An overview of the FT832 CPU Core. Includes documentation on core register set, core instructions, parameters and configuration.

FT832 CPU Core

robfinch@finitron.ca

Table of Contents

Overview

The design of this core has been guided by discussions on the 6502.org forum. Features of the core include truly flat 32 bit addressing and 32 bit indirect addresses. The core is 65832 backwards compatible. Also supported by this core is simple high-performance task switching. A segmented memory protection model has been added to the core as an option. New instructions have been added to support core functionality. Some of the instruction set has been designed around the notion that this core will be required for more heavy duty apps. The WDM opcode is used to extend the functionality of many existing instructions. When the existing instruction opcode is prefixed with 42h it may have additional functionality including extended address modes. For instance prefixing the REP instruction with 42h causes it to use a 16 bit immediate to update both the status register and new extended status register.

Features

Some features include:

Expanded addressing capabilities (32 bit addressing modes)
Instruction caching
Code, data and stack segment registers
Multiple register contexts and high speed context switching
Interpretive operating mode
Single step mode
Task / context vectoring interrupts (in native 32 bit mode)
Combinational signed branches (branches that test both N and Z flags at the same time).
Relative branches to subroutine (for position independent code)
Long branching for regular branch instructions
Multiply instruction
Enhanced support for variable size data (size prefix codes)
Configurable context / task switch support

Programming Model

The programming model is compatible with the W65C816S programming model, with the addition of three new segment registers and a task register. A number of new instructions and addressing modes have been added using the opcode reserved for that purpose (the WDM opcode). There is also an array of 512 task context registers if the core is configured for hardware support of tasks.

Register	Size		
CS	16	code selector	
PB	8	program bank	
PC	16	program counter	
Acc	32	accumulator	
x	32	x index register	
y	32	y index register	
SP	32	stack pointer	
DS	16	data selector	
SS	16	stack selector	
DB	8	data bank	
DPR	16	direct page register	
SR	8	status register	
SRX	8	status register extension	
TR	16	Task Register	

Task Context Register Array (present only if hardware task support is configured):

Register	
0	Register context
…	
511	

Register Settings on Reset

Note that the hidden segment descriptor cache registers are setup for a flat addressing model on reset so that the core may form proper memory addresses for the reset routine. A far jump or call will set the CS descriptor cache registers to the value from the segment descriptor table, which must have been previously set up. The TASS instruction will set the SS descriptor cache registers from the segment descriptor table. Finally the PLDS instruction will set the DS descriptor cache registers from the segment descriptor table.

Reg	Value	Note:
CS	Zero	- reset to zero – required since the CS is not part of the reset vector
PB	$00	- reset to zero – required since the PB is not part of the reset vector
PC	$FFF0	- this register value will be overwritten and automatically loaded from the reset vector in memory on a reset
DS	---	- not set by reset
SS	---	- "
DB	---	"
DPR	---	"
A		"
X		"
Y		"
SP	$000001FF	- since the stack page is being set to page 1, the remainder of the stack pointer is set as well
SR	%xx0x01xx	- interrupts are masked, and decimal mode is cleared (note the m and x bits are set but not visible as part of the status register because the core starts in eight bit emulation mode).
SRX	%xxx0x000	- the emulation mode is set to eight bit, both the 32 and 16 bit emulation flags are cleared, interpreter mode and single step mode are disabled.
TR	$00	- the task register identifies which task is running. It is an internal register, set indirectly by the TSK instruction.
Hidden Registers		
CS_BASE	$00000000	The code segment is set to not writable on reset.
CS_LIMIT	$FFFFFFFF	
DS_BASE	$00000000	The data segment is set to writable
DS_LIMIT	$FFFFFFFF	
SS_BASE	$00000000	The stack segment is set to writable
SS_LIMIT	$FFFFFFFF	

On reset the contents of the task context register array is undefined.

On reset the contents of the segment descriptor table are undefined.

New Registers

There are three new segment selector registers CS, DS and SS standing for Code Segment, Data Segment and Stack Segment respectively. The addition of these registers is a result of discussions on 6502.org. Forum members expressed a desire to have a full 32 bit program bank and data bank registers allowing the base address of the program or data to be placed anywhere in memory. This is the function of a segment register. Rather than modify the existing program bank and data bank registers, three new segment registers were added. This allows the core to be backwards compatible with the 65816/65832 design. If desired the program bank and data bank registers may be set to zero, and the 32 bit CS and DS registers used to place code / data in memory. Alternately the CS and DS registers could be set to zero and the core used as a 65816/65832 compatible core. There are new instructions (PHCS, PHDS, PLDS) to support use of the CS and DS registers in a manner similar to the program bank and data bank registers.

There is an extension to the status register called the SRX register, which contains the emulation mode setting bits. The emulation mode setting is stored as part of the task context. This allows the core to run different emulation modes in different tasks. The 65816/65832 doubles up on the usage of the C and V flags in the status register in order to set the processor mode. This approach was likely used in order to avoid creating another program visible register in the processor. This is acceptable because there isn't really a need to store the emulation mode bits. A new register has been added in this design in order to support additional core options.

The task register TR is used to hold the index of the current context register. This could be thought of as the current process ID. The contents of the task register are made available with the TTA instruction. The task register is set by the task switching instructions.

Status Register Extension

Bit		Usage
0	m816	16 bit emulation mode flag
1	m832	32 bit native mode operation flag
2	in	interpreter mode
3	~	
4	ssm	single step mode
5	~	

6	~	
7	~	

Context Registers

When configured with hardware task functionality (the default configuration) the core includes an array of 512 context registers. Each register holds an entire program visible register set. The contents of the context registers may be set using the LDT instruction (the back link field is not settable). The contents of the context register may also be inherited from the current task when the FORK instruction is executed. The contents of the context registers are readable using the INF (information) instruction.

Context Register Layout

Start Bit Position	Size	Field Number (INF insn)	Field
240	9	12	Back Link
224	16	1	Code Selector
208	16	2	Data Selector
192	16	3	Stack Selector
184	8	4	PB
168	16		PC
136	32	5	ACC
104	32	6	.X
72	32	7	.Y
40	32	8	SP
24	16	9	SR, SRX
16	8	10	DB
0	16	11	DPR

To switch between tasks switch the active context of the processor using the TSK instruction. The currently active context is pointed to by the task register (TR). The jump to context(JCI, JCF, JCR, JCL) instructions may also be used to switch contexts.

Memory Layout for LDT instruction:

```
TaskStartTbl:
       .WORD  0                    ; CS
       .WORD  0                    ; DS
       .WORD  0                    ; SS
       .WORD  Task0         ; PC
       .BYTE  Task0>>16
       .WORD  0                    ; acc
       .WORD  0
       .WORD  0                    ; x
       .WORD  0
       .WORD  0                    ; y
       .WORD  0
       .WORD  $3FFF         ; sp
       .WORD  0
       .BYTE  0                    ; SR
       .BYTE  1                    ; SR extension
       .BYTE  0                    ; DB
       .WORD  0                    ; DPR
       .WORD  0
```

Operating Modes

Interpretive Mode

The core features direct support for interpreters. Interpreters are supported by ping-ponging between two contexts. One context acts to fetch instructions, perhaps a bytecode, and the other context interprets the fetches. A task may be placed in interpretive operating mode by setting bit #2 of the extended status register. In this mode the core fetches instructions into the accumulator rather than the instruction register. Then the task is switched to an interpreting task before the instruction can get to the decode stage. Switching back and forth between contexts is fast. It can be done in as little as six clock cycles.

Because the program counter is directly used to fetch the code to be interpreted the instruction cache ends up being used for storage of the code. This enhances the performance of an interpreter.

The stride of the code fetches in interpreter mode is controlled by bits 0 and 1 (the m832 and m816 flags) of the extended status register.

m832 flag	m816 flag	Fetch stride
0	0	bytes are fetched
0	1	16 bit words are fetched
1	0	32 bit words are fetched
1	1	reserved

Single Step Mode

If single step mode is enabled, a task switch to the single stepping task occurs after each instruction is executed. This allows the task to be single stepped under the control of the single stepping task. The single stepping task/context is defined to be task #3.

Instruction Cache

For better performance, memory is often organized in a hierarchy that consists of caches isolating the access to main memory. Caches are faster than main memory, and higher level caches (closest to the cpu) are faster than lower leveled ones. In the FT832 cpu all instruction accesses are cached. While this doesn't necessarily result in better instruction execution performance for the intended target of the FT832 (a PLD), it does reduce the amount of traffic on the bus. This means that systems sharing the bus can have better performance as bus availability is increased. For instance the TSK instruction takes four cycles to execute, but doesn't use the bus. Hence the bus is available for at least four consecutive clock cycles while the TSK instruction executes.

The default instruction cache is organized as 256, 16 byte lines. An entire cache line is loaded with back-to-back memory read operations as fast as the memory system will allow. The leading byte of an instruction cache line fetch is signified with both VPA and VDA signals being active. This is similar to the first byte of an opcode fetch being signified in the same manner on the 65816.

