Welcome to E-XFL.COM

Embedded - Microcontrollers - Application Specific: Tailored Solutions for Precision and Performance

Embedded - Microcontrollers - Application Specific represents a category of microcontrollers designed with unique features and capabilities tailored to specific application needs. Unlike general-purpose microcontrollers, application-specific microcontrollers are optimized for particular tasks, offering enhanced performance, efficiency, and functionality to meet the demands of specialized applications.

What Are Embedded - Microcontrollers - Application Specific?

Application-specific microcontrollers are engineered to

Details	
Product Status	Obsolete
Applications	Multimedia
Core Processor	TriMedia™
Program Memory Type	-
Controller Series	Nexperia
RAM Size	48K x 8
Interface	I²C, 2-Wire Serial
Number of I/O	169
Voltage - Supply	2.375V ~ 2.625V
Operating Temperature	0°C ~ 85°C
Mounting Type	Surface Mount
Package / Case	292-HBGA
Supplier Device Package	292-BGA
Purchase URL	https://www.e-xfl.com/product-detail/nxp-semiconductors/pnx1301eh-557

16 I2C Interface

17 Synchronous Serial Interface

1.9.7 PNX1300 Series Power Consumption

The power consumption of PNX1300 Series is dependent on the activity of the DSPCPU, the amount of peripherals being used, the frequency at which the system is running as well as the loads on the pins.

The first section presents the power consumption for known applications. The other power related sections present the maximum power consumption. These maximum values are obtained with a 'fake' application that turns on all the peripherals and runs intensive compute on the CPU.

1.9.7.1 Power Consumption for Applications on PNX1300 Series

The and present the power consumption for two typical applications:

- The DVD playback includes video display using the VO peripheral and audio streaming using AO peripheral. The bitstream is brought into the TM-1300 system over the PCI peripheral. The VLD co-processor is used to perform the bitstream parsing. The bitstream is not scrambled therefore the DVDD co-processor is not used and it is turned off.
- The MPEG4 application includes video and audio playback of an enocded CIF stream. The bit stream is brought into the PNX1300 system over the PCI peripheral. The Video and Audio subsystems of the PNX1300 were used to render the video and sound from the decoded stream into the video monitor and speakers.
- The H263 video conferencing application includes the following steps. It captures a CCIR656 video stream at 30 frames/second using the VI peripheral. The incoming video stream is downscaled, on the fly, to SIF resolution by VI. The captured frames are then downscaled to a QSIF resolution using the ICP co-processor. The resulting QSIF image is sent over the PCI bus via the ICP co-processor to a SVGA card (PC monitor display) and encoded by the DSPCPU. The resulting bitstream is then decoded by the DSPCPU and displayed as a SIF image on the same PC monitor (also using the ICP co-processor). All the encoding/decoding part is done in the YUV color space. The display is in the RGB16 color space. Software is not optimized.

Three main technics may be applied to reduce the 'Out of the Box' power consumption.

- Turn off the unused peripherals. Refer to .
- Run the system at the required speed, i.e. some application may not require to run at the full speed grade of the chip.
- Powerdown the system or the DSPCPU each time the DSPCPU reached the Idle task.

A more detailed description can be found in the application note 'TM-1300 Power Saving Features' available at the following website:

http://www.semiconductors.philips.com/trimedia/

Table 1-1. Power Consumption of Example Applications for PNX1300/01/02 (Vdd = 2.5V)

APPLICATIONS	AFTER POWER OPTIMIZATIONS	WITHOUT POWER OPTIMIZATIONS	Optimizations		
			Unused Peripherals Turned Off	System Speed Adjustment	Idle task power management
DVD Playback	2.2 W	3.0 W @ 180 MHz	2.6 W @ 180 MHz	2.6 W @ 180 MHz	2.2 W @ 180 MHz
H.263 Vconf	1.7 W	2.9 W @ 166 MHz	2.7 W @ 166 MHz	1.9 W @ 111 MHz	1.7 W @ 111 MHz

Table 1-2. Power Consumption of Example Applications for PNX1311(Vdd = 2.2V)

APPLICATIONS	AFTER POWER OPTIMIZATIONS	WITHOUT POWER OPTIMIZATIONS	Optimizations		
			Unused Peripherals Turned Off	System Speed Adjustment	Idle task power management
MPEG4 (CIF) A/V Playback	1.2 W	2.5 W @ 166 MHz	2.1 W @ 166 MHz	1.3 W @ 70 MHz	1.2 W @ 70 MHz
H.263 Vconf	1.5 W	2.4 W @ 166 MHz	2.2 W @ 166 MHz	1.7 W @ 111 MHz	1.5 W @ 111 MHz

As previously mentioned the and show that the final power consumption for a realistic application may be lower than the values reported in the next section.

