Creating an Emulator for Turbo Graphx Systems
A Handy Guide
Version 0.2

Warren Wilkinson
wagwilk@telusplanet.net

April 30, 2005

Abstract

This document is to serve as a reference for emulator writers desiring to create an emulator for the Turbo Graphx line of consoles (also commonly refered to as the PC-Engine line of consoles). System hardware, the instructions set, component parts, and their interactions are discussed at length. Software concerns relating to the creation of an PCE emulator are discussed in the second part of the document.

Contents

List of Tables

1 Introduction

This document is a work in progress. There are a number of things I don't know about the PCE family of game consoles; I only had the barebones Turbo Graphics 16. However I would like to support the entire family of consoles with one emulator, so any information about any of the systems is appreciated.

If you'd like to submit feedback, questions or comments email them me here: wagwilktelus-planet.net, Put OpenPCE in the subject line (so my spam filter doesn't eat your message).

Table 1: Status Register Bits

Bit Number	7	6	5	4	3	2	1	0
	S	V		B	D	I	Z	C

2 Description of Hardware

2.1 Main Processor (HuC6280)

The HuC6280 is an 8-bit microprocessor. Apparently it is similar in design to 6502 and 65C02 CPU's, and contains a 65C02 core with several additional instructions (WHAT ARE THEY?) and a few internal peripheral functions (SOME ARE: an interrupt controller, a MMU, a TIMER, a 8-bit parallel I/O port, and a PSG. ARE THERE MORE?) [David Michel, 2003]

There will be more descriptions here, probably will need to cite these guys 6502 docs -¿ [_Bnu, 1999] and 65C02 docs -¿ [Zophar, 2004].

2.1.1 Registers

ACCUMULATOR:
This is THE most important register in the microprocessor. Various machine language instructions allow you to copy the contents of a memory location into the accumulator, copy the contents of the accumulator into a memory location, modify the contents of the accumulator or some other register directly, without affecting any memory. And the accumulator is the only register that has instructions for performing math.[_Bnu, 1999]

THE X INDEX REGISTER:
This is a very important register. There are instructions for nearly all of the transformations you can make to the accumulator. But there are other instructions for things that only the X register can do. Various machine language instructions allow you to copy the contents of a memory location into the X register, copy the contents of the X register into a memory location, and modify the contents of the X, or some other register directly.[_Bnu, 1999]

THE Y INDEX REGISTER:
This is a very important register. There are instructions for nearly all of the transformations you can make to the accumulator, and the X register. But there are other instructions for things that only the Y register can do. Various machine language instructions allow you to copy the contents of a memory location into the Y register, copy the contents of the Y register into a memory location, and modify the contents of the Y, or some other register directly.[_Bnu, 1999]

THE STATUS REGISTER:
This register consists of eight ”flags” (a flag = something that indicates whether something has, or has not occurred). Bits of this register are altered depending on the result of arithmetic and logical operations. These bits are described below:
Bit 0 - C - Carry flag: this holds the carry out of the most significant bit in any arithmetic operation. In subtraction operations however, this flag is cleared - set to 0 - if a borrow is required, set to 1 - if no borrow is required. The carry flag is also used in shift and rotate logical operations.

Bit 1 - Z - Zero flag: this is set to 1 when any arithmetic or logical operation produces a zero result, and is set to 0 if the result is non-zero.

Bit 2 - I: this is an interrupt enable/disable flag. If it is set, interrupts are disabled. If it is cleared, interrupts are enabled.

Bit 3 - D: this is the decimal mode status flag. When set, and an Add with Carry or Subtract with Carry instruction is executed, the source values are treated as valid BCD (Binary Coded Decimal, eg. 0x00-0x99 = 0-99) numbers. The result generated is also a BCD number.

Bit 4 - B: this is set when a software interrupt (BRK instruction) is executed.

Bit 5: not used. Supposed to be logical 1 at all times. It might also be the 'T' bit, if so than its used to control whether or not a few instructions affect a memory location rather than the accumulator. [Robinson, 2005]

Bit 6 - V - Overflow flag: when an arithmetic operation produces a result too large to be represented in a byte, V is set.

Bit 7 - S - Sign flag: this is set if the result of an operation is negative, cleared if positive.

The most commonly used flags are C, Z, V, S.[_Bnu, 1999]

THE PROGRAM COUNTER:
This contains the address of the current machine language instruction being executed. Since the operating system is always "RUN"ning in the Commodore VIC-20 (or, for that matter, any computer), the program counter is always changing. It could only be stopped by halting the microprocessor in some way.[_Bnu, 1999]

THE STACK POINTER
This register contains the location of the first empty place on the stack. The stack is used for temporary storage by machine language pro-grams, and by the computer.[_Bnu, 1999]

2.1.2 Memory Management

The HuC6280 has a 64 KB logical address space and a 2 MB physical address space. To access this entire memory space, the HuC6280 uses a MMU (Memory Managment Unit) that splits the memory space in segment of 8 KB[1]. The logical address space is splitted as follow :

Table 2: Logical Memory range per page

Page	**Logical Address Space (hex)**	Base Register
0	0x0000 - 0x1FFF	MPR0
1	0x2000 - 0x3FFF	MPR1
2	0x4000 - 0x5FFF	MPR2
3	0x6000 - 0x7FFF	MPR3
4	0x8000 - 0x9FFF	MPR4
5	0xA000 - 0xBFFF	MPR5
6	0xC000 - 0xDFFF	MPR6
7	0xE000 - 0xFFFF	MPR7

Each logical 8 KB segment (or page) is associated to a 8-bit register (MPR0-7) that contains the index of the 8 KB segment (or bank) in physical memory to map in this page. Two special instructions are used to access these registers :

TAMi, transfer the content of the accumulator (A) into a MPR register (0 - 7).

TMAi, transfer a MPR register into the accumulator.[David Michel, 2003]

Table 3: Division of Address Pointers

Page Number	**Page Offset**
3 bits	13 bits

MORE INFORMATION IN THE MEMORY MAPPING DOCUMENT SHOULD BE PUT HERE.

[1]8K can be represented by 13 bits

2.1.3 Bank Maps

Table 4: Bank Maps

Address	**Description**
0x00-0x7F	ROM
0xF7	battery backed RAM
0xF8	work RAM
0xFF	hardware I/O page

You are free to map banks anywhere into the 8 pages, but a sort of standard has been defined, all the games seem to use it, so it was used in all the MagicKit's tools too. Here's how banks are mapped by default :

Table 5: Default Bank Mappings

Page	**Logical Address Space (hex)**
0	Bank 0xFF (I/O)
1	Bank 0xF8 (ram)
2	User Defined
3	
4	
5	
6	
7	Bank 0x00 (rom)

Note: After a reset, Bank 0x00 (rom) is automatically mapped into page 7.[David Michel, 2003]█

2.1.4 Timer

The TIMER base frequency is 6.992 KHz.

Table 6: Timer Counter Register

Address	**Read/Write**	**Bits,**	**Purpose**
0x0C00	R/W	bit 7:	(unused)
		bit 6-0:	7-bit down counter

Table 7: Timer Countrol Register

Address	**Read/Write**	**Bits,**	**Purpose**
0x0C01	/W	bit 7-1:	(unused)
		bit 0:	start/stop (0 = off, 1 = on)

Note: Addresses are only valid when I/O bank is mapped to page 0 (which is standard)

An interrupt is raised when the counter generates a carry, in other words, when the counter is to be decremented and its value is zero.[David Michel, 2003]

2.1.5 Gamepad IO Port

The Turbo Graphics featured a single IO port for a gamepad. The multitap extention (discussed in its own section) allows for multiple gamepads to be used with the system.

Table 8: Gamepad I/O port[David Michel, 2003]

Address	**Read/Write**	**Bits,**	**Purpose**
0x1000	/W	bit 7-2:	(unused)
		bit 1:	Gamepad CLR Line
		bit 0:	Gamepad SEL Line
0x1000	R	bit 7:	(unused)
		bit 6:	country (1 = JPN, 0 = USA)
		bit 5-4:	(unused)
		bit 3-0:	gamepad 4-bit data

Note: Addresses are only valid when I/O bank is mapped to page 0 (which is standard)

The gamepad is read in two stages to determine the condition of all the gamepad inputs. More information is in the gamepad section.

John Robinson mentions another bit is used if a CD-ROM is present.

```
gameport status register is missing at least one item. if the cd-rom is present one of the bi
```

[Robinson, 2005]

2.1.6 Interrupts

Table 9: Interrupt disable register

Address	Read/Write	Bits,	Purpose
0x1402	R/W	bit 7-3:	(unused)
		bit 2:	Timer
		bit 1:	IRQ1 (VDC)
		bit 0:	IRQ2 (external)

Note: Writing 1 to a bit disables its corrisponding interrupt, writing a 0 to a bit enables its corrisponding interrupt. Addresses are only valid when I/O bank is mapped to page 0 (which is standard)

Table 10: Interrupt status register

Address	Read/Write	Bits,	Purpose
0x1403	R	bit 7-3:	(unused)
		bit 2:	Timer
		bit 1:	IRQ1 (VDC)
		bit 0:	IRQ2 (external)

Note: A write to the Interrupt Status Register (table 10) acknowledges the internal timer interrupt. If you don't write to this register at the end of the timer interupt handler you will get infinite timer interrupts. table Addresses are only valid when I/O bank is mapped to page 0 (which is standard) [David Michel, 2003]

John Robinson mentions:

```
IRQ2 is shared between the external source (CDROM) and the BRK instruction. it pushes the add
```

[Robinson, 2005]

2.2 Huc6280 Instructions

I'd like to thank _Bnu[_Bnu, 1999] and Jens Ch. Restemeier[Restemeier, 1997] for their documentation on the 6502 and HuC6280 CPU's respectively. Much of this information is copied verbatim, especially from Restemeier's spectacular documentation. In fact, the great majority of this information is copied and pasted right out of Jens' html documentation without his permission – I haven't been able to contact Mr. Restemeier to ask if I may recopy his instruction set, so I'm hoping he will contact me (wagwilk@telusplanet.net).

