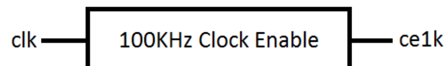


There are two LED controllers in the system. Each controller controls eight LEDs. LED controller 1 controls the lower eight LEDs and is runs through a set pattern loaded in the Lookup ROM. The second LED controller controls the upper eight LEDs and displays the ASCII value of the last byte to be received on the UART.

Control registers for the LED controller

0x0020 Set LED pattern on controller 1.
0x0021 Set LED pattern on controller 2.

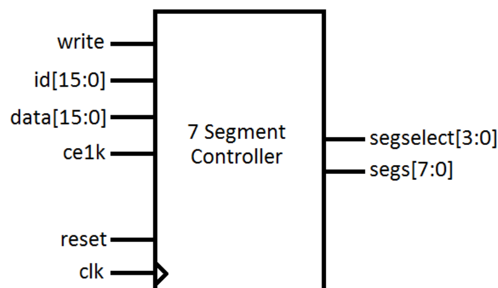
100KHz Clock Enable



The 100KHz clock enable block is a simple module that takes the 100MHz system clock and generates a 100KHz clock enable pulse. No other controls are part of this block.

7 Segment Controller

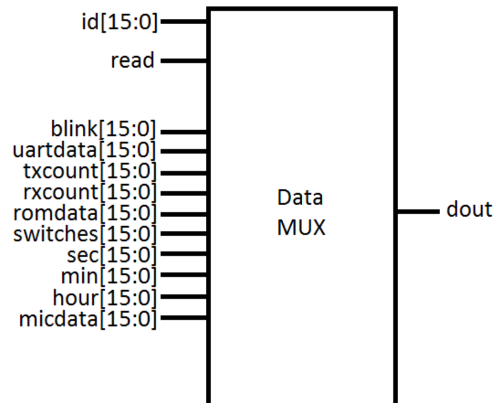
The 7 segment displays on the Nexys 3 board are configured with a common anode. There is only one set of control signals for all four displays. The anodes for the displays are separate. Because of this configuration, the 7 segments can only be turned on one at a time. This is accomplished by rapidly cycling through the anode control pins. The controller takes care of this operation in a way invisible to the user. The values to be displayed are the only thing required. The 100KHz clock enable signal is required for the timing of the anode cycling.



Control registers for the 7-segment display controller

0x0024 7-segment 0.
0x0024 7-segment 1.
0x0024 7-segment 2.
0x0024 7-segment 3.

Data MUX



All the input data to the processor passes through the data MUX. The processor only has one input port so the data MUX is necessary to control the data flow. The data MUX is purely combinational logic. The ID port and read bit act as the select logic for the MUX.

ID values for the data MUX

0x0001 - I2C data (Not used in this design).
0x0002 - I2C status (Not used in this design).
0x0003 - UART data.
0x0004 - UART transmit buffer bytes.
0x0005 - UART receive buffer bytes.
0x0006 - Joystick X position (Not used in this design).
0x0007 - Joystick y position (Not used in this design).
0x0012 - ROM data.
0x0013 - Switches.
0x0030 - Clock controller seconds.
0x0031 - Clock controller minutes.
0x0032 - Clock controller Hours.
0x0033 - Clock controller blink bit.
0x0034 - Microphone data.

MMCM

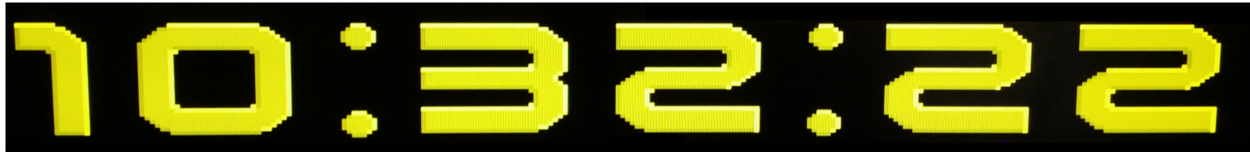
The Mixed Mode Clock Controller (MMCM) takes the system clock as an input and divides it into three output frequencies. The MMCM is an IP core and was generated using built-in Xilinx tools. The input

clock is 100MHz and the three output clocks are 100.682MHz, 50.341MHz and 25.170MHz. The 25.170MHz clock runs the VGA circuits in the system. The standard frequency for a 640 x 480 x 60Hz VGA signal is 25.175MHz. The 100.682MHz clock runs the rest of the system and is 4 times the frequency of the VGA clock.