Based on these results and the following section, the power consumption of PNX1300 Series, using an artificial scenario depicting an extremely demanding application, for commonly used speeds, is as follows:

- PNX1300/01/02 is < 3.4 W @ 166:133 MHz
- PNX1311 is < 2.9 W @ 166:133 MHz
- PNX1302 is < 4.0 W @ 200:133 MHz

8.7 AUDIO IN OPERATION

, and describe the function of the control and status fields of the AI unit. To ensure compatibility with future devices, undefined bits in MMIO registers should be ignored when read, and written as '0's.

Table 8-8. AI MMIO control fields

Field Name	Description
RESET	The AI logic is reset by writing a 0x80000000 to AI_CTL. This bit always reads as a '0'. See for details on software reset.
DIAGMODE	0 ⇒ normal operation (RESET default) 1 ⇒ diagnostic mode (see)
SLEEPLESS	0 ⇒ participate in global power down (RESET default) 1 ⇒ refrain from participating in power down
CAP_ENABLE	Capture Enable flag. If 1, AI unit captures samples and acts as DMA master to write samples to local SDRAM. If '0' (RESET default), AI unit is inactive.
BUF1_INTEN	Buffer 1 full Interrupt Enable. Default 0. 0 ⇒ no interrupt 1 ⇒ interrupt (SOURCE 11) if buffer 1 full
BUF2_INTEN	Buffer 2 full interrupt enable. Default 0 0 ⇒ no interrupt 1 ⇒ interrupt (SOURCE 11) if buffer 2 full
HBE_INTEN	HBE Interrupt Enable. Default 0. 0 ⇒ no interrupt 1 ⇒ interrupt (SOURCE 11) if a highway bandwidth error occurs.
OVR_INTEN	Overrun Interrupt Enable. Default 0 0 ⇒ no interrupt 1 ⇒ interrupt (SOURCE 11) if an overrun error occurs
ACK1	Write a '1' to clear the BUF1_FULL flag and remove any pending BUF1_FULL interrupt request. This bit always reads as 0.
ACK2	Write a '1' to clear the BUF2_FULL flag and remove any pending BUF2_FULL interrupt request. This bit always reads as 0.
ACK_HBE	Write a '1' to clear the HBE flag and remove any pending HBE interrupt request. This bit always reads as 0.
ACK_OVR	Write a '1' to clear the OVERRUN flag and remove any pending OVERRUN interrupt request. This bit always reads as 0.

Table 8-9. AI MMIO status fields (read only)

Field Name	Description
BUF1_ACTIVE	• If '1', buffer 1 will be used for the next incoming sample. If '0', buffer 2 will receive the next sample. • 1 after RESET.

Table 8-9. AI MMIO status fields (read only)

Field Name	Description
BUF1_FULL	• If '1', buffer 1 is full. If BUF1_INTEN is also '1', an interrupt request (source 11) is pending. BUF1_FULL is cleared by writing a '1' to ACK1, at which point the AI hardware will assume that BASE1 and SIZE describe a new empty buffer. • 0 after RESET.
BUF2_FULL	• If '1', buffer 2 is full. If BUF2_INTEN is also '1', an interrupt request (source 11) is pending. BUF2_FULL is cleared by writing a '1' to ACK2, at which point the AI hardware will assume that BASE2 and SIZE describe a new empty buffer. • 0 after RESET.
HBE	• Highway Bandwidth Error. Condition raised when the 64-byte internal AI buffer is not yet written to SDRAM when a new input sample arrives. Indicates insufficient allocation of PNX1300 highway bandwidth for the audio sampling rate/mode. Refer to • 0 after RESET.
OVERRUN	• OVERRUN error occurred, i.e. the CPU did not provide an empty buffer in time, and 1 or more samples were lost. If OVR_INTEN is also 1, an interrupt request (source 11) is pending. The OVERRUN flag can ONLY be cleared by writing a '1' to ACK_OVR. • 0 after RESET.

The AI unit is reset by a PNX1300 hardware reset, or by writing 0x80000000 to the AI_CTL register. Upon RESET, capture is disabled (CAP_ENABLE = 0), and buffer1 is the active buffer (BUF1_ACTIVE=1). A minimum of 5 valid AI_SCK clock cycles is required to allow internal AI circuitry to stabilize before enabling capture. This can be accomplished by programming AI_FREQ and AI_SERIAL and then delaying for the appropriate time interval.

Programing of the AI_SERIAL MMIO register needs to follow the following sequence order:

- set AI_FREQ to ensure that a valid clock is generated (Only when AI is the master of the audio clock system)
- MMIO(AI_CTL) = 1 << 31; /* Software Reset */
- MMIO(AI_SERIAL) = 1 << 31; /* sets serial-master mode, starts AI_SCK */
- MMIO(AI_SERIAL) = (1 << 31) | (SCKDIV value); /* then set DIVIDER values */

The DSPCPU initiates capture by providing two equal size empty buffers and putting their base address and size in the $BASE_n$ and SIZE registers. Once two valid (local memory) buffers are assigned, capture can be enabled by writing a '1' to CAP_ENABLE. The AI unit hardware now proceeds to fill buffer 1 with input samples. Once buffer 1 fills up, BUF1_FULL is asserted, and capture continues without interruption in buffer 2. If BUF1_INTEN is enabled, a SOURCE 11 interrupt request is generated.