How to Read Instructions:

For each instruction listed, a terse description of the function will be given, as well as a table of addressing modes, and a table listing the operations effects on the status flag. The definition should be easily read and understood so I'll proceed to explaining how the tables work, using the ADC (add with carry) command as an example:

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Immediate	ADC #Oper	0x69	2	2
Zero Page	ADC Oper	0x65	2	3
Zero Page,X	ADC Oper,X	0x75	2	4
Absolute	ADC Oper	0x6D	3	4
Absolute,X	ADC Oper,X	0x7D	3	4*
Absolute,Y	ADC Oper,Y	0x79	3	4*
Indirect	ADC (ZZ)	0x72	2	7
(Indirect,X)	ADC (Oper,X)	0x61	2	6
(Indirect),Y	ADC (Oper),Y	0x71	2	5*

The table shows all the possible addressing modes for the command. For each command it gives the Addressing Mode, which have been described previously, the syntax that is supported by mainstream PCE assemblers (note that Oper means the operand, or argument to the operation), the listed Opcode is the numeric value of the instruction (given as a hex value, so $0x69 = 105$ decimal), the number of bytes the instruction takes (in the case of immediate mode, 2, one byte for the instruction opcode and one more for the instruction operand), and finally the number of cpu cycles taken by the original hardware to execute the function.

Besides the addressing table, another table is given which shows how a command effects status flags:

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
?	?	0	-	-	-	?	?

The status register has been discussed previously, it is sufficient to know that:

- means that this flag will not be changed.

? means that this flag will be changed as appropriate for its function.

0 means that this flag will be set low.

1 means that this flag will be set high.

2.2.1 Exhaustive Index of Instructions

ADC Add Memory to Accumulator with carry

Function

Add the data located at the effective address specified by the operand to the set of contents of the accumulator. If the carry bit was set, then add an additional 1 before storing the result in the accumulator. This instruction takes one extra cycle to complete if the decimal mode flag is set.

For the entries with a asterix, add one to the value if a page boundary is crossed. Note that there is some disagreement on whether or not this page boundary makes any actual difference.
John Robinson mentions:

> ADC and SBC have decimal versions when the Decimal flag is set. This operates on the idea that the numbers are BCD numbers. This also affects the way flags are set. One flag isn't set in the BCD way, and then they set some flags only when the result is a normal BCD number or something.
>
> [Robinson, 2005], edited by the author

The decimal mode versions of ADC and SBC do not change the state of the overflow flag.[MacDonald, 2002]

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Immediate	ADC #Oper	0x69	2	2
Zero Page	ADC Oper	0x65	2	3
Zero Page,X	ADC Oper,X	0x75	2	4
Absolute	ADC Oper	0x6D	3	4
Absolute,X	ADC Oper,X	0x7D	3	4*
Absolute,Y	ADC Oper,Y	0x79	3	4*
Indirect	ADC (ZZ)	0x72	2	7
(Indirect,X)	ADC (Oper,X)	0x61	2	6
(Indirect),Y	ADC (Oper),Y	0x71	2	5*

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
?	?	0	-	-	-	?	?

AND "AND" Memory with Accumulator

Function

bitwise logical AND the data located at the effective address specified by the operand with the contents of the accumulator. Each bit in the accumulator is AND'ed with the corresponding bit in memory, with the result being stored in the respective accumulator bit.

For the entries with a asterix, add one to the value if a page boundary is crossed. Note that there is some disagreement on whether or not this page boundary makes any actual difference.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Immediate	AND #Oper	0x29	2	2
Zero Page	AND Oper	0x25	2	4[a]
Zero Page,X	AND Oper,X	0x35	2	4
Absolute	AND Oper	0x2D	3	5[b]
Absolute,X	AND Oper,X	0x3D	3	4*
Absolute,Y	AND Oper,Y	0x39	3	4*
Indirect[c]	AND (Oper)	0x32	2	7
(Indirect,X)	AND (Oper,X)	0x21	2	7[d]
(Indirect,Y)	AND (Oper),Y	0x31	2	7[e]

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
?	-	0	-	-	-	?	-

[a] _Bnu's documentation says 3 for this value
[b] _Bnu's documentation says 4 for this value
[c] Not in _Bnu's documentation
[d] _Bnu's documentation says 6 for this value
[e] _Bnu's documentation says 5 for this value

ASL Arithmetic Shift Left

Function

Shift the contents of the location specified by the operand left one bit. That is, bit one takes on the value originally found in bit zero, bit two takes the value originally in bit one, and so on; bit 7 is transferred into the carry flag; bit 0 is cleared. The arithmetic result of the operation is an unsigned multiplication by two.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Accumulator	ASL A	0x0A	1	2
Zero Page	ASL Oper	0x06	2	6[a]
Zero Page,X	ASL Oper,X	0x16	2	6
Absolute	ASL Oper	0x0E	3	7[b]
Absolute,X	ASL Oper,X	0x1E	3	7

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
?	-	0	-	-	-	?	?

[a] Bnu's documentation says 5 for this value
[b] Bnu's documentation says 6 for this value

2.2.2 Branching Instructions

A branch instruction that crosses a 256-byte or 8192-byte boundary does not take any additional cycles.[MacDonald, 2002]

BBRi Branch on Bit Reset

Function

The ith bit value in zero page memory location Oper is tested. If it is clear a branch is taken; if it is set, the instruction immediately following the three byte BBRi instruction is executed. If the branch is taken, a one-byte signed displacement (dd), fetched from the third byte of the instruction is added to the program counter. Once the branch address has been calculated, the result is loaded into the program counter, transferring control to that location. The allowable range of the displacement is -128 to +127 *from the instruction immediately following the branch.*

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Zero Page, Relative	BBR0 zz, dd	0x0F	3	6
Zero Page, Relative	BBR1 zz, dd	0x1F	3	6
Zero Page, Relative	BBR2 zz, dd	0x2F	3	6
Zero Page, Relative	BBR3 zz, dd	0x3F	3	6
Zero Page, Relative	BBR4 zz, dd	0x4F	3	6
Zero Page, Relative	BBR5 zz, dd	0x5F	3	6
Zero Page, Relative	BBR6 zz, dd	0x6F	3	6
Zero Page, Relative	BBR7 zz, dd	0x7F	3	6

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

BBSi Branch on Bit Set

Function

The ith bit value in zero page memory location Oper is tested. If it is set a branch is taken; if it is clear, the instruction immediately following the three byte BBSi instruction is executed. If the branch is taken, a one-byte signed displacement (dd), fetched from the third byte of the instruction is added to the program counter. Once the branch address has been calculated, the result is loaded into the program counter, transferring control to that location. The allowable range of the displacement is -128 to +127 *from the instruction immediately following the branch.*

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Zero Page, Relative	BBS0 zz, dd	0x8F	3	6
Zero Page, Relative	BBS1 zz, dd	0x9F	3	6
Zero Page, Relative	BBS2 zz, dd	0xAF	3	6
Zero Page, Relative	BBS3 zz, dd	0xBF	3	6
Zero Page, Relative	BBS4 zz, dd	0xCF	3	6
Zero Page, Relative	BBS5 zz, dd	0xDF	3	6
Zero Page, Relative	BBS6 zz, dd	0xEF	3	6
Zero Page, Relative	BBS7 zz, dd	0xFF	3	6

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

BCC Branch on Carry Clear

Function

The carry flag in the status register is tested. If it is clear, a branch is taken; if it is set, the instruction immediately following the two byte BCC instruction is executed. If the branch is taken, a one byte signed displacement, fetched from the second byte of the instruction is added to the program counter. Once the branch address has been calculated, the result is loaded into the program counter, transferring execution to that location. The allowable displacement range is -128 to +127 *from the instruction immediately following the branch.*

Note that BCC determines if the result of a comparison is less than; therefore, BCC is sometimes written as BLT (Branch Less Than). This opcode takes one extra cycle if the the branch is taken, and another extra cycle if a page boundary is crossed in taking the branch.

There is some dispute as to whether branching to another page actually incurs a greater penalty than branching onto the same memory page. Charles McDonald's document claims that there are no penalties to branching to different pages.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Relative	BCC hl	0x90	2	2

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

BCS Branch on Carry Set

Function

The carry flag in the status register is tested. If it is set, a branch is taken; if it is clear, the instruction immediately following the two byte BCS instruction is executed. If the branch is taken, a one byte signed displacement, fetched from the second byte of the instruction is added to the program counter. Once the branch address has been calculated, the result is loaded into the program counter, transferring execution to that location. The allowable displacement range is -128 to +127 *from the instruction immediately following the branch.*

Note that BCS determines if the result of a comparison is greater or equal than; therefore, BCS is sometimes written as BGE (Branch Greater than or Equal). This opcode takes one extra cycle if the the branch is taken, and another extra cycle if a page boundary is crossed in taking the branch.

There is some dispute as to whether branching to another page actually incurs a greater penalty than branching onto the same memory page. Charles McDonald's document claims that there are no penalties to branching to different pages.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Relative	BCS hl	0xB0	2	2

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

BEQ Branch on Equal

Function

The zero flag in the status register is tested. If it is set, a branch is taken; if it is clear, the instruction immediately following the two byte BEQ instruction is executed. If the branch is taken, a one byte signed displacement, fetched from the second byte of the instruction is added to the program counter. Once the branch address has been calculated, the result is loaded into the program counter, transferring execution to that location. The allowable displacement range is -128 to +127 *from the instruction immediately following the branch.*

This instruction gets its name for the reason that when two numbers are compared they are logically subtracted to set the status flags (however the result of this subtraction is not stored). If they are the same number the result will be zero, and thus the zero flag will be set. So by checking the zero flag in BEQ command, we are in effect checking to see that the two numbers were equal in the previous comparison.

Note that BCS determines if the result of a comparison is greater or equal than; therefore, BCS is sometimes written as BGE (Branch Greater than or Equal). This opcode takes one extra cycle if the the branch is taken, and another extra cycle if a page boundary is crossed in taking the branch.

There is some dispute as to whether branching to another page actually incurs a greater penalty than branching onto the same memory page. Charles McDonald's document claims that there are no penalties to branching to different pages.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Relative	BEQ hl	0xF0	2	2

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

BIT Test memory Bits again Accumulator

Function

BIT sets the status flags based on the result of two different operations. First, it sets or clears the N flag to reflect the value of the high bit (bit 7) of the data located at the effective address specified by the operand, and sets of clears the V flag to reflect the contents of the next-to-highest bit (bit 6) of the data addressed. Second it logically ANDs the data located at the effective address with the contents of the accumulator; it changes neither value, but sets the Z flag if the result is zero, or clears it if it is non-zero.