Software Functionality

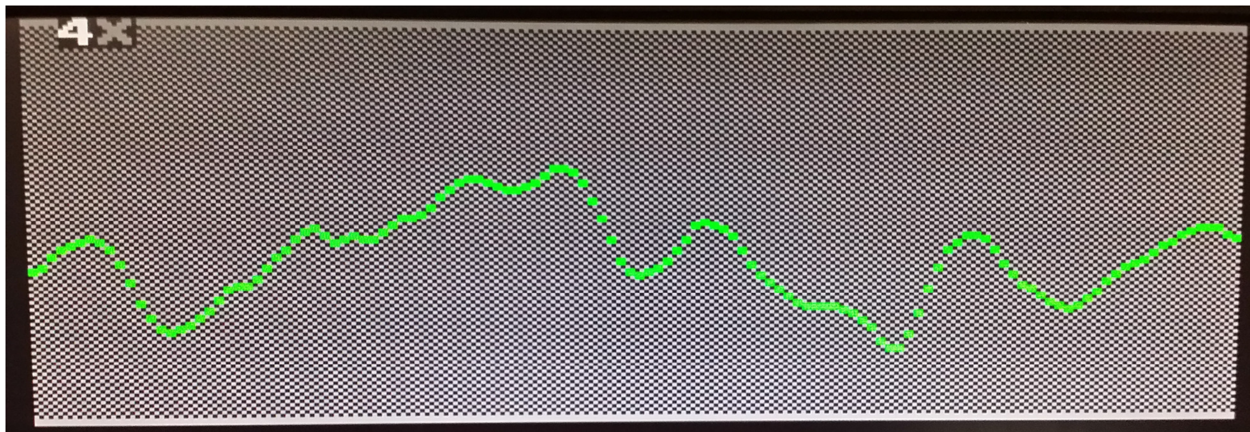
Tiled Clock

The tiled clock is a tiled based clock that displays on the VGA output. The hours, upper minutes and lower minutes can be set by using the three middle buttons on the Basys board. Below is a screenshot of a sample output.



Sprite Audio

The sprite audio takes periodic samples from the microphone and displays them on the VGA output. There is a total of 128 audio sprites. The sample rate can be adjusted by pressing the upper button on the Basys board. The sample rate doubles every time the button is pressed and wraps around to 1x after going to 32x. Below is a screenshot sample of the audio sprites.



Base Color

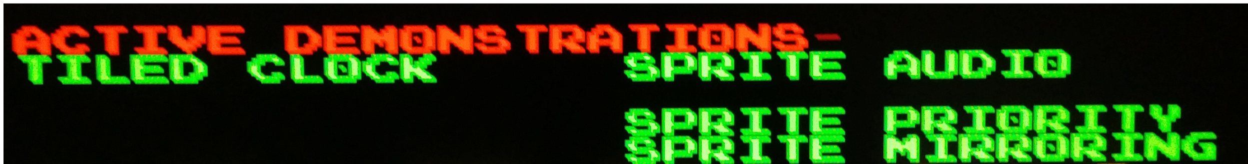
The background color of the screen can be one of 256 VGA colors. Enabling the base color demonstration increments the background through all possible colors.