11.2 PCI INTERFACE AS AN INITIATOR

The following classes of operations invoked by PNX1300 cause the PCI interface to act as a PCI initiator:

- Transparent, single-word (or smaller) transactions caused by DSPCPU loads and stores to the PCI address aperture
- Explicitly programmed single-word I/O or configuration read or write transactions
- Explicitly programmed multi-word DMA transactions.
- ICP DMA

11.2.1 DSPCPU Single-Word Loads/Stores

From the point of view of programs executed by PNX1300's DSPCPU, there are three apertures into PNX1300's 4-GB memory address space:

- SDRAM space (0.5 to 64 MB; programmable)
- MMIO space (2 MB)
- PCI space

MMIO registers control the positions of the address-space apertures (see). The SDRAM aperture begins at the address specified in the MMIO register DRAM_BASE and extends upward to the address in the DRAM_LIMIT register. The 2-MB MMIO aperture begins at the address in MMIO_BASE (defaults to 0xEFE00000 after power-up). All addresses that fall outside these two apertures are assumed to be part of the PCI address aperture. References by DSPCPU loads and stores to the PCI aperture are reflected to external PCI devices by the coordinated action of the data cache and PCI interface.

When a DSPCPU load or store targets the PCI aperture (i.e., neither of the other two apertures), the DSPCPU's data cache automatically carries out a special sequence of events. The data cache writes to the PCI_ADR and (if the DSPCPU operation was a store) PCI_DATA registers in the PCI interface and asserts (load) or de-asserts (store) the internal signal pci_read_operation (a direct connection from the data cache to the PCI interface).

While the PCI interface executes the PCI bus transaction, the DSPCPU is held in the stall state by the data cache. When the PCI interface has completed the transaction, it asserts the internal signal pci_ready (a direct connection from the PCI interface to the data cache).

When pci_ready is asserted, the data cache finishes the original DSPCPU operation by reading data from the PCI_DATA register (if the DSPCPU operation was a load) and releasing the DSPCPU from the stall state.

Explicit Writes to PCI_ADR, PCI_DATA

The PCI_ADR and PCI_DATA registers are intended to be used only by the data cache. Explicit writes are not allowed and may cause undetermined results and/or data corruption.

11.2.2 I/O Operations

Explicit programming by DSPCPU software is the only way to perform transactions to PCI I/O space. DSPCPU software writes three MMIO registers in the following sequence:

1. The IO_ADR register.
2. The IO_DATA register (if PCI operation is a write).
3. The IO_CTL register (controls direction of data movement and which bytes participate).

The PCI interface starts the PCI-bus I/O transaction when software writes to IO_CTL. The interface can raise a DSPCPU interrupt at the completion of the I/O transaction (see BIU_CTL register definition in) or the DSPCPU can poll the appropriate status bit (see BIU_STATUS register definition in). Note that PCI I/O transactions should NOT be initiated if a PCI configuration transaction described below is pending. This is a strict implementation limitation.

The fully detailed description of the steps needed can be found in

11.2.3 Configuration Operations

As with I/O operations, explicit programming by DSPCPU software is the only way to perform transactions to PCI configuration space. DSPCPU software writes three MMIO registers in the following sequence:

1. The CONFIG_ADR register.
2. The CONFIG_DATA register (if PCI operation is a write).
3. The CONFIG_CTL register (controls direction of data movement and which bytes participate).

The PCI interface starts the PCI-bus configuration transaction when software writes to CONFIG_CTL. As with the I/O operations, the biu_status and BIU_CTL registers monitor the status of the operation and control interrupt signaling. Note that PCI configuration space transactions should NOT be initiated if a PCI I/O transaction described above is pending. This is a strict implementation limitation.

The fully detailed description of the steps needed can be found in

11.2.4 DMA Operations

The PCI interface can operate as an autonomous DMA engine, executing block-transfer operations at maximum PCI bandwidth. As with I/O and configuration operations, DSPCPU software explicitly programs DMA operations.

General-purpose DMA

For DMA between SDRAM and PCI, DSPCPU software writes three MMIO registers in the following sequence:

1. The SRC_ADR and DEST_ADR registers.
2. The DMA_CTL register (controls direction of data movement and amount of data transferred).

The ICP block can be separately powered down by setting a bit in the BLOCK_POWER_DOWN register. Refer to

It is recommended that ICP is in an idle state before block level power down is activated.