Cache lines may be pre-loaded so that the performance of specific code is not impacted by line loads. The cache may also be invalidated on a line-by-line basis, or the entire cache can be invalidated. Cache control is via the 'CACHE' command instruction. Note that invalidating or pre-loading a cache line that conflicts with the current instruction's cache line causes the instruction's cache line to be reloaded from memory (otherwise the core wouldn't be able to execute instructions). Care must be taken to place code such that cache line conflicts do not occur if it is desired to preload the cache lines.

The core uses a 16 byte window into the instruction cache from which instruction data is read. All 16 bytes are available in parallel within a single clock cycle. This means that the instruction fetch time is always fixed at a single clock cycle regardless of the length of an instruction. IT also means that an instruction including any prefixes cannot be longer than 16 bytes. The window slides as the program counter value changes. This window will usually span two cache lines. On occasion it may be necessary to fetch two lines from memory in order for an instruction spanning cache lines to execute.

The instruction cache is physically indexed and tagged. The cache is driven by the address resulting from the sum of the code segment and program counter. This results in only a single image of instructions in the cache when different combinations of the program counter and code segment result in the same address.

Segmentation Model

The segmentation model used by FT832 is extremely simple. There are only three segment selector registers (code, data and stack) and addresses are formed by a simple addition of a segment value to the program counter and effective data address. All data access is associated with the data segment. All instruction access is associated with the code segment. All stack accesses are associated with the stack segment. There is no way to override the association of the code segment with instruction access (program counter). For data access the segment may be overridden using one of the segment override prefixes (CS:, SS: SEG:, SEG0, IOS:). The segment associated with stack instructions may not be overridden.

On reset both the code segment and data segment selector registers are set to zero.

The code segment may be set using the JMF, JSF, JCF far instructions. The code segment may also be set in the task start-up record and loaded with the context via the LDT instruction.

The data segment may be set by pushing a value on the stack then pulling the data segment from the stack using the PLDS instruction.

The stack segment selector may be set by transferring a value to it from the accumulator using the TASS instruction.

Selectors

There is a level of indirection when dealing with segments. Segment values are not directly encoded into the instruction instead a selector value is used. The selector values require only 16 bits rather than the 32 bits (or more) required for a segment value. This saves two bytes every time a segment value is needed. For instance when calling a far routine only the 16 bit code selector needs to be saved rather than a 32 bit code segment value. This save two memory accesses and the instruction is two bytes smaller. The selector points to an entry in the segment descriptor table. The actual segment value is found in the segment descriptor table.

Segment Register (or Selector) Format:

15 14	13 12	11 0
$\sim_2$	$\sim_2$	Index$_{12}$

Index$_{12}$: the index into the descriptor table

The Segment Descriptor Table

The segment descriptor table contains information on the location and size for segments. The segment selectors point to entries in this table. This table is a special 4k entry dual-ported memory embedded within the processor. Entries in the table have the following format:

	43 36	35 32	31 0
w0	ACR$_8$	Size$_4$	Base$_{31..0}$
w1	ACR$_8$	Size$_4$	Base$_{31..0}$
…			

Size$_4$ Field

The segment is usually expanded in chunks.

Bits	Data Size
3-0	0000 = 0 0001 = 256

	0010 = 1024 0011 = 4096 0100 = 16384 0101 = 65536 0110 = 256k 0111 = 1MB 1000 = 4 MB 1001 = 16 MB 1010 = 64MB 1011 = 256MB 1100 = 1GB 1101 = 4GB

***ACR$_8$* Field**

7	6	5	4	3	2	1 0
P	A	C	X	W	$\sim_1$	$\sim_2$

P: Present bit

A: accessed

C: conforming

X: executable

W: writeable

It's recommended that after reset the first segment descriptor be set to all zeros. This will then result in protection faults if a NULL segment is passed as a pointer to a routine.

Segmentation Faults

Value		
0	SEGF_NONE	No fault is present
1	SEGF_EXEC	Attempt to execute code in a non-executable segment
2	SEGF_WRITE	Attempt to write to a read only segment
3	SEGF_BOUND	Attempt to access outside of segment bounds. The address is greater than the limit set by the segment size attribute.

These values can be retrieved from the processor fault register of the INF instruction.

Multi-Tasking

Overview

The FT832 core has hardware support for a multi-tasking operating system. One of the requirements for the tasking system is that it be fast. A goal was that context switching be at least as fast as could be done on the 65xxx series. One of the attractive features of the 65xxx series is the limited amount of context which is required to be stored during a context switch. This results in extremely fast context switching. As a result the latency in processing interrupt routines is low. One of the problems with adding additional registers to the programming model is that the context switch time is impacted. In keeping with low latency context switches, switching contexts with the FT832 can be done in as little as six clock cycles. Unlike some other cpu's supporting multi-tasking, the register context isn't saved to memory during a context switch. Instead the register context is saved in a dedicated register array. Access to this register array is single cycle for storing all registers or restoring all registers. This allows the FT832 to be even faster (lower latency) for processing interrupt routines while at the same time supporting an expanded programming model.

A second requirement of the tasking system is that it be simple. Target applications of the FT832 are more for embedded systems rather than being a full-fledged workstation type processor.

Operation

At reset the core begins running software in task / context #0. Since the core does not automatically load from the task start-up table at reset, it is necessary to initialize the register set manually. This is no different than the existing 65xxx series initialization requirements. See the table "Reset Settings on Reset" to determine which registers are pre-set to which values. For other tasks the entire register set may be pre-set from entries in the task start-up table.

The task start-up table is table of 32 byte entries which contain starting values for each of the processor's program visible registers. This table may be located anywhere in memory. The processor's internal registers are not loaded from the start-up table; just the ones that can be programmed. Entries in the start-up table may be loaded into processor's task context registers using the LDT instruction.

Loading a task context from a start-up table entry does not automatically start the task. The task will be started when it is invoked with the TSK instruction.

In native 32 bit mode the core may be configured (default) to use task number for interrupt vectors rather than addresses. It’s lower latency to switch tasks automatically on interrupt rather than first going to an interrupt service routine. Using a task number allows the interrupt processing routine to be located anywhere in memory while the vector contents are only 16 bits.

It is possible to switch to a 32 bit interrupt handler during interrupt processing for 8 or 16 bit emulation modes.

Assembler Notations

Since the core supports 32 bit indirect addressing a new notation is required for assembler code. Thirty-two bit indirect addresses are denoted with { } characters. For instance to access data pointed to with a 32 bit indirect address: LDA {$23},Y

The FT832 core also has operand size control prefixes. These prefixes are specified by appending a dot code onto the instruction they apply to. For instance to apply the BYT prefix to the LDA instruction use the notation “LDA.B”.

Instruction Suffix		
.B	signed byte operand	
.UB	unsigned byte operand	
.H	signed half-word (16 bit) operand	
.UH	unsigned half-word (16 bit) operand	

Mnemonics

For the 6502 all instructions could be represented by a three character code. This made sense as it allowed a compact set of mnemonics when dealing with the very limited amounts of memory available when the 6502 was developed. Moving forward in time with many new instructions it becomes difficult to represent the operation using only three character codes. Since there is substantially more memory available in today’s systems the author decided to forego the three character limitation and use mnemonics which may include more characters.

New Addressing Modes

There are several new addressing modes for existing instructions. Extra-long addressing for both absolute and absolute indexed addresses is available. The extra-long addressing mode is formed by prefixing the regular absolute address modes opcode with the extended opcode indicator byte ($42). This gives access to a 32 bit offset for a number of instructions which were not supported by the absolute long address modes. Extra-long indirect addressing modes are additional addressing mode available in the same manner as extra-long addressing. The indirect address mode instructions are prefixed with the opcode extension byte ($42).

The segment value of an address may be explicitly specified using the SEG prefix. The SEG prefix may be applied to direct page addresses, absolute addresses, absolute long addresses, and absolute extra-long addresses. The assembler syntax has the form:

```
LDA $34:$1234
```

This tells the core to use the segment associated with selector number $34 and load the accumulator with the value at offset $1234 in the segment. If the segment isn't specified either the data or stack selector will be used depending on the addressing mode.

It is possible to use segmented indirect addresses by applying the FAR prefix to the indirect address instruction. As in the following:

```
EOR FAR ($100,X)
```

In this case the address looked-up is a far address. The address will contain two extra bytes in order to specify a selector. In this case the address will contain four bytes. Two for the offset and two for the segment. This may be most useful for specifying subroutine arguments that are far (segmented) addresses. As in:

```
LDA FAR {3,S},Y
```

which uses a six byte indirect address located on the stack (four for offset and two for selector) to load the accumulator from.