Bouncing Sprites

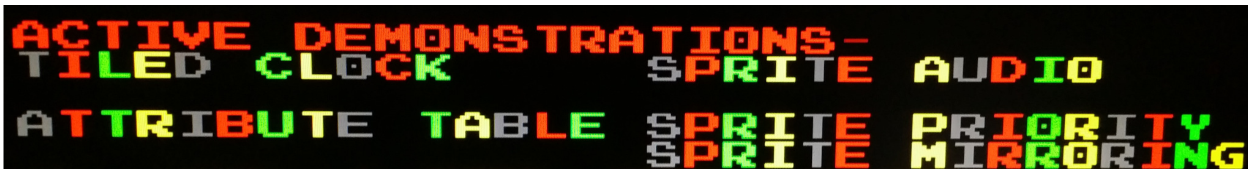
There are eight sprites that bounce around the VGA display at 45 degree angles. Enabling the bouncing sprite demonstration produces the following result:

Attribute Table

Color information is applied to 8x8 pixel blocks of the display. The attribute table demonstration shows the color blocks. Below is a sample of the on-screen text without the attribute table demonstration applied.



And the following is the same text with the attribute table demonstration active.



Sprite Priority

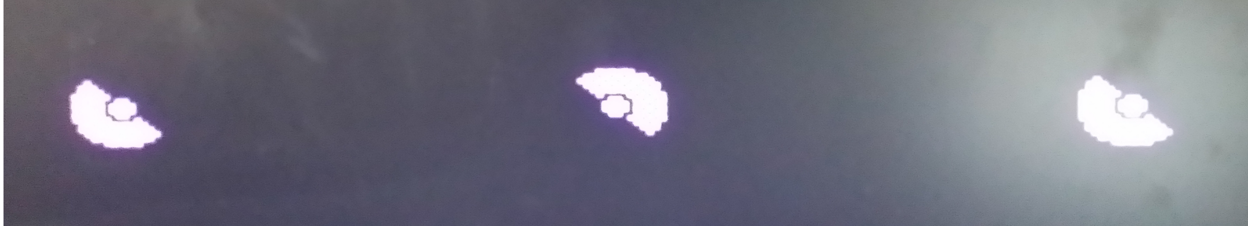
The sprite priority demonstration shows the ability of the sprites to be drawn in front of or behind the background tiles. A character runs across the bottom of the screen and passes in front of pillars and then behind them.

Palette Change

The palette change demonstration changes periodically rotates through the selected palettes for the screen graphics. This creates a flashing effect.

Sprite Mirroring

The PPU has the ability to mirror sprites vertically, horizontally, in both directions or not at all. A set of three objects on the screen have their sprites mirrored creating a rotating effect. The following is a screenshot sample of the sprite mirroring demonstration:



UART Echo and LED Control

One final functionality of the software is to echo back any information typed on the UART. The binary ASCII code of the typed character is also displayed on the eight upper LEDs on the Basys board.

NMPSM Assembler

Assembler Syntax

The NMPSM assembler is a Java based command line assembler. The assembler requires at least one argument to be passed to it during execution. The argument is the name of the assembly language source file and the optional second argument is the name of the destination file. If the second argument is not provided, the name of the output file will have the same name as the input file. The source file contains NMPSM code and the destination file is a coe file. Once the assembler has completed its task, it will either display a message saying it assembled the code successfully or display an error message with a description of the error and line number.

The assembler uses whitespaces as token separators in the source file. This means that one or more white spaces must appear between the commands and any arguments. For example, LOAD rx k is a valid command while LOAD rxk is not. All syntax for each command is given in the instruction set table. Semicolons are used as comments within the source file. Everything on a line after a semicolon will be ignored by the assembler. Instructions are case insensitive and can be either lowercase or uppercase or a combination of both. The assembler can understand numbers written in binary, decimal and hexadecimal. Binary numbers need to be preceded with a % symbol while hexadecimal numbers need to be preceded with a $ symbol. Decimal numbers do not need to be preceded with any symbol.

Assembler Directives

In addition to the processor commands listed in the instruction set table, the assembler also has a small set of assembler directives that can be used to alter its operation. The assembler directives are: .alias, .org, .size and line labels.