14.6.3 ICP Operation

The DSPCPU commands the ICP to perform an operation by loading the DP with a pointer to a parameter block, loading the MPC with a microprogram start address and setting Busy in the SR. For example to cause the ICP to scale and filter an image, set up a block of SDRAM with the image and filter parameters, load the MPC with the starting address of the appropriate microprogram entry point in SDRAM, load the DP with the address of the parameter block, and set Busy in the SR by writing a '1' to it. When the filter operation is complete, the ICP will set Done and issue an interrupt. The DSPCPU clears the interrupt by writing a '1' to ACK_DONE. Note: The interrupt should be set up as a 'level triggered.'

When the DSPCPU sets Busy, the MCU begins reading the microprogram from SDRAM. The microinstructions are read in from SDRAM as required by the ICP, and internal pre-fetching is used to eliminate delays. Setting Busy enables the MCU clock, the first block of microinstructions is automatically read in, and the MCU begins instruction execution at the current address in the MPC. Clearing Busy stops the MCU clock. Busy can be cleared by hardware reset, by the MCU, or by the DSPCPU. Hardware reset clears the Status register, including Busy and Done, and internal registers, such as the TCR. When the MCU completes a microprogram operation, the microprogram typically clears Busy and sets Done, causing an interrupt if IE is enabled.

The DSPCPU performs a software reset by clearing (writing a '0' to) Busy and by writing a '1' to Reset. The DSPCPU can also set Done to force a hardware interrupt, if desired.

14.6.4 ICP Microprogram Set

The ICP comes with a factory-generated microprogram set which implements the functions of the ICP. The microprogram set includes the following functions:

1. Loading the filter coefficient RAMs.
2. Horizontal scaling and filtering from SDRAM to SDRAM of an input image to an output image. The input and output images can be of any size and position that fits in SDRAM. The scaling factors are, in general, limited only by input and output image sizes.
3. Vertical scaling and filtering from SDRAM to SDRAM of an input image to an output image. The input and output images can be of any size and position that fits in SDRAM. The scaling factors are, in general, limited only by input and output image sizes.
4. Horizontal scaling, filtering and YUV to RGB conversion of an input image from SDRAM to an output image to PCI or SDRAM, with an alpha-blended and chroma-keyed RGB overlay and a bit mask. The input and output images can be of any size and position that fit in SDRAM and can be output to the PCI bus or SDRAM. In general, scaling factors are limited only by input and output image sizes.

The microprogram is supplied with the ICP as part of the device driver. The entry point in the microprogram defines which ICP operation is to be done. The entry points are given below in terms of word offsets from the beginning of the microprogram:

Offset	Function
0	Load coefficients
1	Horizontal scaling and filtering
2	Vertical scaling and filtering
3	Horizontal scaling, filtering, YUV to RGB conversion, bit masking (PCI) and overlay (PCI) with alpha blending and chroma keying

14.6.5 ICP Processing Time

The processing time for typical operations on typical picture sizes has been measured.

Measurements were performed with the following configuration:

- CPU clock and SDRAM clock set to 100 MHz
- PCI clock set to 33MHz
- All measurement with PCI as pixel destination were done with an Imagine 128 Series II graphics card, which never caused a slowdown of the ICP operation.
- TRITON2 mother-board with SB82437UX and SB82371SB based Intel® Pentium™ chipset.
- PNX1300 arbiter set to default settings
- PNX1300 latency timer set to maximum value = 0xf8.
- Overlay sizes were the same as picture sizes.

Results are tabulated below for three different cases of available memory bandwidth:

1. No other load to SDRAM, i.e. full SDRAM bandwidth available for ICP. See .

2. SDRAM memory loaded to 95% of its bandwidth by DCACHE traffic from DSPCPU. Priority delay = 1, i.e. ICP did wait one block time before competing for memory. See .

3. SDRAM memory loaded to 95% of its bandwidth by DCACHE traffic from DSPCPU. Priority delay = 16, i.e. ICP did wait 16 block times before competing for memory. See .

Note: A load of 95% of the memory bandwidth is very rarely found in a real system. So the results in these tables may be useful to estimate upper bounds for the computation time in a loaded system.

The priority delays were set to the minimum and maximum possible values, so the computation time for other priority delay values should be somewhere in between.

17.10.1 SSI Control Register (SSI_CTL)

SSI_CTL is a 32-bit read/write control register used to direct the operation of the SSI. The value of this register after a hardware reset is 0x00F00000.

Table 17-5. SSI control register (SSI_CTL) fields.

