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. New instructions have been added to support core functionality.

Programming Model

The programming model is compatible with the W65C816S programming model, with the addition of two 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	32	code segment	
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	32	data segment	
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:
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
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	%xxxxxx00	- the emulation mode is set to eight bit, both the 32 and 16 bit emulation flags are cleared
TR	$00	- the task register identifies which task is running. It is an internal register, set indirectly by the TSK instruction.
		-

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

New Registers

There are two new segment registers CS and DS standing for Code Segment and Data 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, two 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 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.

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.

Context Register Layout

263+ 256	255 224	223 192	191 168	167 136	135 104	103 72	71 40	39 32	31 24	23 16	15 0
Back Link	CS	DS	PC	ACC	.X	.Y	SP	SR	SRX	DB	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).

Memory Layout for LDT instruction:

```
TaskStartTbl:
       .WORD  0                    ; CS
       .WORD  0
       .WORD  0                    ; DS
       .WORD  0
       .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
```

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 two segment registers (code and data) and addresses are formed by a simple addition 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. 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:, SEG:, SEG0, IOS:)

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

The code segment may be set using the JMF, or JSF 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.

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 four 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 #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 task numbers are used 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.

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	

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).

Instruction Set Summary

BGE

BGE stands for branch greater or equal. This is a branch based on a signed comparison of two values. It takes the overflow flag into consideration as well as the negative and zero flags.

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

BGT

BGT stands for branch greater than. This is a branch based on a signed comparison of two values. It takes the overflow flag into consideration as well as the negative and zero flags.

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. It takes the overflow flag into consideration as well as the negative and zero flags.

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

BLT

BLT stands for branch less than. This is a branch based on a signed comparison of two values. It takes the overflow flag into consideration as well as the negative and zero flags.

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

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)

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.

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.

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. Similar to the DEX instruction except 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. Similar to the DEY instruction except 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

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.

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 clock cycles

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

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.

       FORK   #1
       TTA
```

```
       CMP           #1
       BNE           .0001
       ; initialize the registers for task #1 as desired
       ; and perform task #1 code
       ...
       BRA           .0002
.0001:
       ; continue with the original task's code
       ...
.0002:

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

INX4

Increment the .X index register by four. Similar to the INX instruction except 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. Similar to the INY instruction except 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 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

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.

Note that context routines cannot be re-entrant. Calling a context routine that has already been called will form a processing loop. This is because there is only a single back-link to the caller not a stack. Care must be taken not to call context routines in a re-entrant fashion. 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	$\sim_2$	$Immed_5$

4 clock cycles + 2 per bytes copied on stack

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode / save register context

LOAD2	only if copying stack parameters
…	repeats for each byte copied
TSK1	load registers from context
STORE2	only if copying stack parameters
…	repeats for each byte copied

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. Note that context routines cannot be re-entrant. Calling a context routine that has already been called will form a processing loop. This is because there is only a single back-link to the caller not a stack. Care must be taken not to call context routines in a re-entrant fashion. Return from a context routine using the RTC instruction.

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	$Offset_{16}$	$Context\#_{8}$

4 clock cycles

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode / save register context
TSK1	load registers from context

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 32 bit segment field is loaded into the code segment register. The special segment value $FFFFFFFF causes a switch to 8 bit emulation mode. The special segment value $FFFFFFFE causes a switch to 16 bit emulation mode.

Switching to an emulation mode zeros out the code and data segments and the upper portion of the index registers.

Opcode Format (9 bytes)

42	5C	$Offset_{24}$	$Segment_{32}$

3 clock cycles

IF switching modes, the m816, m832 flags are affected in the extended status register. The m and x bits are set to one

Machine States:

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

JSF

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

There is a dead cycle inserted between each memory access.

Opcode Format (9 bytes)

42	22	$Offset_{24}$	$Segment_{32}$

17 clock cycles

No flags are affected by this instruction.

LDT

The LDT 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.