Addressing Modes & OpCodes

Addressing Mode	Syntax	Opcode	bytes	cycles
Immediate	BIT #nn	0x89 nn	2	2
Zero Page	BIT ZZ	0x24 ZZ	2	4[a]
Zero Page,X[a]	BIT ZZ,X	0x34 ZZ	2	4
Absolute	BIT hh ll	0x2C ll hh	3	4[b]
Absolute,X[a]	BIT hh ll,X	0x34 ll hh	3	5

Status Flags

N	V	T	B	D	I	Z	C
Mem7	Mem6	0	-	-	-	?	-

[a]Was 3 in _Bnu's documentation
[b]was 5 in _Bnu's documentation

BMI Branch on MInus

Function

The negative flag in the status register is tested. If it is set, meaning the high bit of the value which most recently affected the N flag was set, a branch is taken. Since numbers are often stored in two's complement, this instruction can be used to detect negative numbers. If it is clear, the instruction immediately following the two-byte BMI instruction is executed. If the branch is taken, a one-byte displacement, fetched from the third byte of the instruction, is added to the program counter. Once the branch address has been calculated, the result is loaded into the program counter, transferring code to that location. The allowable range of the displacement is -128 to +127 *from the instruction immediately following the branch.*

Addressing Modes & OpCodes

Addressing Mode	Syntax	Opcode	bytes	cycles
Relative	BMI hhll	0x30 rr	2	2

Status Flags

N	V	T	B	D	I	Z	C
-	-	0	-	-	-	-	-

BNE Branch on Not Equal

Function

The zero flag in the status register is tested. If it is clear, meaning the last value tested (which affected the zero flag) was zero, a branch is taken; if it is set, meaning the value tested was non-zero, the instruction immediately following the two-byte BNE instruction is executed. If the branch is taken, a one-byte displacement, fetched from the third byte of the instruction, is added to the program counter. Once the branch address has been calculated, the result is loaded into the program counter, transferring code to that location. The allowable range of the displacement is -128 to +127 *from the instruction immediately following the branch.*

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Relative	BNE hhll	0xD0 rr	2	2

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

BPL Branch on PLus

Function

The negative flag in the status register is tested. If it is clear, meaning the high bit of the value which recently affected the N flag was cleared, a branch is taken; if it is set, the instruction immediately following the two-byte BPL instruction is executed. If the branch is taken, a one-byte displacement, fetched from the third byte of the instruction, is added to the program counter. Once the branch address has been calculated, the result is loaded into the program counter, transferring code to that location. The allowable range of the displacement is -128 to +127 *from the instruction immediately following the branch.*

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Relative	BPL hhll	0x10 rr	2	2

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

BRK Force Break

Function

Forces a software interrupt. BRK is unaffected by the I interrupt disable flag. Although BRK is a one-byte instruction, the program counter (which is pushed onto the stack by the instruction) is incremented by two; this lets you follow the break instruction with a one-byte signature byte indicating which break caused the interrupt. Be sure to pad BRK with a single byte to allow an RTI (return from interrupt) instruction to execute correctly. Multiple actions are invoked on a BRK. The program counter is incremented by 2. The high and low bytes of the program counter are pushed onto the stack in order, followed by the status register (P). The program counter is then loaded with the break vector stored at absolute address $00FFF6-$00FFF7. (Remember, the high byte is stored in $00FFF7 and the low byte is stored in $00FFF6.) The decimal flag D is cleared, and the I flag is set (to disable hardware IRQ interrupts) after a break is executed. Additionally, the break flag B in the status register value pushed onto the stack is set.

John Robinson mentions:

> The break flag is modified when this instruction is hit; how exactly it's modified depends upon whose document you read. Most emulators assume that the BRK flag is always 0 until the BRK instruction is executed, then it's set to 1. Charles McDonald's [document] says it's always set to 1 except after the BRK instruction. I don't think this matters because no game I know of actually hits the BRK instruction and I don't think any check the status of it's flag. I follow Charle's way, as his document has been the most up to date on all information, that i've seen.
>
> [Robinson, 2005], edited by the author

pushes the return address plus one to the stack; the next byte after the BRK instruction is always skipped.[MacDonald, 2002]

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	BRK	0x00	1	8 [a]

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	1	0	1	-	-

[a] Bnu's documentation says 7 for this value

BSR Branch to SubRoutine

Function

Similar to the Jump to Subroutine (JSR) instruction, Branch to Subroutine allows execution of a subroutine. However, the offset is specified in relative mode instead of as an absolute address. This saves a byte, but takes one more clock cycle than JSR, so its use is discouraged. The current program counter is pushed onto the stack. A one-byte signed displacement, fetched from the second byte of the instruction, is added to the program counter. Once the subroutine address has been calculated, the result is loaded into the program counter, transferring control to that location. The allowable range of the displacement is -128 to +127 from the instruction immediately following the BSR . This opcode takes one extra cycle if a page boundary is crossed in calling the subroutine.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Relative	BSR hhll	0x44 rr	2	8

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

BVC Branch on overflow Clear

Function

The overflow flag V in the status register is tested. If it is clear, a branch is taken; if it is set, the instruction immediately following the two-byte BVC instruction is executed. If the branch is taken, a one-byte signed displacement, fetched from the third byte of the instruction, is added to the program counter. Once the branch address has been calculated, the result is loaded into the program counter, transferring control to that location. The allowable range of the displacement is -128 to +127 *from the instruction immediately following the branch.*

BVC is almost exclusively used to check that a two's complement arithmetic calculation has not overflowed.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Relative	BVC hhll	0x50 rr	2	2

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

BVS Branch on overflow Set

Function

The overflow flag V in the status register is tested. If it is set, a branch is taken; if it is clear, the instruction immediately following the two-byte BVS instruction is executed. If the branch is taken, a one-byte signed displacement, fetched from the third byte of the instruction, is added to the program counter. Once the branch address has been calculated, the result is loaded into the program counter, transferring control to that location. The allowable range of the displacement is -128 to +127 *from the instruction immediately following the branch.*

BVS is almost exclusively used to check that a two's complement arithmetic calculation has overflowed.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Relative	BVS hhll	0x70 rr	2	2

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

2.2.3 Clear Instructions

CLA CLear Accumulator

Function

The Accumulator is set to 0x00.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	CLA	0x62	1	2

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

CLC CLear Carry flag

Function

The Clear flag in the status register is set to 0.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	CLC	0x18	1	2

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	0

CLD CLear Decimal flag

Function
The Decimal flag in the status register is set to 0.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	CLD	0xD8	1	2

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	0	-	-	-

CLI CLear Interrupt flag

Function
The Interrupt flag in the status register is set to 0.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	CLI	0x58	1	2

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	0	-	-

CLV CLear oVerflow flag

Function
The overflow flag in the status register is set to 0.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	CLV	0xB8	1	2

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	0	0	-	-	-	-	-

CLY CLear Y register

Function

The Y register is set to 0x00.

Addressing Modes & OpCodes

Addressing Mode	Syntax	Opcode	bytes	cycles
Implied	CLY	0xC2	1	2

Status Flags

N	V	T	B	D	I	Z	C
-	-	0	-	-	-	-	-

CLX CLear X register

Function

The X register is set to 0x00.

Addressing Modes & OpCodes

Addressing Mode	Syntax	Opcode	bytes	cycles
Implied	CLX	0x82	1	2

Status Flags

N	V	T	B	D	I	Z	C
-	-	0	-	-	-	-	-

2.2.4 Compare Instructions

CMP Compare Memory and Accumulator

Function

Subtract the data located at the effective address specified by the operand from the contents of the accumulator, setting the carry, zero, and negative flags based on the result, but without altering the contents of either the memory location or the accumulator. The comparison is of unsigned binary values only (decimal mode is ignored), and the result is not saved.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Immediate	CMP #nn	0xC9 nn	2	2
Zero Page	CMP ZZ	0xC5 ZZ	2	4 [a]
Zero Page,X	CMP ZZ,X	0xD5 ZZ	2	4
Absolute	CMP hhll	0xCD ll hh	3	5[b]
Absolute,X	CMP hhll,X	0xDD ll hh	3	5[b]
Absolute,Y	CMP hhll,Y	0xD9 ll hh	3	5[b]
Indirect	CMP (ZZ)	0xD2	2	7
(Indirect,X)	CMP (ZZ,X)	0xC1 ZZ	2	7 [c]
(Indirect),Y	CMP (ZZ),Y	0xD1 ZZ	2	7 [d]

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
?	-	0	-	-	-	?	?

[a]was 3 in _Bnu's documentation
[b]was 4 in _Bnu's documentation
[c]was 6 in _Bnu's documentation
[d]was 5 in _Bnu's documentation

CPX Compare Memory and Index X

Function

Subtract the data located at the effective address specified by the operand from the contents of the X register, setting the carry, zero, and negative flags based on the result, but without altering the contents of either the memory location or the accumulator. The comparison is of unsigned binary values only (decimal mode is ignored), and the result is not saved.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Immediate	CPX #nn	0xE0 nn	2	2
Zero Page	CPX ZZ	0xE4 ZZ	2	4[a]
Absolute	CPX hhll	0xEC ll hh	3	5[b]

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
?	-	0	-	-	-	?	?

[a]was 3 in _Bnu's documentation
[b]was 4 in _Bnu's documentation

CPY Compare Memory and Index Y

Function

Subtract the data located at the effective address specified by the operand from the contents of the Y register, setting the carry, zero, and negative flags based on the result, but without altering the contents of either the memory location or the accumulator. The comparison is of unsigned binary values only (decimal mode is ignored), and the result is not saved.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Immediate	CPY #nn	0xC0 nn	2	2
Zero Page	CPY ZZ	0xC4 ZZ	2	4[a]
Absolute	CPY hh ll	0xCC ll hh	3	5[b]

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
?	-	0	-	-	-	?	?

[a]was 3 in _Bnu's documentation
[b]was 4 in _Bnu's documentation

2.2.5 Speed Instructions

The CSL and CSH instructions change the CPU's clock speed. CSL selects low speed mode which is 1.78 MHz, CSH selects high speed mode which is 7.16 MHz. On power-up the CPU is in low speed mode.