Field	Description
TXR	Transmitter Software Reset (Bit 31). Setting TXR performs the same functions as a hardware reset. Resets all transmitter functions. A transmission in progress is interrupted and the data remaining in the TxSR is lost. The TxFIFO pointers are reset and the data contained will not be transmitted, but the data in the SSI_TxDR and/or TxFIFO are not explicitly deleted. The transmitter status and interrupts are all cleared. This is an action bit. This bit always reads '0'. Writing a '1' in combination with writing a '1' in the RXR field will initiate a reset for the SSI module. Note: this bit is always set together with RXR because a separate transmitter or receiver reset is not implemented.
RXR	Receiver Software Reset (Bit 30). Setting RXR performs the same functions as a hardware reset. Resets all receiver functions. A reception in progress is interrupted and the data collected in the RxSR is lost. The RxFIFO pointers are reset, and the SSI will not generate an interrupt to DSPCPU to retrieve data in the SSI_RxDR and/or RxFIFO. The data in the SSI_RxDR and/or RxFIFO is not explicitly deleted. The receiver status and interrupts are all cleared.This is an action bit.This bit always reads '0'. Writing a '1' in combination with writing a '1' in the TXR field will initiate a reset for the SSI module. Note: this bit is always set together with TXR, because a separate transmitter or receiver reset is not implemented.
TXE	Transmitter Enable (Bit 29). TXE enables the operation of the transmit shift register state machine. When TXE is set and a frame sync is detected, the transmit state machine of the SSI is begins transmission of the frame. When TXE is cleared, the transmitter will be disabled after completing transmission of data currently in the TxSR. The serial output (SSI_TxDATA) is three-stated, and any data present in SSI_TxDR and/or TxFIFO will not be transmitted (i.e., data can be written to SSI_TxDR with TXE cleared; TDE can be cleared, but data will not be transferred to the TxSR). Status fields updated by the Transmit state machine are not updated or reset when an active transmitter is disabled.
RXE	Receive Enable (Bit 28). When RXE is set, the receive state machine of the SSI is enabled. When this bit is cleared, the receiver will be disabled by inhibiting data transfer into SSI_RxDR and/or RxFIFO. If data is being received while this bit is cleared, the remainder of that 16-bit word will be shifted in and transferred to the SSI RxFIFO and/or SSI_RxDR. Status fields updated by the Receive state machine are not updated or reset when an active receiver is disabled.
TCP	Transmit Clock Polarity (Bit 27). The TCP bit value should only be changed when the transmitter is disabled. TCP controls on which edge of TxCLK data is output. TCP=0 causes data to be output at rising edge of TxCLK, TCP=1 causes data to be output at falling edge of TxCLK.
RCP	Receive Clock Polarity (Bit 26). RCP controls which edge of RxCLK samples data. The data is sampled at rising edge when RCP = '1' or falling edge when RCP = '0'.
TSD	Transmit Shift Direction (Bit 25). TSD controls the shift direction of transmit shift register (TxSR). Transmit data is transmitted MSB first when TSD = '0' or LSB first otherwise. The operation of this bit is explained in more detail in section .
RSD	Receive Shift Direction (Bit 24). The RSD bit value should only be changed when the receiver is disabled. RSD controls the shift direction of receive shift register (RxSR). Receive data is received MSB first when RSD = '0', LSB first otherwise. The operation of this bit is explained in more detail in section .
IO1	Mode Select SSI_IO1 pin (Bit 23-22). The IO1 field value should only be changed when the transmitter and receiver are disabled. The IO1[1:0] bits are used to select the function of SSI_IO1 pin. The function may be selected as listed in table .
IO2	Mode Select SSI_IO2 pin (Bit 21-20). The IO2 field value should only be changed when the transmitter and receiver are disabled. The IO2[1:0] bits are used to select the function of SSI_IO2 pin. The function may be selected according to
WIO1	Write IO1 (Bit 19). Value written here appears on the SSI_IO1 pin when the pin is configured to be a general purpose output.
WIO2	Write IO2 (Bit 18). Value written here appears on the SSI_IO2 pin when this pin is configured to be a general purpose output.
TIE	Transmit Interrupt Enable (Bit 17). Enables interrupt by the TDE flag in the SSI status register (transmit needs refill) Also enables interrupt of the TUE (transmitter underrun error) and TXFES (transmit framing error)
RIE	Receive Interrupt Enable (Bit 16). When RIE is set, the DSPCPU will be interrupted when RDF in the SSI status register is set (receive complete). It will also be interrupted on ROE (receiver overrun error) and on RXFES (receive framing error).
FSS	Frame Size Select (Bits 15-12). The FSS[3:0] bits control the divide ratio for the programmable frame rate divider used to generate the frame sync pulses. The valid setup value ranges from 1 to 16 slot(s). The value '16' is accomplished by storing a 0 in this field.