Software Examples

Power on reset code

The following code switches the core to ‘816 mode then sets ups several segment descriptors in the segment descriptor table.

```
       .org   $E000

start:
       SEI                      ; not strictly necessary after power on reset
       CLD
       CLC                      ; switch to '816 mode
       BIT           start      ; set overflow bit
       XCE
       REP           #$30       ; set 16 bit regs & mem
       NDX           16         ; tell the assembler
       MEM           16

       ; Setup all the segment descriptors
       ; This must be done before the stack can be accessed, and
       ; before subroutine calls and task switching can be made.

       ; First setup segment #0, the segment the core starts running under.
       ; It's setup for flat addressing.
       LDA           #0
       XBAW                      ; swap upper and lower 16 bits of 32 bit acc
       LDA           #0          ; 32 bit acc contains zero now
       LDX           #$98D       ; present, writeable, executable, max size.
       TAY                       ; select entry #0
       SDU                       ; special instruction to update segment descriptor table
       INY                       ; advance to next descriptor entry

       ; Flag remaining segments as not present
       LDX           #$000       ; not present, non-executable, non-writeable, min size
.0003:
       SDU                              ; update segment descriptor
       INY
       CPY           #4096       ; table has 4096 entries
       BNE           .0003

       ; setup code segment #1
       LDX           #$905         ; executable, 64k
       TAY
       INY
       SDU
       ; setup data segment #2
       LDX           #$88B         ; writeable, 256M
       INY
       SDU
       ; setup stack segment #3 (for 65c02 mode)
       LDX           #$882         ; writeable, 1k (based at zero)
       INY
       SDU
       ; setup stack segment #4 (for 65c816 mode)
       LDX           #$885         ; writeable, 64k (based at zero)
       INY
       SDU
       ; seg #5 is the maxed out segment
       LDX           #$98D         ; executable, writable, max size
       INY
       SDU
       ; seg #9 starts in the ddr ram
       LDA           #2                   ; begin at $20000
       XBAW
       LDA           #0
       LDX           #$98D         ; executable, writable, max size
       LDY           #9
       SDU
       ; now set the code segment (test far jump)
       JMF           1:.0004
.0004:
       ; set the stack segment
       ; (must be before PEA/PLDS)
       LDA           #5
       TASS
       LDA           #$6BFF        ; set top of stack
       TAS
```

Instruction Set Summary

What's Covered

Only the enhanced instruction set instructions are documented here. Documentation on the remaining instructions which are W65C816S compatible is well done in the W65C816 programming manual. One notable difference is the instruction timings. The clock cycle counts for this core are not guaranteed to match those of a genuine 65C816.

Timing

Instruction timings may be dependent on memory access time for those instructions which access memory. The clock cycles counts are assuming that memory can be accessed in a single cycle. In many systems this is not the case and the memory system will insert wait states. Internal states are performed in a single clock cycle. Instruction fetch is single cycle to retrieve all bytes associated with the instruction assuming the instruction is located in the cache.

AAX

Add .X register to accumulator.

Opcode Format (2 bytes)

42	8A

3 clock cycles

The C, N, V, and Z flags are updated by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode / execute the instruction

ASR

This instruction shifts the accumulator to the right while preserving the sign bit. The least significant bit is placed into the carry flag.

Opcode Format (2 bytes)

42	0A

3 clock cycles

The C, N, and Z flags are updated by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode / execute the instruction

BGT

BGT stands for branch greater than. This is a branch based on a signed comparison of two values. It takes only the negative and zero flags into consideration. The branch is taken if both the negative and zero flags are false. This instruction improves code density and performance compared to performing a sequence of instructions to synthesize this operation. Overflow can be checked with the BVS instruction prior to executing BGT.

Opcode Format (3/5 bytes)

42	10	$Disp_8$		short
42	10	FFh	$Disp_{16}$	long

3 clock cycles (regardless of taken or not taken).

No flags are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode / execute the instruction

BLE

BLE stands for branch less or equal. This is a branch based on a signed comparison of two values. Only the negative and zero flags are tested. This instruction is the same as a combination of BEQ and BMI. This instruction improves code density and performance compared to performing a sequence of instructions to synthesize this operation. If an overflow check is required the BVS instruction can be used prior to executing this instruction.

Opcode Format (3/5 bytes)

42	B0	$Disp_8$	
42	B0	FFh	$Disp_{16}$

3 clock cycles (regardless of taken or not taken).

No flags are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode / execute the instruction

BSL

Branch to a subroutine long. The program bank and program counter are stored on the stack. Next a relative branch is taken to the sum of the program bank and counter with the displacement specified in the instruction. The relative branch may span program banks. The RTL instruction should be used to return from the called routine. The BSL instruction allows the development of position independent code.

Opcode Format (5 bytes)

42	48	$Disp_{24}$

7 clock cycles (4 + 3 memory accesses).

No flags are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode / execute the instruction
STORE1	set segment
STORE2	push PB
STORE2	push PC[15:8]
STORE2	push PC[7:0]

BSR

Branch to a subroutine. The program counter is stored on the stack. Next the program counter is loaded with the relative address calculated as the sum of the current program counter and a displacement specified in the instruction. The relative branch “wraps around” within the program bank. The program bank register is not modified by this instruction. The RTS instruction should be used to return from a routine called by this instruction. The BSR instruction allows the development of position independent code.

Opcode Format (4 bytes)

42	28	Disp_{16}

6 clock cycles (4 + 2 memory accesses).

No flags are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode / execute the instruction
STORE1	set segment
STORE2	push PC[15:8]
STORE2	push PC[7:0]

BYT

The BYT prefix causes the memory access of the following instruction to be byte sized regardless of the settings in the status register. The BYT prefix may be specified in assembler code by appending a “.B” to the instruction mnemonic. When a register is being loaded as a result of a byte sized memory operation, the value from memory is sign extended to the size of the register.

Opcode Format (2 bytes)

42	8B

2 clock cycles

No flags are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode following instruction
….	

Sample Assembler:

```
LDA.B $1000,Y   ; load and sign extend byte into sixteen bit accumulator
```

CACHE

CACHE issues a command to the cache. Currently only three commands are supported:

00 – invalidate entire instruction cache, (3 clock cycles)

01 – invalidate instruction cache line identified by accumulator (3 clock cycles)

02 – preload instruction cache line identified by accumulator (19 clock cycles 3 + 16 memory)

When the instruction cache line needs to be identified the accumulator holds the address desired to be invalidated, not the line number. The line number is determined by the address. Currently with a 16 byte cache line size the address is shifted right four times and masked with $FF to determine the line number. The cache line is loaded using back-to-back memory read operations.

Opcode Format (3 bytes)

42	E0	Immediate$_8$

No flags are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode / execute the instruction
ICACHE2	Only for preloads
…	“ repeats 15 more times

CMC

This instruction complements the carry flag. While not used very often, it can be tricky to complement the carry flag. Availability of this instruction eases some programming tasks.

Opcode Format (2 bytes)

42	18

3 clock cycles

The carry flag is inverted.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode / execute the instruction

CS:

This is a segment override prefix indicating to use the CS register in calculating a data address rather than the DS register. This prefix is treated as part of the current instruction. No interrupt will be allowed between the prefix and following instruction.

This prefix is typically used to access tables and other constants which are placed into the code segment.

A store or read-modify-write instruction prefixed with the CS: prefix will cause the memory store operation to be ignored. CS: is treated as a read-only segment.

Opcode Format (2 bytes)

42	1B

2 clock cycle

No flags are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode the following instruction
….	states for following instruction

DEX4

Decrement the .X index register by four. This instruction is similar to the DEX instruction except that it decrements by four rather than by one. With a 32 bit word size for most registers arrays are often 32 bits (four bytes). Indexing into word arrays requires adjusting the index by four.

Opcode Format (2 bytes)

42	CA

3 clock cycles

N and Z flags are affected.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode / execute the instruction

DEY4

Decrement the .Y index register by four. This instructions is similar to the DEY instruction except that it decrements by four rather than by one. With a 32 bit word size for most registers arrays are often 32 bits (four bytes). Indexing into word arrays requires adjusting the index by four.

Opcode Format (2 bytes)

42	88

3 clock cycles

N and Z flags are affected.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode / execute the instruction

FIL

Fill memory. This instruction fills a block of memory with the value contained in the .X register. The .Y register along with the data bank specified in the instruction points to the beginning of the block of memory. The .Y register is incremented and the accumulator decremented until the accumulator reaches minus one. The accumulator holds one less than the count of how many bytes to fill. In 8/16 modes, at the end of the fill operation the data bank is loaded with the value specified in the instruction. The data bank setting is ignored in 32 bit mode. The MVN / MVP instructions may also be used to fill a block of memory however the FILL instruction is faster as it only performs memory stores.

FILL normally fills with bytes however it may be prefixed with a size code in order to cause the fill to work with either half-words (16 bits) or words (32 bits). For example the assembler syntax for a word oriented fill is: FILL.W. Using a size code prefix causes the fill operation to take two more clock cycles per unit filled.