CSH and CSL take 3 cycles each, but that was tested with the CPU already set to the respective clock speed. It currently isn't known if either instruction takes more or less time when switching between different speeds.[MacDonald, 2002]

CSH Change Speed High

Function

Sets the HuC6280 to "high speed", or normal speed mode. The only use for this instruction is after a system reset, to ensure that the processor is in its high speed mode.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	CSH	0xD4	1	3

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

CSL Change Speed Low

Function

Sets the HuC6280 to low speed. The only use for this instruction appears to be for the US "country check" code; it does not appear anywhere else in HuC6280 code. Its use is discouraged.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	CSL	0x54	1	3

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

2.2.6 Decrement Instructions

DEC Decrement Memory

Function

Decrement by one the contents of the location specified by the operand (subtract one from the value). DEC neither affects nor is affected by the carry flag.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Zero Page	DEC ZZ	0xC6 ZZ	2	6[a]
Zero Page,X	DEC ZZ,X	0xD6 ZZ	2	6
Absolute	DEC hhll	0xCE ll hh	3	7[b]
Absolute,X	DEC hhll,X	0xDE ll hh	3	7
Accumulator	DEC A	0x3A	1	2

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
?	-	0	-	-	-	?	-

[a]was 5 in _Bnu's documentation
[b]was 6 in _Bnu's documentation

DEX Decrement X

Function

Decrement by one the contents of the X register (subtract one from the value). DEX neither affects nor is affected by the carry flag.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	DEX	0xCA	1	2

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
?	-	0	-	-	-	?	-

DEY Decrement Y

Function

Decrement by one the contents of the Y register (subtract one from the value). DEY neither affects nor is affected by the carry flag.

Addressing Modes & OpCodes

Addressing Mode	Syntax	Opcode	bytes	cycles
Implied	DEY	0x88	1	2

Status Flags

N	V	T	B	D	I	Z	C
?	-	0	-	-	-	?	-

EOR Exclusive-OR memory with accumulator

Function

Bitwise logical Exclusive OR (XOR) the data located at the effective address specified by the operand with the contents of the accumulator. Each bit in the accumulator is XORed with the corresponding bit in memory, with the result being stored in the respective accumulator bit.

There is some confusion about the Hex Value for the instruction when it is in Absolute mode. _Bnu's documentation says 0x40 is EOR in the absolute varient, and 0x4D is the RTI instruction in implied mode. Jens Restemeier's instructions say the opposite: 0x4D is the EOR absolute varient, and 0x40 is the RTI instruction implied varient.

Addressing Modes & OpCodes

Addressing Mode	Syntax	Opcode	bytes	cycles
Immediate	EOR #nn	0x49 nn	2	2
Zero Page	EOR ZZ	0x45 ZZ	2	4[a]
Zero Page,X	EOR ZZ,X	0x55 ZZ	2	4
Absolute	EOR hhll	0x40 ll hh	3	5[b]
Absolute,X	EOR hhll,X	0x5D ll hh	3	5[b]
Absolute,Y	EOR hhll,Y	0x59 ll hh	3	5[b]
Indirect	EOR (ZZ)	0x52 ZZ	2	7
(Indirect,X)	EOR (ZZ,X)	0x41 ZZ	2	7[c]
(Indirect),Y	EOR (ZZ),Y	0x51 ZZ	2	7[d]

Status Flags

N	V	T	B	D	I	Z	C
?	-	0	-	-	-	?	-

[a]was 3 in _Bnu's documentation

[b]was 4 in _Bnu's documentation

[c]was 6 in _Bnu's documentation

[d]was 5 in _Bnu's documentation, unless a page boundary was crossed, in which case it was 6

2.2.7 Increment Instructions

INC INCrement memory by one

Function

Increments contents of the location specified by the operand (add one to the value). INC neither affects nor is affected by the carry flag.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Zero Page	INC ZZ	0xE6 ZZ	2	6[a]
Zero Page,X	INC ZZ,X	0xF6 ZZ	2	6
Absolute	INC hhll	0xEE ll hh	3	7[b]
Absolute,X	INC hhll,X	0xFE ll hh	3	7
Accumulator	INC A	0x1A	1	2

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
?	-	0	-	-	-	?	-

[a] was 5 in _Bnu's documentation
[b] was 6 in _Bnu's documentation

INX INcrement X by one

Function

Increment by one contents of the X register (add one to the value). INX neither affects nor is affected by the carry flag.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	INX	0xE8	1	2

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
?	-	0	-	-	-	?	-

INY INcrement Y by one

Function

Increment by one contents of the Y register (add one to the value). INY neither affects nor is affected by the carry flag.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	INY	0xC8	1	2

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
?	-	0	-	-	-	?	-

2.2.8 Jump Instructions

An indirect JMP instruction with the low byte of the address set to $FF will correctly read the high byte at the next address, instead of wrapping to address 0 like the 6502 does. (so jmp [$FEFF] reads the MSB from address $FF00, not $FE00)[MacDonald, 2002]

JMP JuMP to new location

Function

Transfer control to the address specified by the operand field. The program counter is loaded with the target address.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Absolute	JMP hhll	0x4C ll hh	3	4[a]
Absolute Indirect	JMP (hhll)	0x6C ll hh	3	7[b]
Absolute Indirect X	JMP hhll, X	0x7C ll hh	3	7

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

[a]was 3 in _Bnu's documentation
[b]was 5 in _Bnu's documentation

JSR Jump to SubRoutine

Function

Transfer control to the subroutine at the location specified by the operand, after first pushing the current program counter value onto the stack as a return address. The value of the PC which is pushed onto the stack is the location of the last (third) byte of the JSR instruction, not the address of the next opcode.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Absolute	JSR hh ll	0x20 ll hh	3	7[a]

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

[a]was 6 in _Bnu's documentation

2.2.9 Load Instructions

LDA LoaD Accumulator from memory

Function

Load the accumulator with the data located at the effective address specified by the operand.

Addressing Modes & OpCodes

Addressing Mode	Syntax	Opcode	bytes	cycles
Immediate	LDA #nn	0xA9 nn	2	2
Zero Page	LDA ZZ	0xA5 ZZ	2	4[a]
Zero Page,X	LDA ZZ,X	0xB5 ZZ	2	4
Absolute	LDA hhll	0xAD ll hh	3	5[b]
Absolute,X	LDA hhll,X	0xBD ll hh	3	5[b]
Absolute,Y	LDA hhll,Y	0xB9 ll hh	3	5[b]
Indirect	LDA (ZZ)	0xB2 ZZ	2	7
(Indirect,X)	LDA (Oper,X)	0xA1 ZZ	2	7[c]
(Indirect),Y	LDA (Oper),Y	0xB1 ZZ	2	7[d]

Status Flags

N	V	T	B	D	I	Z	C
?	-	0	-	-	-	?	-

[a]was 3 in _Bnu's documentation
[b]was 4 in _Bnu's documentation
[c]was 6 in _Bnu's documentation
[d]was 5 in _Bnu's documentation

LDX LoaD index X with memory

Function

Load the X register with the data located at the effective address specified by the operand.

Addressing Modes & OpCodes

Addressing Mode	Syntax	Opcode	bytes	cycles
Immediate	LDX #nn	0xA2 nn	2	2
Zero Page	LDX ZZ	0xA6 ZZ	2	4[a]
Zero Page,Y	LDX ZZ,Y	0xB6 ZZ	2	4
Absolute	LDX hhll	0xAE ll hh	3	5[b]
Absolute,Y	LDX hhll,Y	0xBE ll hh	3	5[c]

Status Flags

N	V	T	B	D	I	Z	C
?	-	0	-	-	-	?	-

[a]was 3 in _Bnu's documentation
[b]was 4 in _Bnu's documentation
[c]was 4 in _Bnu's documentation, unless a page boundary was crossed, in which case it was 5

LDY LoaD index Y with memory

Function

Load the Y register with the data located at the effective address specified by the operand.

Addressing Modes & OpCodes

Addressing Mode	Syntax	Opcode	bytes	cycles
Immediate	LDX #nn	0xA0 nn	2	2
Zero Page	LDX ZZ	0xA4 ZZ	2	4[a]
Zero Page,X	LDX ZZ,X	0xB4 ZZ	2	4
Absolute	LDX hhll	0xAC ll hh	3	5[b]
Absolute,X	LDX hhll,X	0xBC ll hh	3	5[c]

Status Flags

N	V	T	B	D	I	Z	C
?	-	0	-	-	-	?	-

[a]was 3 in _Bnu's documentation
[b]was 4 in _Bnu's documentation
[c]was 4 in _Bnu's documentation, unless a page boundary was crossed, in which case it was 5

LSR Logical Shift Right

Function

Logical shift the contents of the location specified by the operand right one bit. That is, bit zero takes on the value originally found in bit one, bit one takes the value originally in bit two, and so on; bit 7 is cleared; bit 0 is transferred into the carry flag. The arithmetic result of the operation is an unsigned division by two.

Addressing Modes & OpCodes

Addressing Mode	Syntax	Opcode	bytes	cycles
Accumulator	LSR A	0x4A	1	2
Zero Page	LSR ZZ	0x46 ZZ	2	6[a]
Zero Page,X	LSR ZZ,X	0x56 ZZ	2	6
Absolute	LSR hhll	0x4E ll hh	3	7[b]
Absolute,X	LSR hhll,X	0x5E ll hh	3	7

Status Flags

N	V	T	B	D	I	Z	C
0	-	0	-	-	-	?	?

[a]was 5 in _Bnu's documentation
[b]was 6 in _Bnu's documentation

NOP No OPeration

Function

NOP performs no action, and is often used for timing loops or temporarily removing certain instructions. A NOP instruction consumes two cycles.

Addressing Modes & OpCodes

Addressing Mode	Syntax	Opcode	bytes	cycles
Implied	NOP	0xEA	1	2

Status Flags

N	V	T	B	D	I	Z	C
-	-	0	-	-	-	-	-

ORA OR memory with Accumulator

Function

Bitwise logical OR the data located at the effective address specified by the operand with the contents of the accumulator. Each bit in the accumulator is ORed with the corresponding bit in memory, with the result being stored in the respective accumulator bit.