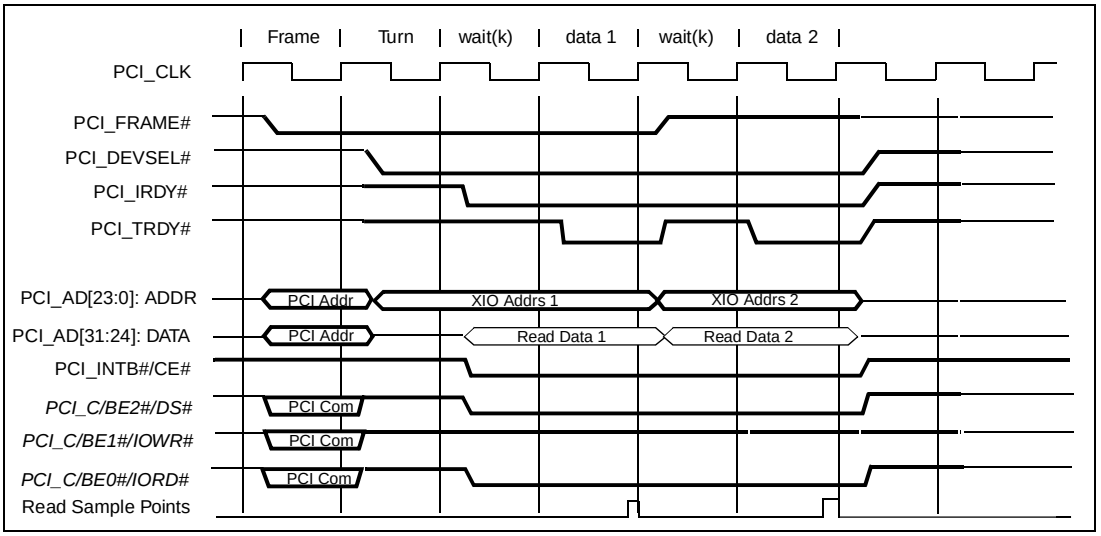


Figure 22-15. PCI-XIO Bus timing: DMA burst read, 2 bytes, 1 or more wait states

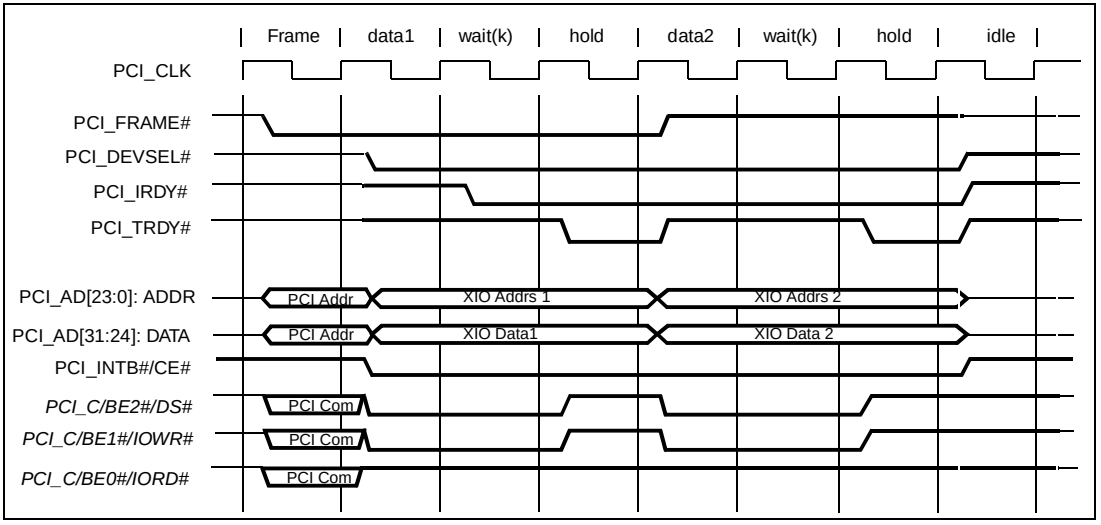


Figure 22-16. PCI-XIO Bus timing: DMA burst write, 2 bytes, 1 or more wait states

asri

Arithmetic shift right by immediate amount

SYNTAX

```
[ IF rguard ] asri(n) rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
    r*dest*<31:31–*n*> ← r*src1*<31>
    r*dest*<30–*n*:0> ← r*src1*<31:*n*>
}

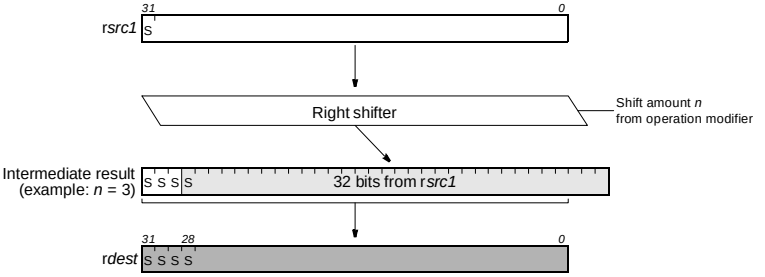
ATTRIBUTES

Function unit	shifter
Operation code	10
Number of operands	1
Modifier	7 bits
Modifier range	0..31
Latency	1
Issue slots	1, 2

SEE ALSO

DESCRIPTION

As shown below, the `asri` operation takes a single argument in r*src1* and an immediate modifier *n* and produces a result in r*dest* that is equal to r*src1* arithmetically shifted right by *n* bits. The value of *n* must be between 0 and 31, inclusive. The MSB (sign bit) of r*src1* is replicated as needed to fill vacated bits from the left.



The `asri` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x7008000f	`asri(1) r30 → r50`	r50 ← 0x38040007
r30 = 0x7008000f	`asri(2) r30 → r60`	r60 ← 0x1c020003
r10 = 0, r30 = 0x7008000f	`IF r10 asri(4) r30 → r70`	no change, since guard is false
r20 = 1, r30 = 0x7008000f	`IF r20 asri(4) r30 → r80`	r80 ← 0x07008000
r40 = 0x80030007	`asri(4) r40 → r90`	r90 ← 0xf8003000
r30 = 0x7008000f	`asri(31) r30 → r100`	r100 ← 0x00000000
r40 = 0x80030007	`asri(31) r40 → r110`	r110 ← 0xffffffff

Bitwise logical exclusive-OR

bitxor

SYNTAX