The .alias directive is used to provide human readable names to various RAM addresses. For example, the following code will substitute $11F into “regName”:

```
.alias regName $1FF
LOAD regName $0001
```

The .org directive moves the current address to which the next lines of code will be written in the .coe file. If the code is moved to an address that already has code in it or an address that exceeds the maximum size of the program, an error will occur and the assembler will not assemble the program.

The .size directive tells the assembler the maximum size of the program. The minimum size of a program is 1 instruction while the maximum size of a program is 65536 instructions. If no size directive is stated in the program, the size of the program defaults to 512 instructions. 512 instructions will fit into a single block RAM.

Labels are not technically assembler directives, but they are used by the assembler to make identifying functions easier. Labels must start with a letter and contain only numbers or letters and the underscore character (_). Any other characters will cause the assembler to report an error. Label names must be unique within the program and cannot be commands. Label names are case sensitive so the labels "Label" and "label" are unique labels. Only one label may appear on a line, but labels may appear on consecutive lines. Labels may be on a line of their own or may appear at the beginning of a line that contains a command. Labels must end with a colon.

Differences between Old Project and New Project

Faster System Clock

The original Nexys 2 board used a 50MHz system clock. The Basys board has a 100MHz system clock. This allows for a more powerful PPU.

Single crystal operation

The original project required two clock crystals: 50MHz system clock and a 25.175MHz VGA clock. The effects of meta-stability had to be taken into account and is discussed below.

MEMS microphone

The original project used an electret microphone. The old microphone PMOD was replaced by a MEMs microphone PMOD. The replacement required no modifications to the control hardware and was a direct replacement.

Upgraded PPU

The old PPU was only able to handle 64 sprites on the display at one time. Due to the increased clock frequency and more efficient coding techniques, 256 sprites are now able to be displayed at a time.

Block RAM Wrapper Bug

A problem was experienced when upgrading the project from the old Spartan 3 part to the new Artix 7 part. Several block RAM primitives were instantiated in the PPU. The macros used to instantiate the block RAMs have changed. Wrappers have been provided to automatically change the macros when they are encountered. The wrappers work for several different configurations of block RAM but there is one configuration where the wrapper does not work. The following macro does not translate to a new design properly:

RAMB16_S9_S36

The block RAM primitive had to be changed manually to the following:

BRAM_TDP_MACRO

UART Implementation

The old design did not have UART functionality. The new design has a UART as described previously in this report.

Improved Audio Sprite Software Routine

Due to the limited number of sprites the original PPU could display, the sprite audio demonstration was limited to 32 sprites. The new sprite audio demonstration has 128 sprites.

DCM to MMCM conversion

The old design used a DCM to provide clock signals to the various modules. The new chip uses an MMCM instead of a DCM. The IP core for the clock manager needed to be changed. As a result of the new MMCM, a BUFG also had to be implemented as the MMCM needed to be isolated from the system clock in order to be instantiated. Because of the enhanced operation of the MMCM, the separate VGA clock crystal was no longer necessary. A clock frequency of 25.170 was able to be generated by the MMCM. This is very close to the VGA specification of 25.175MHz.