Addressing Modes & OpCodes

Addressing Mode	Syntax	Opcode	bytes	cycles
Immediate	ORA #nn	0x09 nn	2	2
Zero Page	ORA ZZ	0x05 ZZ	2	4 [a]
Zero Page,X	ORA ZZ,X	0x15 ZZ	2	4
Absolute	ORA hhll	0x0D ll hh	3	4[b]
Absolute,X	ORA hhll,X	0x1D ll hh	3	5[c]
Absolute,Y	ORA hhll,Y	0x19 ll hh	3	5[c]
Indirect	ORA (ZZ)	0x12 ZZ	2	7
(Indirect,X)	ORA (ZZ,X)	0x01 ZZ	2	7[d]
(Indirect),Y	ORA (ZZ),Y	0x11 ZZ	2	7[e]

Status Flags

N	V	T	B	D	I	Z	C
?	-	0	-	-	-	?	-

[a]was 3 in _Bnu's documentation

[b]was 4 in _Bnu's documentation

[c]was 4 in _Bnu's documentation, unless a page boundary was crossed, in which case the value was 5

[d]was 6 in _Bnu's documentation

[e]was 5 in _Bnu's documentation

2.2.10 Push and Pull Instructions

PHA PusH Accumulator on stack

Function

Pushes the value in the accumulator onto the stack.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	PHA	0x48	1	3

Status Flags

N	V	T	B	D	I	Z	C
-	-	0	-	-	-	-	-

PHP PusH Processor Status on stack

Function

Push the process status register P onto the stack.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	PHP	0x08	1	3

Status Flags

N	V	T	B	D	I	Z	C
-	-	0	-	-	-	-	-

PHX PusH X register onto stack

Function

Push the X register onto the stack.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	PHX	0xDA	1	3

Status Flags

N	V	T	B	D	I	Z	C
-	-	0	-	-	-	-	-

PHY PusH Y register onto stack

Function

Push the Y register onto the stack.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	PHY	0x5A	1	3

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
?	-	0	-	-	-	-	-

PLA PuLl Accumulator from stack

Function

Pull the value on the top of the stack into the accumulator. The previous contents of the accumulator are destroyed.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	PLA	0x68	1	4

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
?	-	0	-	-	-	?	-

PLP PuLl Processor Status from stack

Function

Pull the value on the top of the stack into the processor status register P. The previous contents of the status register are destroyed.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	PLP	0x28	1	4

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
?	?	?	?	?	?	?	?

PLX PuLl X register from stack

Function

Pull the value on the top of the stack into the X register. The previous contents of the X register are destroyed.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	PLX	0xFA	1	4

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

PLY PuLl Y register from stack

Function

Pull the value on the top of the stack into the Y register. The previous contents of the Y register are destroyed.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	PLY	0x7A	1	4

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

ROL ROtate one bit Left (memory or accumulator)

Function

Rotate the contents of the location specified by the operand left one bit. That is, bit one takes on the value originally found in bit zero, bit two takes the value originally in bit one, and so on; bit zero takes on the value in the carry flag; bit seven is transferred into the carry.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Accumulator	ROL A	0x2A	1	2
Zero Page	ROL ZZ	0x26 ZZ	2	6[a]
Zero Page,X	ROL ZZ,X	0x36 ZZ	2	6
Absolute	ROL hhll	0x2E ll hh	3	7[b]
Absolute,X	ROL hhll,X	0x3E ll hh	3	7

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
?	-	0	-	-	-	?	?

[a]was 5 in _Bnu's documentation
[b]was 6 in _Bnu's documentation

ROR ROtate one bit Right (memory or accumulator)

Function

Rotate the contents of the location specified by the operand right one bit. That is, bit zero takes on the value originally found in bit one, bit one takes the value originally in bit two, and so on; bit seven takes on the value in the carry flag; bit zero is transferred into the carry.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Accumulator	ROR A	0x6A	1	2
Zero Page	ROR ZZ	0x66 ZZ	2	6[a]
Zero Page,X	ROR ZZ,X	0x76 ZZ	2	6
Absolute	ROR hhll	0x6E ll hh	3	7[b]
Absolute,X	ROR hhll,X	0x7E ll hh	3	7

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
?	-	0	-	-	-	?	?

[a]was 5 in _Bnu's documentation
[b]was 6 in _Bnu's documentation

2.2.11 Return Instructions

RTI ReTurn from Interrupt

Function

Pull the status register and the program counter from the stack in order. Normally used to return from an interrupt call (such as BRK), this instruction can also be used to pull the status register P, and the program counter low and high bytes from the stack into the P and program counter registers.

There is some confusion about the Hex Value for the instruction when it is in Absolute mode. _Bnu's documentation says 0x40 is EOR in the absolute varient, and 0x4D is the RTI instruction in implied mode. Jens Restemeier's instructions say the opposite: 0x4D is the EOR absolute varient, and 0x40 is the RTI instruction implied varient.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	RTI	0x4D	1	7[a]

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
?	?	?	?	?	?	?	?

[a]was 6 in _Bnu's documentation

RTS Return from subroutine

Function

Pull the program counter from the stack, incrementing the 16-bit value by one before loading the program counter with it. The low byte of the program counter is pulled from the stack first, followed by the high byte.

Addressing Modes & OpCodes

Addressing Mode	Syntax	Opcode	bytes	cycles
Implied	RTS	0x60	1	7[a]

Status Flags

N	V	T	B	D	I	Z	C
-	-	0	-	-	-	-	-

[a]was 6 in _Bnu's documentation

2.2.12 Swap Instructions

SAX Swap Accumulator and X register

Function

The values of the accumulator and the X Register are swapped.

Addressing Modes & OpCodes

Addressing Mode	Syntax	Opcode	bytes	cycles
Implied	SAX	0x22	1	3

Status Flags

N	V	T	B	D	I	Z	C
-	-	0	-	-	-	-	-

SAY Swap Accumulator and Y register

Function

The values of the accumulator and the Y Register are swapped.

Addressing Modes & OpCodes

Addressing Mode	Syntax	Opcode	bytes	cycles
Implied	SAY	0x42	1	3

Status Flags

N	V	T	B	D	I	Z	C
-	-	0	-	-	-	-	-

SBC Swap Subtract from accumulator (with borrow)

Function

Subtract the data located at the effective address specified by the operand from the contents of the accumulator. Subtract one more from the result if the carry flag is set, and store the final result in the accumulator. This opcode takes one extra cycle if the decimal mode flag D is set.

Accumulator - Memory - Carry = Result (Stored in Accumulator)

John Robinson mentions:

> ADC and SBC have decimal versions when the Decimal flag is set. This operates on the idea that the numbers are BCD numbers. This also affects the way flags are set. One flag isn't set in the BCD way, and then they set some flags only when the result is a normal BCD number or something.
>
> [Robinson, 2005], edited by the author

The decimal mode versions of ADC and SBC do not change the state of the overflow flag.[MacDonald, 2002]

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Immediate	SBC #nn	0xE9 nn	2	2
Zero Page	SBC ZZ	0xE5 ZZ	2	4[a]
Zero Page,X	SBC ZZ,X	0xF5 ZZ	2	4
Absolute	SBC hhll	0xED ll hh	3	5[b]
Absolute,X	SBC hhll,X	0xFD ll hh	3	5[c]
Absolute,Y	SBC hhll,Y	0xF9 ll hh	3	5[c]
Indirect	SBC (ZZ)	0xF2 ZZ	2	7
(Indirect,X)	SBC (ZZ,X)	0xE1 ZZ	2	7[d]
(Indirect),Y	SBC (ZZ),Y	0xF1 ZZ	2	7[e]

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
?	?	0	-	-	-	?	?

[a]was 3 in _Bnu's documentation
[b]was 4 in _Bnu's documentation
[c]was 4 in _Bnu's documentation, unless a page boundary was crossed where it was 5
[d]was 6 in _Bnu's documentation
[e]was 5 in _Bnu's documentation

2.2.13 Set Instructions

SEC Set Carry Flag

Function

The carry flag C in the status register is set to 1.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	SEC	0x38	1	2

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	1

SED Set Decimal Mode Flag

Function

The decimal mode flag D in the status register is set to 1. This enables BCD arithmetic.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	SED	0xF8	1	2

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	1	-	-	-

SEI Set interrupt disable flag

Function

The interrupt disable flag I in the status register is set to 1. This disables hardware interrupt processing.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	SEI	0x78	1	2

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	1	-	-

SET Set T flag

Function

The T flag in the status register is set to 1. The T flag is called the ”Memory Operation Flag;”, when this flag is set all the instructions that normally use the A register act differently, I don’t know exactly if all the instructions are affected but I’m sure for AND, EOR, OR & ADC. In place of using the A register the instruction use the memory location in ZP pointed by the X register, so for example if you use SET followed by ADC #10, the CPU will do ZP[X] = ZP[X] + 10.

Addressing Modes & OpCodes

Addressing Mode	Syntax	Opcode	bytes	cycles
Implied	SET	0xF4	1	2

Status Flags

N	V	T	B	D	I	Z	C
-	-	1	-	-	-	-	-

2.2.14 Store HuC6270 functions (STi)

ST0, ST1, ST2 write immediate data directly to the VDC (at physical addresses 1FE000−1FE003), the address is not translated through the CPUs memory mapping hardware.[MacDonald, 2002]

ST0 Store HuC6270 No. 0

Function

The immediate argument is stored in the HuC6270’s address register. This command is equivalent to storing the immediate argument in $1FE000. The HuC6270 “No. 0” register is also known as the HuC6270 Address/Status Register; more information is available in the HuC6270 summary. According to the Develo Book , this operation sets /CE7, A1, and A0 to logical LOW.