```
[ IF rguard ] bitxor rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← r*src1* ⊕ r*src2*

ATTRIBUTES

Function unit	alu
Operation code	48
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

DESCRIPTION

The `bitxor` operation computes the bitwise, logical exclusive-OR of the first and second arguments, r*src1* and r*src2*. The result is stored in the destination register, r*dest*.

The `bitxor` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xf310ffff, r40 = 0xffff0000	`bitxor r30 r40 → r90`	r90 ← 0x0ceffff
r10 = 0, r50 = 0x88888888	`IF r10 bitxor r30 r50 → r80`	no change, since guard is false
r20 = 1, r30 = 0xf310ffff, r50 = 0x88888888	`IF r20 bitxor r30 r50 → r100`	r100 ← 0x7b987777
r60 = 0x11119999, r50 = 0x88888888	`bitxor r60 r50 → r110`	r110 ← 0x99991111
r70 = 0x55555555, r30 = 0xf310ffff	`bitxor r70 r30 → r120`	r120 ← 0xa645aaaa

Floating-point absolute value

fabsval

SYNTAX

```
[ IF rguard ] fabsval rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if (float)rsrc1 < 0 then
      rdest ← –(float)rsrc1
   else
      rdest ← (float)rsrc1
}
```

ATTRIBUTES

Function unit	falu
Operation code	115
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

DESCRIPTION

The `fabsval` operation computes the absolute value of the argument r*src1* and stores the result into r*dest*. All values are in IEEE single-precision floating-point format. If an argument is denormalized, zero is substituted for the argument before computing the absolute value, and the IFZ flag in the PCSW is set. If `fabsval` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit`writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r *dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `fabsvalflags` operation computes the exception flags that would result from an individual `fabsval`.

The `fabsval` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r *guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`fabsval r30 → r90`	r90 ← 0x40400000 (3.0)
r35 = 0xbf800000 (-1.0)	`fabsval r35 → r95`	r95 ← 0x3f800000 (1.0)
r40 = 0x00400000 (5.877471754e-39)	`fabsval r40 → r100`	r100 ← 0x0 (+0.0), IFZ set
r45 = 0xffffffff (QNaN)	`fabsval r45 → r105`	r105 ← 0xffffffff (QNaN)
r50 = 0xffbfffff (SNaN)	`fabsval r50 → r110`	r110 ← 0xffffffff (QNaN), INV set
r10 = 0, r55 = 0xff7fffff (–3.402823466e+38)	`IF r10 fabsval r55 → r115`	no change, since guard is false
r20 = 1, r55 = 0xff7fffff (–3.402823466e+38)	`IF r20 fabsval r55 → r120`	r120 ← 0x7f7fffff (3.402823466e+38)

fsignflags

IEEE status flags from floating-point sign

SYNTAX

```
[ IF rguard ] fsignflags rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
    r*dest* ← ieee_flags(sign((float)r*src1*))

ATTRIBUTES

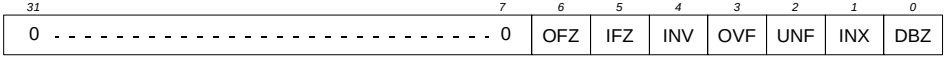
Function unit	fcomp
Operation code	153
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

DESCRIPTION

The `fsignflags` operation computes the IEEE exceptions that would result from computing the sign of r*src1* and stores a bit vector representing the exception flags into r*dest*. The argument value is in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. If the argument is denormalized, zero is substituted before computing the sign, and the IFZ bit in the result is set.

The `fsignflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`fsignflags r30 → r100`	r100 ← 0
r40 = 0xbf800000 (-1.0)	`fsignflags r40 → r105`	r105 ← 0
r50 = 0x80800000 (-1.175494351e-38)	`fsignflags r50 → r110`	r110 ← 0
r60 = 0x80400000 (-5.877471754e-39)	`fsignflags r60 → r115`	r115 ← 0x20 (IFZ)
r10 = 0, r70 = 0xffffffff (QNaN)	`IF r10 fsignflags r70 → r116`	no change, since guard is false
r20 = 1, r70 = 0xffffffff (QNaN)	`IF r20 fsignflags r70 → r117`	r117 ← 0x10 (INV)
r80 = 0xff800000 (-INF)	`fsignflags r80 → r120`	r120 ← 0

ileq

Signed compare less or equal

pseudo-op for igeq

SYNTAX

```
[ IF rguard ] ileq rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 <= rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	14
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

DESCRIPTION

The `ileq` operation is a pseudo operation transformed by the scheduler into an `igeq` with the arguments exchanged (`ileq`'s r*src1* is `igeq`'s r*src2* and vice versa). (Note: pseudo operations cannot be used in assembly source files.)