In native 32 bit mode the data bank specified in the instruction is not used to determine the fill address. The block filled may span a bank boundary. The instruction remains three bytes long.

A segment override prefix may be applied to this instruction however the prefix will cause the clock cycle count to increase by two per each byte stored. It may be faster to save before the instruction and restore the data segment afterwards. Note that attempts to fill the code segment (using the CS: prefix) will be ignored.

The fill operation is interruptible.

This mnemonic is closely resembles the FILL pseudo-op, as a memory aid as to which is which instruction mnemonics are usually three characters long. The FILL pseudo-op statically fills a region of memory at time of assembly. The FIL instruction fills memory dynamically at run-time.

Opcode Format (3 bytes)

42	44	DBR_8

6 clock cycles per byte stored (4 + 2 memory accesses)

No flags are affected by this instruction.

Machine States:

The following machine states repeat until the store is complete.

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode / execute the instruction
STORE1	
STORE2	store x[7:0] and Increment / decrement registers
STORE2[1]	store x[15:8] and Increment / decrement registers
STORE2[2]	store x[23:16] and Increment / decrement registers
STORE2[2]	store x[31:24] and Increment / decrement registers
MVN816	Test accumulator for -1

1 this cycle is present only for half-word and word fill operations.

2 this cycle is present only for word fill operations.

FORK

Fork starts a new task by making a copy of the current task's registers. Note that all registers are copied including the data segment and stack pointer. It will often be desirable to subsequently set the stack pointer and possibly the data segment to new values so that the new task has its own local data. Note that a task started with FORK does not require an entry in the task start-up table or use of the LDT instruction. Since FORK inherits all the register values from the current task, the core remains in the same mode. An attempt to fork the same task as the one that is already running is ignored.

IF the core is not configured to use back-links:

> The FORK operation pushes the current task register onto the stack. Since both the original task and the newly forked task are using the same stack pointer, interrupts should be disabled while the fork operation is taking place, until a new stack is setup for the forked task.

The operand to this instruction specifies which task context register to use to store the new task's registers in.

Opcode Format:

42	A0	Immediate$_{16}$	Immediate Mode
42	AA		Accumulator Mode

3/6 clock cycles (4 + 2 memory accesses)

The task register (TR) is updated by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode / execute the instruction
STORE1[1]	set stack segment
STORE2[1]	push TR[15:8]
STORE2[1]	push TR[7:0]

1 this state is used only when the core isn't configured for back-links

Sample:

```
; Since fork causes a task switch, the original task may return to the
; instruction following the fork. Which task is actually running can be
; determined from the task register with the TTA instruction.

       SEI
       FORK   #1
       TTA
       CMP           #1
       BNE           .0001
       PLA          ; get the return task off the stack into .acc
       LDX    #$7000; set a new stack pointer
       TXS
       PHA          ; copy return tid back to stack
       CLI          ; safe for interrupts now
       ; initialize the registers for task #1 as desired
       ; and perform task #1 code
       ...
       BRA           .0002
.0001:
       CLI
       ; continue with the original task's code
       ...
.0002:

; It is more likely that FORK be called using the accumulator as a parameter.
```

INF

The INF instruction can be used to return general information about the processor including the contents of task registers, the core number, version number, and segment descriptor table entries.

Bits 4 to 15 of the .X index register indicate which context to return information for. Bits 0 to 3 of the .X index register indicate which field of information to return. Bits 0 to 11 of the .Y index register determine which segment descriptor table entry to return. The .Y register need be set only when it is desired to return information about a segment. The information is returned in the accumulator register.

X[3:0]	Information returned	
0	CS register	
1	DS register	
2	SS register	
3	Program counter and program bank	
4	accumulator	
5	.X index register	
6	.Y index register	
7	SP – stack pointer	
8	SR,SRX – status register and extended status register	
9	DBR – data bank register	
10	DPR – direct page register	
11	back link	
12..15	reserved	

If bits 4 to 15 of the .X register are all ones, then INF returns global information about the core.

Global Information Returned:

X[3:0]	Information returned	
0	core number	
1	core version	
2		

3		
4		
5		
6		
7		
8		
9		
10		
11		
12	segment base address	
13	bits 4 to 11 are segment access rights bits 0 to 3 are the segment size field	
14		
15	processor fault status bits 0 to 3 are segmentation faults	

Opcode Format (2 bytes)

42	4A

4 clock cycles

N and Z flags are affected.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode / select context reg
INF1	obtain info, select original context reg

INX4

Increment the .X index register by four. This instruction is similar to the INX instruction except that it increments by four rather than by one. With a 32 bit word size for most registers arrays are often 32 bits (four bytes). Indexing into word arrays requires adjusting the index by four.

Opcode Format (2 bytes)

42	E8

3 clock cycles

N and Z flags are affected.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode / execute the instruction

INY4

Increment the .Y index register by four. This instruction is similar to the INY instruction except it increments by four rather than by one. With a 32 bit word size for most registers arrays are often 32 bits (four bytes). Indexing into word arrays requires adjusting the index by four.

Opcode Format (2 bytes)

42	C8

3 clock cycles

N and Z flags are affected.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode / execute the instruction

IOS:

IOS: - forces the segment value to $FFD00000 during an address calculation, an address range reserved for I/O. The segment limit is set to 1MB for I/O. The segment value is a core parameter, which has $FFD00000 as the default. This allows shorter addressing modes to be used to access the I/O. It also avoids the problem of how to find the I/O address when the data segment is in use. I/O addresses are at fixed physical locations. Modifying the data segment to be non-zero means that the I/O addresses are no longer available at the same memory locations. Without using a pre-determined segment for I/O, the I/O addresses would have to be calculated for each data segment in use.

Interrupts are not allowed between this prefix and the following instruction.

Opcode Format (2 bytes)

42	7B

2 clock cycles

No flags are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode the following instruction
….	states for following instruction

JCF

JCF – Jump to context far routine long allows specification of a new context when jumping to a target address. The 24 bit offset field is loaded into the program counter and program bank. The 16 bit selector field is used to lookup a segment value which is then loaded into the code segment. The specified context register is then used to set the data segment and other registers. Registers .A, .X, .Y, and flags may copied from the current context to the new one if the preserve (P) bit is set in the instruction. This allows parameter passing to the routine in the new context. Up to 16 bytes may be popped off the stack of the caller and placed onto the context's stack, allowing parameters to be passed on the stack.

If the core is not configured to use back-links:

The current task register is pushed onto the stack of the called context after any parameters are copied.

Return from a context routine using the RTC instruction.

The intended use for this instruction is in synchronous context switching where the called context will execute then return to the invoking context in a manner analogous to a subroutine call. The called context should not be asynchronous running.

The appeal of a context based routine is that the data segment and stack pointer may be private to the context routine and these registers are switched automatically by the context routine call. Calling a context based routine is also faster than a subroutine call as it is register based rather than memory based.

It is envisioned that most of the time there will be only a single code segment associated with a context. In that case one of the other context call instructions can be used (JCR, JCL, JCI).

It is not possible to jump to the current context. An attempt to do so will cause the instruction to be treated like a NOP operation.

See also the JCR instruction.

Opcode Format (10 bytes)

42	82	Offset$_{24}$	Selector$_{16}$	Context#$_{16}$	P	~$_2$	Immed$_5$

7 clock cycles + 1 + 2 per bytes copied on stack + 2 more memory accesses

Machine States:

The JCF instruction first copies the specified number of bytes from the stack into an internal buffer. The stack pointer is incremented by the number of bytes copied. Next the task registers are switched including the stack pointer. Then the internal buffer is copied back to the stack. The stack pointer is decremented by the number of bytes copied to the stack. Finally the previous value of the task register is pushed onto the stack.

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode / save register context
LOAD1	set segment for addressing
LOAD2	only if copying stack parameters
…	repeats for each byte copied
TSK1	load registers from context
TSK2	setup code segment
TSK3	setup data segment
TSK4	setup stack segment
STORE2	only if copying stack parameters
…	repeats for each byte copied
STORE2[1]	push old TR[15:8]
STORE2[1]	push old TR[7:0]

1 Only if the core is not configured for back-links

JCI

JCI – Jump to context routine indirect allows specification of a new context when jumping to a target address. The lower 24 bits of the accumulator are loaded into the program counter and program bank. The upper eight bits of the accumulator identify the context register. The specified context register is then used to set the code segment, data segment and other registers. Registers .A, .X, .Y, and flags are copied from the current context to the new one. This allows parameter passing to the routine in the new context. Return from a context routine using the RTC instruction.

If the core is not configured for back-links:

The current task register is pushed onto the stack of the new context.

This instruction allows access to only the first 256 context registers. If a context register greater than 255 is required then the JCL instruction must be used.

The intended use for this instruction is in synchronous context switching where the called context will execute then return to the invoking context in a manner analogous to a subroutine call. The called context should not be asynchronous running.