Addressing Modes & OpCodes

Addressing Mode	Syntax	Opcode	bytes	cycles
Implied	ST0 #nn	0x03 nn	2	4

Status Flags

N	V	T	B	D	I	Z	C
-	-	0	-	-	-	-	-

ST1 Store HuC6270 No. 1

Function

The immediate argument is stored in the HuC6270's low data register. This command is equivalent to storing the immediate argument in $1FE002. The HuC6270 "No. 1" register is also known as the HuC6270 Low Data Register; more information is available in the HuC6270 summary. According to the Develo Book , this operation sets /CE7 and A0 to logical LOW, while setting A1 to logical HIGH.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	ST1 #nn	0x13 nn	2	4

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

ST2 Store HuC6270 No. 2

Function

The immediate argument is stored in the HuC6270's high data register. This command is equivalent to storing the immediate argument in $1FE003. The HuC6270 "No. 2" register is also known as the HuC6270 High Data Register; more information is available in the HuC6270 summary. According to the Develo Book , this operation sets /CE7 to logical LOW, while setting A0 and A1 to logical HIGH.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	ST2 #nn	0x23 nn	2	4

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

SMBi Set Memory Bit i

Function

Set the specified bit in the zero page memory location specified in the operand. The bit to clear is specified by a number concatenated to the end of the mnemonic, resulting in 8 distinct Opcodes.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Zero page	SMB0 ZZ	0x87 ZZ	2	7
Zero page	SMB1 ZZ	0x97 ZZ	2	7
Zero page	SMB2 ZZ	0xA7 ZZ	2	7
Zero page	SMB3 ZZ	0xB7 ZZ	2	7
Zero page	SMB4 ZZ	0xC7 ZZ	2	7
Zero page	SMB5 ZZ	0xD7 ZZ	2	7
Zero page	SMB6 ZZ	0xE7 ZZ	2	7
Zero page	SMB7 ZZ	0xF7 ZZ	2	7

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

STA STore Accumulator in memory

Function

Stores the value in the accumulator to the effective address specified by the operand.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Zero Page	STA ZZ	0x85 ZZ	2	4[a]
Zero Page,X	STA ZZ,X	0x95 ZZ	2	4
Absolute	STA hhll	0x8D ll hh	3	5[b]
Absolute,X	STA hhll,X	0x9D ll hh	3	5
Absolute,Y	STA hhll,Y	0x99 ll hh	3	5
Indirect	STA (ZZ)	0x92 ZZ	2	7[c]
(Indirect,X)	STA (ZZ,X)	0x81 ZZ	2	7[c]
(Indirect),Y	STA (ZZ),Y	0x91 ZZ	2	7[c]

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

[a] was 3 in _Bnu's documentation
[b] was 4 in _Bnu's documentation
[c] was 6 in _Bnu's documentation

STX STore X register in memory

Function

Store the value in the X register to the effective address specified by the operand.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Zero Page	STX ZZ	0x86 ZZ	2	4[a]
Zero Page,Y	STX ZZ,Y	0x96 ZZ	2	4
Absolute	STX hh ll	0x8E ll hh	3	5[b]

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

[a]was 3 in _Bnu's documentation
[b]was 4 in _Bnu's documentation

STY STore Y register in memory

Function

Store the value in the Y register to the effective address specified by the operand.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Zero Page	STY ZZ	0x84 ZZ	2	4[a]
Zero Page,Y	STY ZZ,Y	0x94 ZZ	2	4
Absolute	STY hh ll	0x8C ll hh	3	5[b]

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

[a]was 3 in _Bnu's documentation
[b]was 4 in _Bnu's documentation

STZ STore Zero in memory

Function

Store the value 0x00 to the effective address specified by the operand. This is useful for initialising memory.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Zero Page	STZ ZZ	0x64 ZZ	2	4
Zero Page, X	STZ ZZ, X	0x74 ZZ	2	4
Absolute	STZ hhll	0x9C ll hh	3	5
Absolute, X	STZ hhll, X	0x9E ll hh	3	5

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

SXY Swap X and Y registers

Function

Swaps the values stored in the X and Y registers.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	SXY	0x02	1	3

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

2.2.15 Block Transfer Functions

- Block transfer instructions push Y, A, X to the stack in that order, and then pop X, A, Y from the stack in that order when finished.[MacDonald, 2002]

- For the alternating block transfer instructions (TAI and TIA), they alternate the source or destination address by adding and then subtracting one; not by inverting bit 0 of the address.[MacDonald, 2002]

- The length parameter to a block transfer instruction specifies the number of bytes to transfer. For example, 0010$willtransfer16bytes,and$0000 will transfer 64K bytes, not zero.[MacDonald, 2002]

- Block transfer instructions cannot be interrupted. If an interrupt is supposed to occur, it occurs once the instruction finishes.[MacDonald, 2002]

- When using any block transfer instruction to read addresses 0800$through$1400 in the I/O page, the value zero is always returned for every address, regardless of the CPU speed. (So you can't read the joystick port, timer, or IRQ registers) The I/O buffer is not changed either.[MacDonald, 2002]

Writing to the same range of addresses using the block transfer instructions will work, and the I/O buffer will be modified.[MacDonald, 2002]

TAI Transfer Alternate Increment

Function

Execute a memory move where the source address alternates between two addresses, and the destination address increments with each loop cycle. This is an extremely powerful instruction, mainly used for transferring data from the special video memory (e.g., backgrounds, etc.) to the main memory.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Block Mode	TAI SHSL DHDL LHLL	0xF3 SL SH DL DH LL LH	7	$17 + 6x$

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

TAM Transfer Accumulator to MPRi

Function

Loads Memory Mapping Register i with the value in the accumulator. More about the MPR registers can be found in the Memory Mapping summary. It is possible to load more than one MPR at a time by setting more than one bit in the immediate argument to TAM.

[In my understanding, if you set bits 0 and 3 high and the remainder low, upon execution these Memory Mapping Registers 0 and 3 would get set with the value in the accumulator. Since the number of cycles does not depend on how many are being set, its very likely that they are set in parallel by the hardware, perhaps by using expensive associative memory.]

John Robinson mentions:

> TAMI can transfer one value into multiple Memory registers. so if you do TAMI 30, #$FF it will transfer 30 into all of them (and then probably crash unless 30 is your working code page ;)) but #$C0 would set only the pages 6 and 7.
>
> [Robinson, 2005]

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	TAM #nn	0x53 nn	2	5

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

TAX Transfer Accumulator to X register

Function

Transfer the value in the accumulator to register X.

Addressing Modes & OpCodes

Addressing Mode	Syntax	Opcode	bytes	cycles
Implied	TAX	0xAA	1	2

Status Flags

N	V	T	B	D	I	Z	C
?	-	0	-	-	-	?	-

TAY Transfer Accumulator to Y register

Function

Transfer the value in the accumulator to register Y.

Addressing Modes & OpCodes

Addressing Mode	Syntax	Opcode	bytes	cycles
Implied	TAY	0xA8	1	2

Status Flags

N	V	T	B	D	I	Z	C
?	-	0	-	-	-	?	-

TDD Transfer Decrement Decrement

Function

Execute a memory move where the source and destination addresses decrement with each loop cycle. This is an extremely powerful instruction, mainly used for copying and moving data around in main memory.

Addressing Modes & OpCodes

Addressing Mode	Syntax	Opcode	bytes	cycles
Block Move	TDD SHSL,DHDL, LHLL	0xC3 SL SH DL DH LL LH	7	17 + 6x

Status Flags

N	V	T	B	D	I	Z	C
-	-	0	-	-	-	-	-

TIA Transfer Increment Alternate

Function

Execute a memory move where the source address increments, and the destination address alternates between two addresses with each loop cycle. This is an extremely powerful instruction, mainly used for transferring data to the special video memory (e.g., backgrounds, etc.) from the main memory.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Block Move	TIA SHSL,DHDL, LHLL	0xE3 SL SH DL DH LL LH	7	17 + 6x

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

TIN Transfer Increment None

Function

Execute a memory move where the source address increments with each loop cycle. This is an extremely powerful instruction, mainly used for transferring data from the special video memory (e.g., backgrounds, etc.) to the main memory.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Block Move	TIN SHSL,DHDL, LHLL	0xD3 SL SH DL DH LL LH	7	17 + 6x

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

TII Transfer Increment Increment

Function

Execute a memory move where the source address increments with each loop cycle. This is an extremely powerful instruction, mainly used for transferring data from the special video memory (e.g., backgrounds, etc.) to the main memory.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Block Move	TII SHSL,DHDL, LHLL	0x73 SL SH DL DH LL LH	7	17 + 6x

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

TMAi Transfer MPRi to Accumulator

Function

Transfers the value in Memory Mapping Register i to the accumulator. More information about the MPRs can be found on the Memory Mapping summary. Only one bit in the immediate argument can be set to 1.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	TMA #nn	0x43 #nn	2	4

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

TRB Test and Reset Memory Bits Against Accumulator

Function

Logically AND together the complement of the value in the accumulator with the data at the effective address specified by the operand. Store the result at the memory location. This clears each bit for which the corresponding accumulator bit is set, making it an ideal opcode for masking data. N and V and Z are set as in the BIT opcode instruction. These flags are set based on the ANDing of the uncomplemented accumulator value with the memory value.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Zero Page	TRB ZZ	0x14 ZZ	2	6
Absolute	TRB hhll	0x1C ll hh	3	7

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
M7	M6	0	-	-	-	?	-

TSB Test and Set Memory Bits Against Accumulator

Function

Logically OR together the value in the accumulator with the data at the effective address specified by the operand. Store the result at the memory location. This sets each bit for which the corresponding accumulator bit is set, making it an ideal opcode for masking data. N and V and Z are set as in the BIT opcode instruction. These flags are set based on the ANDing of the accumulator value with the memory value.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Zero Page	TSB ZZ	0x04 ZZ	2	6
Absolute	TSB hhll	0x0C ll hh	3	7

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
M7	M6	0	-	-	-	?	-

TST Test and Reset Memory Bits

Function

Logically AND together the immediate operand with the data at the effective address specified by the operand. This sets each bit for which the corresponding immediate argument bit is set, making it an ideal opcode for masking data. N and V and Z are set as in the BIT opcode instruction.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Imm. Zero Page	TST #nn, ZZ	0x83 nn ZZ	3	7
Imm. Zero Page, X	TST #nn, ZZ, X	0xA3 nn ZZ	3	7
Immediate Absolute	TST #nn, hhll	0x93 nn ll hh	4	8
Imm. Absolute, X	TST #nn, hhll, X	0xB3 nn ll hh	4	8

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
M7	M6	0	-	-	-	?	-

TSX Transfer stack pointer to X register

Function

Transfer the value in the stack pointer S to the X register. The value of the stack pointer is not changed.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	TSX	0xBA	1	2

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
?	-	0	-	-	-	?	-

TXA Transfer X register to accumulator

Function

Transfer the value in the X register to the accumulator. The value of the X register is not changed.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	TXA	0x8A	1	2

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
?	-	0	-	-	-	?	-

TXS Transfer X register to Stack Pointer

Function

Transfer the value in the X register to the stack pointer. The value of the X register is not changed.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	TXS	0x9A	1	2

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
-	-	0	-	-	-	-	-

TYA Transfer Y register to accumulator

Function

Transfer the value in the Y register to the accumulator. The value of the Y register is not changed.