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
       .WORD  0                    ; DS
       .WORD  0
       .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
```

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. 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.

44 clock cycles

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	

PHCS

PHCS – pushes the code segment on the stack. Four bytes are pushed onto the stack.

Bytes are written ‘back-to-back’ without dead cycles in between.

Opcode Format (2 bytes)

42	4B

7 clock cycles

No flags are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode the instruction
STORE2	store MSB to memory
STORE2	
STORE2	
STORE2	store LSB to memory

PHDS

PHDS – pushes the data segment on the stack. Four bytes are pushed onto the stack.

Bytes are written ‘back-to-back’ without dead cycles in between.

Opcode Format (2 bytes)

42	0B

7 clock cycles

No flags are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode the page 2 prefix
DECODE	Decode the instruction
STORE2	store LSB to memory
STORE2	
STORE2	
STORE2	store MSB to memory

PLDS

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

Bytes are read ‘back-to-back’ without dead cycles in between.

Opcode Format (2 bytes)

42	2B

8 clock cycles

No flags are affected by this instruction.

Machine States:

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

RTC

The RTC instruction (return from context routine) switches contexts from the current back to the invoking context. This is accomplished by reading the back-link field in the current context register. 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	$Immed_8$

4 clock cycles

No flags are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode page 2 prefix
DECODE	Decode / execute –save register set
TASK1	load the register set

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. There is a dead cycle between each byte loaded by the instruction.

Opcode Format (2 bytes)

42	6B

17 clock cycles

No flags are affected by this instruction.

RTI

In native mode the RTI instruction (return from interrupt) switches tasks from the current task back to the interrupted task. This is accomplished by reading the back-link field in the current task's context register. 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

3 clock cycles (native mode operation)

No flags are affected by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode / execute –save register set
TASK1	load the register set

RTT

The RTT instruction (return from task) switches tasks from the current task back to the invoking task. This is accomplished by reading the back-link field in the current task's context register. 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

4 clock cycles

All registers are restored by this instruction.

Machine States:

IFETCH	Fetch the instruction
DECODE	Decode page 2 prefix
DECODE	Decode / execute –save register set
TASK1	load the register set

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.

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

TSK

The TSK instruction is similar to a subroutine call except that it invokes another task rather than a subroutine. When the TSK instruction is executed, it stores the current task number in a back-link field in the new task's context register. This allows a task switch back to the original invoking task when the task is finished running via the RTT (return from task) instruction. 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

4 clock cycles

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)
TASK1	load new task state

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.

Opcode Format (2 bytes)

42	1A

3 clock cycles

N and Z flags are affected by this instruction.

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.
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.

I/O Ports

	In/Out	Width		
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									PLDS	BIT xlabs	AND xlabs	ROL xlabs	
3-	BLT disp	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		STS							PHCS	LDT xlabs,x	EOR xlabs	LSR xlabs	
5-		EOR {d},y	EOR {d}	EOR {d,s},y						EOR xlabs,y		ZS:	JMF seg:offs	EOR xlabs,x	LSR xlabs,x	
6-	RTT	ADC {d,x}										RTF	LDT xlabs	ADC xlabs	ROR xlabs	
7-		ADC {d},y	ADC {d}	ADC {d,s},y						ADC xlabs,y		IOS:	JML [xlabs,x]	ADC xlabs,x	ROR xlabs,x	
8-		STA {d,x}	JCL						DEY4			BYT:	STY xlabs	STA xlabs	STX xlabs	
9-	BGE disp	STA {d},y	STA {d}	STA {d,s},y						STA xlabs,y	TASS	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	TSSA	UHF:	LDY xlabs,x	LDA xlabs,x	LDX xlabs,x	
C-		CMP {d,x}	REP #						INY4		DEX4		CPY xlabs	CMP xlabs	DEC xlabs	
D-		CMP {d},y	CMP {d}	CMP {d,s},y	PEA { }					CMP xlabs,y		CLK	JML [xlabs]	CMP xlabs,x	DEC xlabs,x	
E-	CACHE #	SBC{d,x}	SEP #						INX4		NOP2		CPX xlabs	SBC xlabs	INC xlabs	
F-	PCHIST	SBC {d},y	SBC{d}	SBC {d,s},y	PEA xlabs					SBC xlabs,y			JSL [xlabs,x]	SBC xlabs,x	INC xlabs,x	