The appeal of a context based routine is that the data segment and stack pointer may be private to the context routine and these registers are switched automatically by the context routine call. Calling a context based routine is also faster than a subroutine call as it is register based rather than memory based.

It is not possible to jump to the current context. An attempt to do so will cause the instruction to be treated like a NOP operation.

Opcode Format (2 bytes)

42	80

7 or 10 clock cycles (8 + 2 memory accesses)

Machine States:

IFETCH	Fetch the instruction

DECODE	Decode the page 2 prefix
DECODE	Decode / save register context
TSK1	load registers from context
TSK2	setup code segment
TSK3	setup data segment
TSK4	setup stack segment
STORE1[1]	set stack segment
STORE2[1]	push old TR[15:8]
STORE2[1]	push old TR[7:0]

1 Only if the core is configured to use the stack and not back-links.

Sample Code:

```
; Task #10 is an interpretive task
; How to implement an interpreter branch operation
       TSK           #10                 ; get the branch displacement
       AAX                               ; add .A and .X
       ORA           #$0A000000    ; select context register #10 (in high eight bits of acc).
       JCI                               ; indirect jump to the context to set PC (JCI [acc])
       BRA           NEXT2         ; the JCI context switch will cause the next instruction fetch
                                         ; so we can skip over the TSK #10
```

JCL

JCL – Jump to context routine long allows specification of a new context when jumping to a target address. The 24 bit offset field is loaded into the program counter and program bank. The specified context register is then used to set the code segment, data segment and other registers. Registers .A, .X, .Y, and flags may copied from the current context to the new one if the preserve (P) bit is set in the instruction. This allows parameter passing to the routine in the new context. Up to 16 bytes may be popped off the stack of the caller and placed onto the context's stack, allowing parameters to be passed on the stack.

If the core is not configured to use back-links:

> The current task register is pushed onto the stack of the called context after any parameters are copied.

Return from a context routine using the RTC instruction.

The intended use for this instruction is in synchronous context switching where the called context will execute then return to the invoking context in a manner analogous to a subroutine call. The called context should not be asynchronous running.

The appeal of a context based routine is that the data segment and stack pointer may be private to the context routine and these registers are switched automatically by the context routine call. Calling a context based routine is also faster than a subroutine call as it is register based rather than memory based.

This instruction cannot be used to call a routine in the current context. Attempting to do so will cause the instruction to act like a NOP operation.

See also the JCR instruction.

Opcode Format (8 bytes)

42	82	Offset$_{24}$	Context#$_{16}$	P	~$_2$	Immed$_5$

7 clock cycles + 1 + 2 per bytes copied on stack + 2 more memory accesses

Machine States:

The JCL instruction first copies the specified number of bytes from the stack into an internal buffer. The stack pointer is incremented by the number of bytes copied. Next the task registers are switched including the stack pointer. Then the internal buffer is copied back to the stack. The stack pointer is decremented by the number of bytes copied to the stack. Finally the previous value of the task register is pushed onto the stack.

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode / save register context
LOAD1	set segment for addressing
LOAD2	only if copying stack parameters
…	repeats for each byte copied
TSK1	load registers from context
TSK2	setup code segment
TSK3	setup data segment
TSK4	setup stack segment
STORE2	only if copying stack parameters
…	repeats for each byte copied
STORE2[1]	push old TR[15:8]
STORE2[1]	push old TR[7:0]

1 Only if the core is not configured for back-links

JCR

JCR – Jump to context routine allows specification of a new context when jumping to a target address. The 16 bit offset field is loaded into the program counter. The program bank is set to zero. The specified context register is then used to set the code segment, data segment and other registers. Registers .A, .X, .Y, and flags are copied from the current context to the new one. This allows parameter passing to the routine in the new context. Return from a context routine using the RTC instruction.

If the core is not configured to use back-links:

> The previous value of the task register is pushed onto the stack of the new context.

Much of the time it will be desirable to implement a jump table in the called context. If this jump table is placed near the start of the code segment, then this short form addressing instruction can be used. Otherwise if a 24 bit address specification is required the JCL instruction can be used.

To conserve memory this instruction allows access to only the first 256 context registers. If a context register greater than 255 is required then the JCL instruction must be used.

The intended use for this instruction is in synchronous context switching where the called context will execute then return to the invoking context in a manner analogous to a subroutine call. The called context should not be asynchronous running.

The appeal of a context based routine is that the data segment and stack pointer may be private to the context routine and these registers are switched automatically by the context routine call. Calling a context based routine is also faster than a subroutine call as it is register based rather than memory based.

This instruction cannot be used to call a routine in the current context. Attempting to do so will cause the instruction to act like a NOP operation.

Opcode Format (5 bytes)

42	20	Offset_{16}	$\text{Context\#}_{8}$

7 clock cycles

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode / save register context
TSK1	load registers from context
TSK2	setup cs segment
TSK3	setup ds segment
TSK4	setup ss segment
STORE1[1]	set segment
STORE2[1]	push old TR[15:8]
STORE2[1]	push old TR[7:0]

1 Only if the core is not configured to use back-links.

JMF

JMF – Jump Far allows specification of a new segment when jumping to a target address. The 24 bit offset field is loaded into the program counter and program bank. The 16 bit selector field is used to lookup a segment value which is then loaded into the code segment register.

Opcode Format (7 bytes)

42	5C	Offset$_{24}$	Selector$_{16}$

4 clock cycles

No registers are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode / Execute the instruction
JMF1	set code base register

JSF

JSF – Jump to Subroutine Far, allows specification of a new segment when calling a subroutine. Both the code selector and program counter value are pushed onto the stack. A total of five bytes are pushed onto the stack.

Note that it is much faster to switch tasks with the TSK instruction than it is to jump and return from a far subroutine. In many cases when a accessing a new code segment is desired, what is really desired is to invoke a different task. For instance the operating system may be a 'far' distance away from code that is running. Rather than doing a far jump to operating system code, a task switch can be done instead. Synchronous calls to the operating system could be implemented with the JCR instruction.

Opcode Format (7 bytes)

42	22	Offset_{24}	Selector_{16}

10 clock cycles (5 + 5 memory accesses)

No flags are affected by this instruction.

Machine States:

The store operation stores from higher to lower memory addresses as values are being pushed onto the stack.

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode / Execute the instruction
STORE1	set segment for addressing
STORE2	store CS[15:8]
STORE2	store CS[7:0]
STORE2	store PC[23:16]
STORE2	store PC[15:8]
STORE2	store PC[7:0]
JMF1	set new code segment

LDT

The LDT (load task context register) instruction has two forms of addressing. The first, indexed addressing form allows an entry from a table to be loaded. The indexed form shifts the .X register left five times before using it to index into a table as table entries are 32 bytes in size. The .X register also indicates which task context register to load. The second form of the instruction allows loading a context register from memory without indexing; however the .X register still indicates which context register to load. Extra-long address mode is used to allow the table to be placed anywhere in memory.

Opcode Format (6 bytes)

42	4C	Address_{32}	Indexed by .X
42	6C	Address_{32}	non-indexed

Memory Layout for LDT instruction:

```
TaskStartTbl:
       .WORD  0                     ; CS
       .WORD  0                     ; DS
       .WORD  0                     ; SS
       .WORD  Task0          ; PC
       .BYTE  Task0>>16
       .WORD  0                     ; acc
       .WORD  0
       .WORD  0                     ; x
       .WORD  0
       .WORD  0                     ; y
       .WORD  0
       .WORD  $3FFF          ; sp
       .WORD  0
       .BYTE  0                     ; SR
       .BYTE  1                     ; SR extension
       .BYTE  0                     ; DB
       .WORD  0                     ; DPR
       .WORD  0
```

The LDT instruction can take a large number (44) clock cycles to execute. It has to load 32 bytes from memory into the context register. Note that in many cases the entire tasking system can be setup before interrupts are enabled. Many embedded applications do not require dynamic creation of tasks. So the LDT instruction does not necessarily impact interrupt latencies. IF interrupt latency is a concern then the FORK instruction which has a much lower latency could be used to start a task. However the FORK instruction does not set new register values.

There is a dead cycle between each word (4 bytes) loaded from memory so that the instruction doesn't hog the bus too much.

This instruction is not interruptible.

44 clock cycles (12 + 32 memory accesses)

Machine States:

IFETCH	Fetch the instruction	
DECODE	Decode the page 2 prefix	
DECODE	Decode the instruction	
LDT1		
LOAD2	load LSB from memory (CS[7:0])	
LOAD2		
LOAD2		
LOAD2	load MSB from memory (CS[31:24])	
LDT1		Repeats six more times to load the remaining registers.
LOAD2	load LSB from memory (DS[7:0])	
LOAD2		
LOAD2		
LOAD2	load LSB from memory (DS[31:24])	
…		
LDT1	update task context registers	

LLA:

LLA – The LLA prefix causes a load of the linear address of the following instruction into the accumulator. The linear address is the address after segmentation has been applied. The memory operation of the following instruction is not performed, the operation is aborted once the address has been calculated. The total number number of clock cycles is less than that required for the operation.