Addressing Modes & OpCodes

Addressing Mode	**Syntax**	**Opcode**	**bytes**	**cycles**
Implied	TYA	0x98	1	2

Status Flags

N	**V**	**T**	**B**	**D**	**I**	**Z**	**C**
?	-	0	-	-	-	?	-

2.2.16 Addressing Modes

Instructions need operands to work on. There are various ways of indicating where the processor is to get these operands. The different methods used to do this are called addressing modes. The 6502 offers 11 modes, as described below.

Immediate In this mode the operand's value is given in the instruction itself. In assembly language this is indicated by "#" before the operand. eg. LDA #$0A - means "load the accumulator with the hex value 0x0A" In machine code different modes are indicated by different codes. So LDA would be translated into different codes depending on the addressing mode. In this mode, it is: $A9 $0A[_Bnu, 1999]

Absolute and Zero-page Absolute In these modes the operands address is given.
eg. LDA 0x31F6 - (assembler)
0xAD 0x31F6 - (machine code)
If the address is on zero page - i.e. any address where the high byte is 00 - only 1 byte is needed for the address. The processor automatically fills the 00 high byte. eg. LDA 0xF4 0xA5 0xF4 Note the different instruction codes for the different modes. Note also that for 2 byte addresses, the low byte is store first, eg. LDA 0x31F6 is stored as three bytes in memory, 0xAD 0xF6 0x31. Zero-page absolute is usually just called zero-page.[_Bnu, 1999]

Implied No operand addresses are required for this mode. They are implied by the instruction. eg. TAX - (transfer accumulator contents to X-register)[_Bnu, 1999]

Accumulator In this mode the instruction operates on data in the accumulator, so no operands are needed. eg. LSR - logical bit shift right[_Bnu, 1999]

Indexed and Zero-page Indexed In these modes the address given is added to the value in either the X or Y index register to give the actual address of the operand. eg. LDA $31F6, Y[_Bnu, 1999]

Note that the different operation codes determine the index register used. In the zero-page version, you should note that the X and Y registers are not interchangeable. Most instructions which can be used with zero-page indexing do so with X only.[_Bnu, 1999]

Indirect This mode applies only to the JMP instruction - JuMP to new location. It is indicated by parenthesis around the operand. The operand is the address of the bytes whose value is the new location.[_Bnu, 1999]

Pre-indexed indirect In this mode a zer0-page address is added to the contents of the X-register to give the address of the bytes holding the address of the operand. The indirection is indicated by parenthesis in assembly language.

Note a) When adding the 1-byte address and the X-register, wrap around addition is used - i.e. the sum is always a zero-page address. eg. FF + 2 = 0001 not 0101 as you might expect. DON'T FORGET THIS WHEN EMULATING THIS MODE. b) Only the X register is used in this mode.[_Bnu, 1999]

Post-indexed indirect In this mode the contents of a zero-page address (and the following byte) give the indirect addressm which is added to the contents of the Y-register to yield the actual address of the operand. Again, inassembly language, the instruction is indicated by parenthesis.
eg. LDA ($4C), Y
Note that the parenthesis are only around the 2nd byte of the instruction since it is the part that does the indirection.

Note: only the Y-register is used in this mode.[_Bnu, 1999]

Relative This mode is used with Branch-on-Condition instructions. It is probably the mode you will use most often. A 1 byte value is added to the program counter, and the program continues execution from that address. The 1 byte number is treated as a signed number - i.e. if bit 7 is 1, the number given byt bits 0-6 is negative; if bit 7 is 0, the number is positive. This enables a branch displacement of up to 127 bytes in either direction.[_Bnu, 1999]

Notes: a) The program counter points to the start of the instruction after the branch instruction before the branch displacement is added. Remember to take this into account when calculating displacements. b) Branch-on-condition instructions work by checking the relevant status bits in the status register. Make sure that they have been set or unset as you want them. This is often done using a CMP instruction. c) If you find you need to branch further than 127 bytes, use the opposite branch-on-condition and a JMP.[_Bnu, 1999]

3 Emulator Design

This latter part of this document is a treatises on the design, implementation, and testing of a complete PCE emulator.

emulate: Computer Science. To imitate the function of (another system), as by modifications to hardware or software that allow the imitating system to accept the same data, execute the same programs, and achieve the same results as the imitated system.

A first concern for our emulations design, is the "*c*ompletness" of the emulation. I'm refering to aspects of PCE that can be ignored, or re-interpreted without a regular PCE software user noticing any real difference between a real PCE and the emulated system; For example, most input to an emulator is reinterpreted into a format the emulated system would understand. Few emulators require you to add obscure hardware devices to plug in ancient gamepads. There will always be some differences between an emulator and the real system, these differences are usually to make the emulator more convienient; the question becomes how much deviation will the emulator employ.

There are two extremes: the first being that the emulator remain as true to the system as possible, only deviating were absolutely nessessary (like reading rom data from a file, and keyboard events, and mapping the system output to a drawing space or speakers on the host system). Internal operations within the system should occur exactly as they would in a real PCE, by using phony registers, and emulating the PCE memory within the application heap. The other extreme is to only worry about the output being appropriate. This emulator doesn't care what happens in a real PCE, so long as it can use the PCE instructions to build a comparible algorithm for the host system.

Of the two cases, the situation where the system is emulated instruction by instruction is by far the simpler and more common case. Its benefits are that it is both simple and much easier. The downsides are that the emulator structure is fairly riged, making optimization more difficult (although, when your emulating a system that runs at 4 mHz, optimization isn't really a requirement).

3.1 Programming Language

The language of implementation is a critical choice to the success of an emulator. The choice of emulator will effect the speed, reliability, extendability, project size and, most importantly, simplicity of the emulator. Although many languages can be used to implement the emulator, I'm certain C or in some cases C++ are the best languages for an emulator and the reason these langauges are superior is pointer manipulation.

That said, there are some downsides to C and C++, the lack of garbage collection being a big one; Automatic garbage collection is extremely nice and reduces development time substantially. Another problem is portability, although C and C++ compilers are ubiquitous a recompilation will be needed for each platform the emulator is ported too.

The downsides are easily outweighed by the upsides. A C or C++ program generates an executable and requires no additional software on the host machine to run. There is a large body of supporting libraries and software for C and C++ particularly for multimedia applications. C and C++ are sufficiently low-level to allow us to convieniently map rom instructions to various functions.

Those are my reasons for choosing C or C++, there are also factors that make alternative languages unsuitable. Java for instance, is a (imo) awesomely cool language with a large and growing body of libraries. The advantages of writing an emulator in Java are great: garbage collection, portability, easy access to testing frameworks, standard GUI API's, etc. However Java does not let us operate at a low enough level; function pointers and direct memory access, features which Java does not excel at, will make our lives a lot easier when emulating the PCE CPU.

Other languages like Prolog (a logical language) or Haskel (a functional language) allow for very high level descriptions of the problem. Unfortunately, these languages, like many others, are entirely unsuitable for the task of emulation. Often they simply don't have the libraries for multimedia, or are not really intended for tasks involving deadlines such as emulation.

Still other imperitive languages, like Perl, Lisp, or Python, fall flat once again because of the absence of suitable multimedia libraries. Additionally, I don't believe any of these languages offers pointers[2].

In conclusion, we have to take the good with the bad. Pointer manipulation is simply too good to pass up, as are the wonderful multimedia API's available to C and C++ programs (DirectX, openGL, SDL, and Allegro to name a few). The cost of this, is the convenience that comes with the higher level languages.

3.2 Programming Philosophy

In designing a program, any program even an emulator, it is most desirable to keep the system extremely simple. Eric Raymond's book "The Art of Unix Programming"[Raymond, 2004] has 17 excellent points that are copied here.

1. Rule of Modularity: Write simple parts connected by clean interfaces.
2. Rule of Clarity: Clarity is better than cleverness.
3. Rule of Composition: Design programs to be connected to other programs.
4. Rule of Separation: Separate polic from mechanism; separate interfaces from engines.
5. Rule of Simplicity: Design for simplicity; add complexity only where you must.
6. Rule of Parsimony: Write a big program only when it is clear by demonstration that nothing else will do.
7. Rule of Transparency: Design for visibility to make inspection and debugging easier.
8. Rule of Robustness: Robustness is the child of transparency and simplicity.
9. Rule of Representation: Fold knowledge into data so program logic can be stuiped and robust.
10. Rule of Least Suprise: In interface design, always do the least suprising thing.
11. Rule of Silence: When a program has nothing suprising to say, it should say nothing.
12. Rule of Repair: When you must fail, fail noisily and as soon as possible.
13. Rule of Economy: Programmer time is expensive; conserve it in preference to machine time.
14. Rule of Generation: Avoid hand-hacking; write programs to write programs when you can.
15. Rule of Optimization: Prototype before polishing. Get it working before you optimize it.
16. Rule of Diversity: Distrust all claims for the "one true way".
17. Rule of Extensibility: Design for the furture, because it will be here sooner than you think.

[2]If I'm wrong in either of these regards, please contact me via email to wagwilk@telusplanet.net

Of these rules, 3, 4, 5, 13, and 14 are the ones I'm most concerned with. I'd be very happy if I could design my emulator such that it could easily be connected to two seperate interfaces: A debuging one, where the emulator variables (such as memory, sprites, etc) are all viewable, and a playing interface, where there is only the emulators drawing window and output sound. By containing the emulator code away from the interface, like Rule 4, I can modify the front end without introducing bugs into the backend; different output interfaces (one built on SDL, another built on openGL) can be written and also be very simple.

Rule 13 is one of my major goals: I don't want to spend a lot of time writing the emulator. I want it a well designed emulator that functions appropriately, but I don't want to develop it through hours and hours of debugging and refactoring. I want to do it well the first time.

Rule 14 is absolutely brilliant and it's applied in John Robinson's [Robinson, 2005] emulator to good effect. I'll certainly be looking into areas where I can create code-generating tools.