Meta-Stability

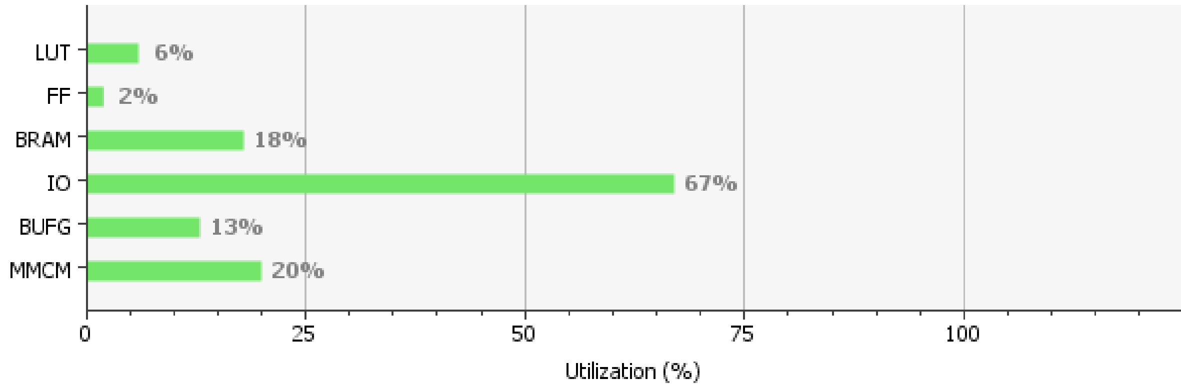
An interesting note about this project is that it was first implemented with two separate clock crystals. The first clock was the 50MHz system clock. The second was a 25.175 MHz VGA clock. The two crystals were uncorrelated to each other and would cause the processor to hang under certain timing instances. This was due to interrupt signals from the VGA controller (running at 25.175MHz) arriving at the processor (running at 50MHz) at a critical time. The solution to keeping the system stable was to add meta-stability hardening flip-flops in series with the interrupts. A flip-flop was also added in series with the reset line. The current project uses a single crystal and an MMCM to produce the required operating frequencies so meta-stability is no longer an issue with the interrupts but still a potential issue with the reset as it is controlled by a user accessible button.

Things Not Implemented

Two modules were under development for this project but were not included in the final design as there was not enough time to implement them properly. An I2C control module was created and verified to work in other projects but it was not working when applied to this system. The I2C module was going to control an alphanumeric display and show various messages. The I2C module appeared to be working properly when hooked up to an oscilloscope, but the display was not initializing properly. The control unit in the display was providing an ACK response to requests from the controller but no response was visible from the display.

A joystick control module was also created for the project. The control module was receiving a stream of digital SPI information from the joystick but the data was not being shifted properly so the joystick position was not being reported accurately.

Resource Usage



Resource	Utilization	Available	Utilization %
LUT	1273	20800	6.12
FF	760	41600	1.83
BRAM	9	50	18.00
IO	71	106	66.98
BUFG	4	32	12.50
MMCM	1	5	20.00

Timing Report

Setup

Worst Negative Slack (WNS):	0.285 ns
Total Negative Slack (TNS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	2147

Hold

Worst Hold Slack (WHS):	0.038 ns
Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	2147

Pulse Width

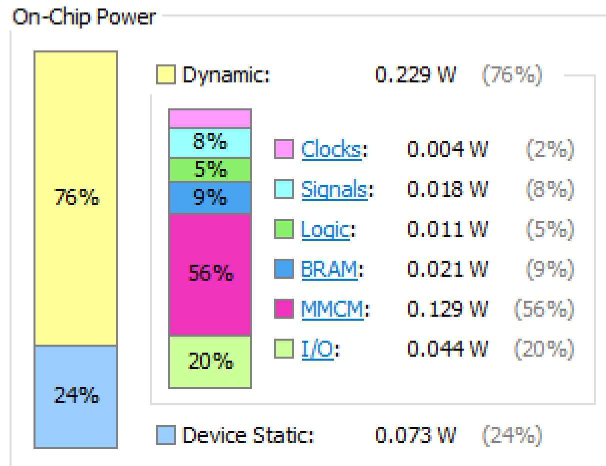
Worst Pulse Width Slack (WPWS):	3.000 ns
Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	753

All user specified timing constraints are met.

Power Report

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.301 W**
Junction Temperature:	**26.5 ℃**
Thermal Margin:	58.5 ℃ (11.6 W)
Effective ϑJA:	5.0 ℃/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low



Conclusion

Converting this project over from a Spartan 3 part provided a few challenges. The first was that the DCM had to be replaced with an MMCM. The replacement of the clock manager allowed for the VGA crystal to be removed from the design. Another challenge was introduced because of the non-functional block RAM wrapper and required a block RAM primitive to be replaced by hand. The faster clock frequency of the new system provided an opportunity to enhance the features of the PPU by quadrupling the number of sprites it can handle at any given time. Functionality of the overall system was enhanced by adding a UART to the design.