Opcode Format (2 bytes)

42	FA

2 clock cycles

No flags are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode the following instruction
….	states for following instruction

MUL

MUL – Performs an unsigned multiply of the .A and .X registers and leaves the product in the accumulator and .X register. When multiplying byte registers the 16 bit product is available in the .A (low order) and .B (higher order) registers. Multiply respects the register size settings. Higher order product bits are available with the XBA and XBAW instruction when operating in 8/16 bit mode.

Bits 0 to 31 of the product are placed in the accumulator.

Bits 32 to 63 of the product are placed into the .X register.

Opcode Format (2 bytes)

42	2A

3 clock cycles

The N flag is set to bit 31 of the result. The Z flag is set if the result is zero. The V flag is set if the high order 32 bits of the product are non-zero.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode / execute the instruction

PHCS

PHCS – pushes the code selector on the stack. Two bytes are pushed onto the stack.

It is sometimes desirable to transfer the CS register to the DS register. For instance, in order to write to the code segment. This can be done by pushing the CS register then popping the DS register. Pushing the CS register may also be used in synthesizing a far subroutine call. The CS register cannot be pulled from the stack. In order to change the CS register use a far subroutine call or jump instruction.

Opcode Format (2 bytes)

42	4B

6 clock cycles (4 + 2 memory accesses)

No flags are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode the instruction
STORE1	set segment for addressing
STORE2	store MSB to memory
STORE2	store LSB to memory

PHDS

PHDS – pushes the data selector on the stack. Two bytes are pushed onto the stack. This instruction is useful when passing pointers for intersegment data access.

Opcode Format (2 bytes)

42	0B

6 clock cycles (4 + 2 memory accesses)

No flags are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode the instruction
STORE1	set segment for addressing
STORE2	store LSB to memory
STORE2	store MSB to memory

PHP

PHP – pushes the processor status register onto the stack.

In native 32 bit mode both the status register and extended status register are pushed. In 65C816/65c02 compatible modes only the status register is pushed.

Opcode Format (1 bytes)

08

5 clock cycles (3 + 2 memory accesses)

No flags are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the instruction
STORE1	set segment for addressing
STORE2[1]	store SRX
STORE2	store SR

1 only in native mode

PLDS

PLDS – pulls the data segment from the stack. Two bytes are pulled from the stack.

Pulling the DS from the stack is one of two ways that the DS register can be set. The other way to set the DS register is to define it in a task start-up record then use the LDT instruction to load the task context.

Opcode Format (2 bytes)

42	2B

6 clock cycles (4 + 2 memory accesses)

No flags are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode the instruction
LOAD1	setup segment
LOAD2	load LSB from memory
LOAD2	load MSB from memory

PLP

PLP – pulls the processor status register from the stack.

In native 32 bit mode both the status register and extended status register are pulled. In 65C816/65c02 compatible modes only the status register is pulled.

Opcode Format (1 bytes)

28

5 clock cycles (3 + 2 memory accesses)

No flags are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the instruction
LOAD1	set segment for addressing
LOAD2	load SR
LOAD2[1]	load SRX

1 only in native mode

REP

REP – resets bits in the processor’s status register.

The REP instruction has an additional opcode which allows setting bits in the extended status register.

Status register and extended status register bits are reset according to the immediate pattern specified. A ‘one’ bit in the immediate constant causes the status register or extended status register bit to be reset to zero. A zero bit in the immediate constant causes no change for the corresponding bit in the status register.

Opcode Format (4 bytes)

42	C2	Immediate$_{16}$

3 clock cycles

Multiple flags may be affected by this instruction depending on the immediate constant.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	decode / execute instruction

RTC

The RTC instruction (return from context routine) switches contexts from the current back to the invoking context. This is accomplished by popping the return context number from the stack or if the core is configured to use back-links by reading the back-link. This instruction also copies the .A, .X, .Y and flags registers to the returned context. The operation of this instruction is almost identical to the RTT instruction with the exception that register values are returned. Additionally, up to 255 bytes may be popped off the stack.

This instruction paired with the JCR instruction allows a context to be treated like a subroutine. The instruction is used with synchronous context calls.

Opcode Format (3 bytes)

42	40	Immed_8

7 or 10 clock cycles (8 + 2 memory accesses)

No flags are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode page 2 prefix
DECODE	Decode / execute –save register set
LOAD1[1]	set segment
LOAD2[1]	load TR[7:0]
LOAD2[1]	load TR[15:8]
TASK1	load the register set
TASK2	setup code segment
TASK3	setup data segment
TASK4	setup stack segment

1 only if the core is not configured to use back-links

RTF

The RTF instruction performs a far return from subroutine operation. This is similar to a long subroutine return operation (RTL) except that the code segment is loaded from the stack in addition to the program counter and program bank. In addition the stack pointer may be incremented by an amount specified by the instruction in order to pop arguments off the stack.

Opcode Format (3 bytes)

42	6B	Immed$_8$

12 clock cycles (5 + 7 memory accesses)

No flags are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode page 2 prefix
DECODE	Decode / execute –save register set
LOAD1	set segment for addressing
LOAD2	load PC[7:0]
LOAD2	load PC[15:8]
LOAD2	load PC[23:16]
LOAD2	load CS[7:0]
LOAD2	load CS[15:8]
LOAD2	load CS[23:16]
LOAD2	load CS[31:24]
RTS1	increment PC / add to SP

RTI

The operation of this instruction has been modified for native mode. In native mode the RTI instruction (return from interrupt) switches tasks from the current task back to the interrupted task. This is accomplished by popping the return context number from the stack or reading the back-link field depending on how the core is configured. It has the same effect as the RTT instruction and either instruction may be used to return from an interrupt task. In emulation modes this instruction works in manner compatible with the 65c02/65c816 cores.

The RTI instruction is one byte shorter and one clock cycle faster than the RTT instruction. However this is only valid in native mode. The RTT instruction may be used in emulation modes as well as native mode.

Opcode Format (1 bytes)

40

6 or 9 clock cycles 7 + 2 memory accesses (native mode operation)

No flags are affected by this instruction.

Machine States (native mode):

IFETCH	Fetch the instruction
DECODE	Decode / execute –save register set
LOAD1[1]	set segment
LOAD2[1]	load TR[7:0]
LOAD2[1]	load TR[15:8]
TASK1	load the register set
TASK2	setup code segment
TASK3	setup data segment
TASK4	setup stack segment

1 only if the core is not configured to use back-links

RTL

This is an additional form for the existing RTL instruction. The RTL instruction performs a long return from subroutine operation. A twenty-four bit value is popped from the stack and placed into the program counter and program bank registers. In addition the stack pointer may be incremented by an amount specified by the instruction in order to pop arguments off the stack.

Opcode Format (3 bytes)

42	68	Immed$_8$

8 clock cycles (5 + 3 memory accesses)

No flags are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode page 2 prefix
DECODE	Decode / execute –save register set
LOAD1	set segment for addressing
LOAD2	load PC[7:0]
LOAD2	load PC[15:8]
LOAD2	load PC[23:16]
RTS1	increment PC / add to SP

RTS

This is an additional form for the existing RTS instruction. The RTS instruction performs a short return from subroutine operation. A sixteen bit value is popped from the stack and placed into the program counter. The program bank is not affected. In addition the stack pointer may be incremented by an amount specified by the instruction in order to pop arguments off the stack.

Opcode Format (3 bytes)

42	C0	Immed$_8$

7 clock cycles (5 + 2 memory accesses)

No flags are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode page 2 prefix
DECODE	Decode / execute –save register set
LOAD1	set segment for addressing
LOAD2	load PC[7:0]
LOAD2	load PC[15:8]
RTS1	increment PC / add to SP

RTT

The RTT instruction (return from task) switches tasks from the current task back to the invoking task. This is accomplished by popping the return context number from the stack or by reading the back-link field. A full context switch takes place; all registers are restored from the context returned to. A similar operation is the RTC instruction which allows values in registers to be passed back to the invoking task.

Opcode Format (2 bytes)

42	60

7 or 10 clock cycles (8 + 2 memory accesses)

All registers are restored by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode page 2 prefix
DECODE	Decode / execute –save register set
LOAD1[1]	set stack segment
LOAD2[1]	pop TR[7:0]
LOAD2[1]	pop TR[15:8]
TASK1	load the register set
TASK2	setup code segment
TASK3	setup data segment
TASK4	setup stack segment

1 only if the core is not configured to use back-links

SDU

SDU stands for Segment Descriptor Update. An SDU operation puts values into one of the segment descriptor table entries. The accumulator contains the segment base value and the .X register contains the access rights and segment size values. The .Y register is used to index into the table. There are 4096 table entries. The contents of the table is undefined after reset. The contents of an entry should be defined before it is accessed with a selector.