In summary I think a good design is one in which the emulator (if run with the -v flag) outputs the command its executing and the results in simple table format so a developer can always find where the program was when it crashed. Most of the emulator's code will be generated (since its mostly a large switch statement anyway)[3]. For communicating data, the emulator will use unix Sockets; a socket for video data, one socket for audio data, another for mentioning what memory value has changed to what, an input socket for controls, etc etc. The number of these sockets created (and hence the amount of data that the emulator exports over sockets) will be controlled during compilation. If the DEBUG flag is specified, the entire product will be compiled to support a debugging session, with sockets outputting palette information, memory information, sprite information and more.

Although this adds complexity overall, it reduces each components complexity. The emulator doesn't draw anything, it doesn't sound anything, it doesn't even check input. This gives me additional flexibility for testing, and allows future extendability (perhaps an intermediate program is put between the video reader/screen drawer and the emulator. This program could copy the data across the network to other machines, so the emulator can be watched from numerous terminals. Combine this with an input reader that reads keyboard and network events, and suddenly the emulator is multiplayer! Not a single line of code has been added to the actual emulator, so its bug free!). I'll need to determine whether or not unix sockets are the way to go, they may not be portable, or an easier method of communication between the programs might exist.

3.3 Software Components

These are some of the components that will interact with the emulator to give it function. Components should communicate with almost self-documenting messages, that make it very easy for new tools to be built and debugged. For example, rather than the emulator sending message $0x0A$, then the number of parameters $0x02$ and then $0x45, 0xFF$, the message should be simple and clear: "MEMCHANGE AT 0x45 to 0xFF" – Wouldn't that be much easier for debugging?

3.3.1 Debugger Components

These components are designed to capture data and display it in a useable form. When combined with the emulator running in stepwise fashion, these software components will create a full featured debugging environment.

Register Viewer: This component should create a graphical window that shows the contents of each register. It should also interpret the values if they're bitflags.

Memory viewer: This component should show all the contents of the memory for the entire PCE. Ideally it should communicate what codepage the memory belongs to, or what its role is.

[3]Perhaps lex can be used to create this large table.

sprite viewer: This component should allow the user to view all the frames of all the sprites.

palette viewer: This component should show the user all of the palette information.

audio capture device: This component should be able to capture audio signals properly, even if the emulator is working in stepwise fashion. It should be able to play back the audio, and save it.

video frame capture device: This component should be able to capture video data and render it appropriately. It should be able to save the data.

(Optional) Code Viewer: If there is some way to tell what code has been used to generate the binary file being executed, a code viewer could easily be added. Breakpoints could be supported easily as well.

(Optional) ASM Viewer: If there is some way to tell what file has been loaded, an viewer could be written that converts that to ASM. Breakpoints could be supported easily as well.

3.3.2 Video Components

Video components should render the frames appropriately and render them in realtime.

3.3.3 Audio Components

Audio components should queue the audio samples for playback at the correct rate.

3.3.4 Input Components

Input components are programs written to gather input and feed it properly into the emulator.

input capture device: This compononent should gather input in realtime and send it to the emulator.

input repeater device: This component should read input data from a file and send it to the emulator. (useful for making tests).

Real Time Debuggers / Memory Checkers

1. Valgrind (`http://valgrind.org/`)
2. memfetch
3. CCured

Testing Frameworks

1. QTUnit
2. QCPPUnit
3. CPPUnit
4. CXXUnit
5. CUnit
6. CUT
7. Check

8. testify - test CLI responses

Profiler tools

1. C/C++ Program Perfometer.

Other

1. CMT++: Metrics testing framework (`http://www.testwell.fi/cmtdesc.html`)
2. CTC++: Test Coverage Analyzer for C/C++ (`http://www.testwell.fi/ctcdesc.html`)
3. BFBTester: Brute Force Binary Tester
4. PatternTesting (doesn't work with C)
5. cxxchecker - C++ source-code style check (`https://gna.org/projects/cxxchecker/`)
6. SLOCCount - Source lines of code count (`http://www.dwheeler.com/sloccount/`)
7. cqual - Add constraints to variables, ensure they are met (`http://www.cs.umd.edu/~jfoster/cqual/`)
8. ccide - Add nice decision tables to C code (`http://www.ccide.com/`)

4 Appendix

4.1 Changes

Changes in Version 0.2

- Copied information from Jens Restemeier's documentation [Restemeier, 1997], to flesh out the instruction set.
- Reorganized the tex files.
- Added some basic information on my own emulator design.
- Added instructions ordered by numeric value to the appendix.

4.2 Remaining To Do

- Fix notation for certain instructions
- Discuss code reuse
- Determine whether operations across a page boundary take more cycles.
- Determine which instruction 0x40 is and which instruction 0x4D is. There is confusion about this (see the EOR and RTI instructions).
- Add content like a madman!

4.3 Ordered List of Instructions

0x00	BRK
0x01	ORA
0x02	SXY
0x03	ST0
0x04	TSB
0x05	ORA
0x06	ASL
0x08	PHP
0x09	ORA
0x0A	ASL
0x0C	TSB
0x0D	ORA
0x0E	ASL
0x0F	BBR
0x10	BPL
0x11	ORA
0x12	ORA
0x13	ST1
0x14	TRB
0x15	ORA
0x16	ASL
0x18	CLC
0x19	ORA
0x1A	INC
0x1C	TRB
0x1D	ORA
0x1E	ASL
0x1F	BBR
0x20	JSR
0x21	AND
0x22	SAX
0x23	ST2
0x24	BIT
0x25	AND
0x26	ROL
0x28	PLP
0x29	AND
0x2A	ROL
0x2C	BIT
0x2D	AND
0x2E	ROL
0x2F	BBR
0x30	BMI
0x31	AND
0x32	AND
0x34	BIT
0x34	BIT
0x35	AND
0x36	ROL
0x38	SEC
0x39	AND
0x3A	DEC
0x3D	AND
0x3E	ROL
0x3F	BBR
0x40	EOR
0x41	EOR
0x42	SAY
0x43	TMA
0x44	BSR
0x45	EOR
0x46	LSR
0x48	PHA
0x49	EOR
0x4A	LSR
0x4C	JMP
0x4D	RTI
0x4E	LSR
0x4F	BBR
0x50	BVC
0x51	EOR
0x52	EOR
0x53	TAM
0x54	CSL
0x55	EOR
0x56	LSR
0x58	CLI
0x59	EOR
0x5A	PHY
0x5D	EOR
0x5E	LSR
0x5F	BBR
0x60	RTS
0x61	ADC
0x62	CLA
0x64	STZ
0x65	ADC
0x66	ROR
0x68	PLA
0x69	ADC
0x6A	ROR
0x6C	JMP
0x6D	ADC
0x6E	ROR
0x6F	BBR
0x70	BVS
0x71	ADC
0x72	ADC
0x73	TII
0x74	STZ
0x75	ADC
0x76	ROR
0x78	SEI
0x79	ADC
0x7A	PLY
0x7C	JMP
0x7D	ADC
0x7E	ROR
0x7F	BBR
0x81	STA
0x82	CLX
0x83	TST
0x84	STY
0x85	STA
0x86	STX
0x87	SMB
0x88	DEY
0x89	BIT
0x8A	TXA
0x8C	STY
0x8D	STA
0x8E	STX
0x8F	BBS
0x90	BCC
0x91	STA
0x92	STA
0x93	TST
0x94	STY
0x95	STA
0x96	STX
0x97	SMB
0x98	TYA
0x99	STA
0x9A	TXS
0x9C	STZ
0x9D	STA
0x9E	STZ
0x9F	BBS
0xA0	LDX
0xA1	LDA
0xA2	LDX
0xA3	TST
0xA4	LDX
0xA5	LDA
0xA6	LDX
0xA7	SMB
0xA8	TAY
0xA9	LDA
0xAA	TAX
0xAC	LDX
0xAD	LDA
0xAE	LDX
0xAF	BBS
0xB0	BCS
0xB1	LDA
0xB2	LDA
0xB3	TST
0xB4	LDX
0xB5	LDA
0xB6	LDX
0xB7	SMB
0xB8	CLV
0xB9	LDA
0xBA	TSX
0xBC	LDX
0xBD	LDA
0xBE	LDX
0xBF	BBS
0xC0	CPY
0xC1	CMP
0xC2	CLY
0xC3	TDD
0xC4	CPY
0xC5	CMP
0xC6	DEC
0xC7	SMB
0xC8	INY
0xC9	CMP
0xCA	DEX
0xCC	CPY
0xCD	CMP
0xCE	DEC
0xCF	BBS
0xD0	BNE
0xD1	CMP
0xD2	CMP
0xD3	TIN
0xD4	CSH
0xD5	CMP
0xD6	DEC
0xD7	SMB
0xD8	CLD
0xD9	CMP
0xDA	PHX
0xDD	CMP
0xDE	DEC
0xDF	BBS
0xE0	CPX
0xE1	SBC
0xE3	TIA
0xE4	CPX
0xE5	SBC
0xE6	INC
0xE7	SMB
0xE8	INX
0xE9	SBC
0xEA	NOP
0xEC	CPX
0xED	SBC
0xEE	INC
0xEF	BBS
0xF0	BEQ
0xF1	SBC
0xF2	SBC
0xF3	TAI
0xF4	SET
0xF5	SBC
0xF6	INC
0xF7	SMB
0xF8	SED
0xF9	SBC
0xFA	PLX
0xFD	SBC
0xFE	INC
0xFF	BBS

Glossary

Bank A bank is synomous with a page. In order for a bank to be used it must be mapped to a logical address within the TG16. Available Bank maps are shown in table 4, on page 6.

VDC Second video processor on the TG16

Index

References

[_Bnu, 1999] _Bnu (1999). 6502 microprocessor reference manual. http://www.zophar.net/tech/files/6502.txt.

[David Michel, 2003] David Michel, C. M. (2003). Magickit pce hardware documentation. http://www.magicengine.com/mkit/.

[MacDonald, 2002] MacDonald, C. (2002). Turbografx-16 hardware notes. http://cgfm2.emuviews.com/txt/pcetech.txt.

[Raymond, 2004] Raymond, E. S. (2004). *The Art of Unix Programming.* Addison-Wesley.

[Restemeier, 1997] Restemeier, J. C. (1997). Unofficial pc-engine reference. http://www.classicgaming.com/epr/pc-engin/pcedoc_ps.zip.

[Robinson, 2005] Robinson, J. S. (2005). private communications.

[Zophar, 2004] Zophar (2004). 65c02 reference manual. http://www.zophar.net/tech/files/6502ref.html.