	43 36	35 32	31 0
0	ACR_8	$Size_4$	$Base_{31..0}$
1	ACR_8	$Size_4$	$Base_{31..0}$
…			
4095	ACR_8	$Size_4$	$Base_{31..0}$

The segment table must be updated before segments can be used.

Opcode Format (2 bytes)

42	BA

3 clock cycles

No flags are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode/execute the instruction

Sample Code:

```
       ; Setup all the segment descriptors
       ; This must be done before the stack can be accessed, and
       ; before subroutine calls and task switching can be made.
       LDA           #0
       XBAW
       LDA           #0
       LDX           #$98D         ; present, executable, writeable, max size
       TAY
.0003:
       SDU                                ; update segment descriptor
       INY
       CPY           #4096
       BNE           .0003
```

SEG:

SEG: - forces use of the specified segment value for address calculations. The prefix with segment value is six bytes. No interrupt is allowed to occur between the prefix and the following instruction.

Opcode Format (6 bytes)

42	3B	Immediate_{32}

2 clock cycle

No flags are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode / execute the prefix
DECODE	Decode the following instruction
…	continue with states for the following instruction

SEG0:

SEG0: - forces the segment value zero to be used during address calculations. This is only a two byte prefix. Using this prefix effectively allows access to physical addresses. It can be useful when accessing system components which are at fixed locations in memory (video frame buffer). No interrupt is allowed to occur between the prefix and the following instruction. This is an alias for the ZS: prefix.

Opcode Format (2 bytes)

42	5B

2 clock cycle

No flags are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode / execute the prefix
DECODE	Decode the following instruction
…	continue with states for the following instruction

SEP

SEP –sets bits in the processor's status register.

The SEP instruction has an additional opcode which allows setting bits in the extended status register.

Status register and extended status register bits are set according to the immediate pattern specified. A 'one' bit in the immediate constant causes the status register or extended status register bit to be set to one. A zero bit in the immediate constant causes no change for the corresponding bit in the status register.

Opcode Format (4 bytes)

42	E2	Immediate$_{16}$

3 clock cycles

Multiple flags may be affected by this instruction depending on the immediate constant.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	decode / execute instruction

SS:

This is a segment override prefix indicating to use the SS register in calculating a data address rather than the DS register. This prefix is treated as part of the current instruction. No interrupt will be allowed between the prefix and following instruction.

This prefix is typically used to access values which are placed into the stack segment.

Opcode Format (2 bytes)

42	6A

2 clock cycle

No flags are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode the following instruction
….	states for following instruction

TASS

Transfer accumulator to stack segment selector. All sixteen bits of the accumulator are transferred. Setting the selector causes a lookup from the segment descriptor table. The values looked-up are cached in an internal register for use with one of the stack instructions.

Opcode Format (2 bytes)

42	5A

4 clock cycles

No flags are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 opcode
DECODE	Decode and execute the instruction
TASS1	lookup segment values

TJ:

The TJ: (standing for Task Jump) prefix modifies the following task switch operation so that it does not store the return task number on the stack. It effectively turns the task call into a task jump operation. However if this prefix is used the RTT and RTC instructions cannot automatically determine which task to return to. The TJ prefix may allow faster task switching in some systems as it eliminates the memory accesses from the task switch.

This prefix is only useable when the core is not configured to use back-link and uses the stack instead.

Opcode Format (2 bytes)

42	CB

2 clock cycle

No flags are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode / execute the prefix
DECODE	Decode the following instruction
…	continue with states for the following instruction

TSK

The TSK instruction is similar to a subroutine call except that it invokes another task rather than a subroutine. The current task number is stored either on the stack or in a back-link field depending on the core configuration. This allows a task switch back to the original invoking task when the task is finished running via the RTT (return from task) instruction. However, if the task being invoked is an interpretive task then the return task number is not stored on the stack for performance reasons. Instead the core internally tracks which task to return to.

If the core is not configured to use back-links:

When the TSK instruction is executed, it stores the current task number on the stack of new task.

An attempt to switch to the same task as the one that is already running is ignored. In that case the instruction executes in 3 clock cycles.

The context register must have been previously set by the LDT instruction, or by the FORK instruction.

The TSK instruction first stores all the program visible registers in the current context register, then loads all the program visible registers from the context register being switched to.

TSK sets the task register (TR) so that the currently running task may be identified by the processor.

The task system allows the core operating mode to be switched at task switch time.

Opcode Format:

42	A2	Immediate$_{16}$	Immediate Mode
42	3A		Accumulator Mode

7 or 10 clock cycles (8 + 2 memory accesses) or 7 clock cycles if switching to an interpretive task.

All registers are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction

DECODE	Decode the page 2 opcode
DECODE	Decode and execute the instruction (save current task state)
TSK1	load new task state
TSK2	setup code segment
TSK3	setup data segment
TSK4	setup stack segment
STORE1[1]	set stack segment
STORE2[1]	push old TR[15:8]
STORE2[1]	push old TR[7:0]

1 only if the core is not configured to use back-links

TSSA

Transfer stack segment selector to accumulator. All sixteen bits of the accumulator are loaded.

Opcode Format (2 bytes)

42	7A

3 clock cycles

N and Z flags are affected by this instruction. N is set to bit 15 of the SS register. Z is set if the SS register is zero.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 opcode
DECODE	Decode and execute the instruction

TTA

Transfer task register (TR) to accumulator. This instruction allows a program to determine which task is active. Note that there is no instruction to transfer to the task register. Transfers to the task register are accomplished by switching the task with the TSK instruction.

When the accumulator width is set to eight bits, this instruction transfers a full 16 bits to the accumulator. The ‘B’ register is set to bits 8 to 15 of the task register.

Opcode Format (2 bytes)

42	1A

3 clock cycles

N and Z flags are affected by this instruction. N is set to bit 15 of the task register. Z is set if the task register is zero.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 opcode
DECODE	Decode and execute the instruction

XBAW

Exchange high order and low order word of accumulator. Bits 0 to 15 are exchanged with bits 16 to 31. Operation of this instruction is similar to the XBA instruction. This instruction can be used to obtain access to bits 16 to 31 of the multiplier product. This instruction combined with the XBA instruction can be used to switch the byte order around.

Opcode Format (2 bytes)

42	EB

3 clock cycles

N is set to bit 15 of the result. Z is set if bits 0 to 15 of the result are zero.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 opcode
DECODE	Decode and execute the instruction

ZS:

ZS: - forces the segment value zero to be used during address calculations. This is only a two byte prefix. Using this prefix effectively allows access to physical addresses. It can be useful when accessing system components which are at fixed locations in memory (video frame buffer). No interrupt is allowed to occur between the prefix and the following instruction.

Opcode Format (2 bytes)

42	5B

Since this instruction eliminates the instruction fetch for the following instruction, it reduces the cycle count of the following instruction by one. This means that prefix is executed in two clock cycles.

2 clock cycle

No flags are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the prefix
DECODE	Decode the following instruction
…	continue with states for the following instruction

Core Parameters

Parameter	Default value	What it does
EXTRA_LONG_BRANCHES	1	Causes the core to generate hardware to support extra-long branching for the general purpose branch instructions.
IO_SEGMENT	$FFD00000	The segment value used when the IOS: prefix is present in the instruction stream
PC24	1	Causes the program counter to be a true 24 bit program counter (increments automatically across banks). Set to zero to force a 16 bit program counter which wraps around at a bank boundary. Setting this value to zero may generate slightly less hardware and is consistent with the 65c816.
POPBF	0	If set to one, allows popping the break flag from the stack. The default setting is consistent with 65xxx operation.
TASK_VECTORING	1	Controls whether or not the core uses task id’s for interrupt vectors instead of addresses. This parameter must be set to zero to be consistent with 65xxx behaviour.

Configuration Defines

	Default Value	What it does
SUPPORT_TASK	1	Causes the core to include hardware for task switching. Un-defining this symbol may result in a slightly smaller core (10%).
TASK_MEM	512	Specifies the number of entries in the task context array. This should be a power of two. Increasing this value will increase the amount of RAM used. Note that reducing this value may not result in lower RAM usage as RAM resources typically have a minimum size.
TASK_MEM_ABIT	8	The bit number of the most significant bit needed to access the task memory. This parameter will need to be changed to be consistent with the TASM_MEM parameter.
TASK_BL	1	Causes the core to use back-links for task switching rather than storing the task number on the stack.
SUPPORT_SEG	1	Causes the core to implement the segmentation model. Un-defining this symbol removes the segment registers and associated instructions from the core resulting in a slightly smaller core.
ICACHE_4K	1	Causes the core to use a 4kB instruction cache.
ICACHE_16K	0	Causes the core to use a 16kB instruction cache. Cannot be defined at the same time as ICACHE_4K.
SUPPORT_BCD	1	Causes the core to include logic to support BCD addition and subtraction. BCD support is necessary to remain compatible with the 65xxx series.
SUPPORT_NEW_INSN	1	Causes the core to include new instructions. Commenting out this definition will significantly reduce the size of the core; however instructions supporting new core features will not be available.
SUPPORT_INTMD	1	Causes the core to include logic for interpretive operating mode. Commenting out this definition will marginally reduce the size of the core.

I/O Ports

	In/Out	Width		
corenum	I	32	core number, if left unassigned zero is assumed. This input is reflected by the INF instruction.	
rst	I	1	reset, active low – resets the core	
clk	I	1	input clock, this clock is not directly used to clock the core. Instead it is gated internally to allow the core clock to be stopped with the STP instruction.	
clko	O	1	output clock. – this is the input clock gated and drives the core. this clock may stop if the STP instruction is executed.	
phi11	O	1	Phase one of the input clock divided by 32. This is a low speed clock output designed to drive peripherals.	
phi12	O	1	Phase two of the input clock divided by 32. This is a low speed clock output designed to drive peripherals.	
phi81	O	1	Phase one of the input clock divided by 8. This is a low speed clock output designed to drive peripherals / low speed memory.	
phi82	O	1	Phase two of the input clock divided by 8. This is a low speed clock output designed to drive peripherals / low speed memory.	
nmi	I	1	active low input for non-maskable interrupt	
irq	I	1	active low input for interrupt	
abort	I	1	active low input for abort interrupt	
e	O	1	‘e’ flag indicator reflects the status of the emulation flag	
mx	O	1	m and x status output ‘m’ when clock is high, otherwise ‘x’	
rdy	I	1	active high ready input, pull low to insert wait states	
be	I	1	bus enable, tri-states the address, data, and r/w lines when active	
vpa	O	1	valid program address, set high during an instruction cache line fetch	
vda	O	1	valid data address, set high during a data access, also set high during the first cycle of an instruction cache line fetch	
mlb	O	1	memory lock, active high	
vpb	O	1	vector pull, set high during a vector fetch	
rw	O	1	read/write, active high for read, low for write cycle	
ad	O	32	address bus	
db	I/O	8	data bus , input for read cycles, output for write cycles	

Opcode Map

Opcode Map – 8 bit mode W65C816 compatible

🟦	= W65C816S instructions

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
0-	BRK	ORA (d,x)	COP	ORA d,s	TSB d	ORA d	ASL d	ORA [d]	PHP	OR #i8	ASL acc	PHD	TSB abs	ORA abs	ASL abs	ORA AL
1-	BPL disp	ORA (d),y	ORA (d)	ORA (d,s),y	TRB d	OR d,x	ASL d,x	ORA [d],y	CLC	OR abs,y	INA	TAS	TRB abs	ORA abs,x	ASL abs,x	ORA AL,x
2-	JSR abs	AND (d,x)	JSL abs24	AND d,s	BIT d	AND d	ROL d	AND [d]	PLP	AND #i8	ROL acc	PLD	BIT abs	AND abs	ROL abs	AND AL
3-	BMI disp	AND (d),y	AND (d)	AND (d,s),y	BIT d,x	AND d,x	ROL d,x	AND [d],y	SEC	AND abs,y	DEA	TSA	BIT abs,x	AND abs,x	ROL abs,x	AND AL,x
4-	RTI	EOR (d,x)	WDM	EOR d,s	MVP	EOR d	LSR d	EOR [d]	PHA	EOR #i8	LSR acc	PHK	JMP abs	EOR abs	LSR abs	EOR AL
5-	BVC disp	EOR (d),y	EOR (d)	EOR (d,s),y	MVN	EOR d,x	LSR d,x	EOR [d],y	CLI	EOR abs,y	PHY	TCD	JML abs24	EOR abs,x	LSR abs,x	EOR AL,x
6-	RTS	ADC (d,x)	PER	ADC d,s	STZ d	ADC d	ROR d	ADC [d]	PLA	ADC #i8	ROR acc	RTL	JMP (abs)	ADC abs	ROR abs	ADC AL
7-	BVS disp	ADC (d),y	ADC (d)	ADC (d,s),y	STZ d,x	ADC d,x	ROR d,x	ADC [d],y	SEI	ADC abs,y	PLY	TDC	JMP (abs,x)	ADC abs,x	ROR abs,x	ADC AL,x
8-	BRA disp	STA (d,x)	BRL disp	STA d,s	STY d	STA d	STX d	STA [d]	DEY	BIT #	TXA	PHB	STY abs	STA abs	STX abs	STA AL
9-	BCC disp	STA (d),y	STA (d)	STA (d,s),y	STY d,x	STA d,x	STX d,y	STA [d],y	TYA	STA abs,y	TXS	TXY	STZ abs	STA abs,x	STZ abs,x	STA AL,x
A-	LDY #i8	LDA (d,x)	LDX #i8	LDA d,s	LDY d	LDA d	LDX d	LDA [d]	TAY	LDA #i8	TAX	PLB	LDY abs	LDA abs	LDX abs	LDA AL
B-	BCS disp	LDA (d),y	LDA (d)	LDA (d,s),y	LDY d,x	LDA d,x	LDX d,y	LDA [d],y	CLV	LDA abs,y	TSX	TYX	LDY abs,x	LDA abs,x	LDX abs,x	LDA AL,x
C-	CPY #i8	CMP (d,x)	REP #	CMP d,s	CPY d	CMP d	DEC d	CMP [d]	INY	CMP #i8	DEX	WAI	CPY abs	CMP abs	DEC abs	CMP AL
D-	BNE disp	CMP (d),y	CMP (d)	CMP (d,s),y	PEI	CMP d,x	DEC d,r	CMP [d],y	CLD	CMP abs,y	PHX	STP	JML (a)	CMP abs,x	DEC abs,x	CMP AL,x
E-	CPX #i8	SBC(d,x)	SEP #	SBC d,s	CPX d	SUB d	INC d	SBC [d]	INX	SBC #i8	NOP	XBA	CPX abs	SBC abs	INC abs	SBC AL,
F-	BEQ disp	SBC (d),y	SBC(r)	SBC (d,s),y	PEA	SUB d,x	INC d,r	SBC [d],y	SED	SBC abs,y	PLX	XCE	JSR (abs,x)	SBC abs,x	INC abs,x	SBC AL,x

Opcode Map – Page 2 Opcodes

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
0-	BRK2	ORA {d,x}									ASR acc	PHDS	TSB xlabs	ORA xlabs	ASL xlabs	
1-	BGT disp	ORA {d},y	ORA {d}	ORA {d,s},y					CMC	OR xlabs,y	TTA	CS:	TRB xlabs	ORA xlabs,x	ASL xlabs,x	
2-	JCR	AND {d,x}	JSF seg:offs						BSR		MUL	PLDS	BIT xlabs	AND xlabs	ROL xlabs	
3-		AND {d},y	AND {d}	AND {d,s},y						AND xlabs,y	TSK acc	SEG:	BIT xlabs,x	AND xlabs,x	ROL xlabs,x	
4-	RTC	EOR {d,x}	WDM2		FIL				BSL		INF	PHCS	LDT xlabs,x	EOR xlabs	LSR xlabs	
5-		EOR {d},y	EOR {d}	EOR {d,s},y						EOR xlabs,y	TASS	ZS:	JMF seg:offs	EOR xlabs,x	LSR xlabs,x	
6-	RTT	ADC {d,x}	JCF						RTL #		SS:	RTF	LDT xlabs	ADC xlabs	ROR xlabs	
7-		ADC {d},y	ADC {d}	ADC {d,s},y						ADC xlabs,y	TSSA	IOS:	JML [xlabs,x]	ADC xlabs,x	ROR xlabs,x	
8-	JCI	STA {d,x}	JCL						DEY4		AAX	BYT:	STY xlabs	STA xlabs	STX xlabs	
9-		STA {d},y	STA {d}	STA {d,s},y						STA xlabs,y	WRD:	UBT:	STZ xlabs	STA xlabs,x	STZ xlabs,x	
A-	FORK #	LDA {d,x}	TSK #								FORK	HAF:	LDY xlabs	LDA xlabs	LDX xlabs	
B-	BLE disp	LDA {d},y	LDA {d}	LDA {d,s},y						LDA xlabs,y	SDU	UHF:	LDY xlabs,x	LDA xlabs,x	LDX xlabs,x	
C-	RTS #	CMP {d,x}	REP #						INY4		DEX4	TJ:	CPY xlabs	CMP xlabs	DEC xlabs	
D-		CMP {d},y	CMP {d}	CMP {d,s},y	PEA { }					CMP xlabs,y	FAR	CLK	JML [xlabs]	CMP xlabs,x	DEC xlabs,x	
E-	CACHE #	SBC{d,x}	SEP #						INX4	INC z,#	NOP2	XBAW	CPX xlabs	SBC xlabs	INC xlabs	
F-	PCHIST	SBC {d},y	SBC{d}	SBC {d,s},y	PEA xlabs					SBC xlabs,y	LLA:		JSL [xlabs,x]	SBC xlabs,x	INC xlabs,x	