The `ileq` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than or equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The `ileq` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`ileq r30 r40 → r80`	r80 ← 1
r10 = 0, r60 = 0x100, r30 = 3	`IF r10 ileq r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, 0x100	`IF r20 ileq r50 r60 → r90`	r90 ← 0
r70 = 0x80000000, r40 = 4	`ileq r70 r40 → r100`	r100 ← 1
r70 = 0x80000000	`ileq r70 r70 → r110`	r110 ← 1

prefetch with index

prefr

SYNTAX

```
[ IF rguard ] prefr rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
    cache_block_mask = ~(cache_block_size - 1)
    data_cache <- mem[(rsrc1 + rscr2) & cache_block_mask]
}
```

ATTRIBUTES

Function unit	dmemspec
Operation code	210
Number of operands	2
Modifier	No
Modifier range	-
Latency	-
Issue slots	5

SEE ALSO

DESCRIPTION

The prefr operation loads the one full cache block size of memory value from the address computed by ((rsrc1+rscr2) & cache_block_mask) and stores the data into the data cache. This operation is not guaranteed to be executed. The prefetch unit will not execute this operation when the data to be prefetched is already in the data cache. A prefr operation will not be executed when the cache is already occupied with 2 cache misses, when the operation is issued.

The `prefr` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the execution of the prefetch operation. If the LSB of r*guard* is 1, prefetch operation is executed; otherwise, it is not executed..

EXAMPLES

Initial Values	Operation	Result
r10 = 0xabcd, r12 = 0xd cache_block_size = 0x40	`prefr r10 r12`	Loads a cache line for the address space from 0xabc0 to 0x0xac3f from the main memory. If the data is already in the cache, the operation is not executed.
r10 = 0xabcd, r11 = 0, r12=0xd, cache_block_size = 0x40	`IF r11 prefr r10 r12`	since guard is false, prefr operation is not executed
r10 = 0xabff, r11 = 1, r12 =0x1, cache_block_size = 0x40	`IF r11 prefr r10 r12`	Loads a cache line for the address space from 0xac00 to 0x0xac3f from the main memory. If the data is already in the cache, the operation is not executed.

NOTE: This operation may only be supported in TM-1000, TM-1100, TM-1300 and PNX1300/01/02/11. It is not guaranteed to be available in future generations of Trimedia products.

8-bit store

pseudo-op for h_st8d(0)

st8

SYNTAX

```
[ IF rguard ] st8 rsrc1 rsrc2
```

FUNCTION

if r*guard* **then**
mem[r*src1*] ← r*src2*<7:0>

ATTRIBUTES

Function unit	dmem
Operation code	29
Number of operands	2
Modifier	No
Modifier range	—
Latency	n/a
Issue slots	4, 5

SEE ALSO

DESCRIPTION

The `st8` operation is a pseudo operation transformed by the scheduler into an `h_st8d(0)` with the same arguments. (Note: pseudo operations cannot be used in assembly files.)

The `st8` operation stores the least-significant 8-bit byte of r*src2* into the memory location pointed to by the address in r*src1*. This operation does not depend on the bytesex bit in the PCSW since only a single byte is stored.

The result of an access by `st8` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `st8` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the addressed memory location (and the modification of cache if the location is cacheable). If the LSB of r*guard* is 1, the store takes effect. If the LSB of r*guard* is 0, st8 has no side effects whatever; in particular, the LRU and other status bits in the data cache are not affected.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r80 = 0x44332211	`st8 r10 r80`	[0xd00] ← 0x11
r50 = 0, r20 = 0xd01, r70 = 0xaabbccdd	`IF r50 st8 r20 r70`	no change, since guard is false
r60 = 1, r30 = 0xd02, r70 = 0xaabbccdd	`IF r60 st8 r30 r70`	[0xd02] ← 0xdd

Unsigned compare equal with immediate

ueqli

SYNTAX

```
[ IF rguard ] ueqli(n) rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
  if r*src1* = *n* **then**
    r*dest* ← 1
  else
    r*dest* ← 0
}

ATTRIBUTES

Function unit	alu
Operation code	38
Number of operands	1
Modifier	7 bits
Modifier range	0..127
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

DESCRIPTION

The `ueqli` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is equal to the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The `ueqli` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ueqli(2) r30 → r80`	r80 ← 0
r30 = 3	`ueqli(3) r30 → r90`	r90 ← 1
r30 = 3	`ueqli(4) r30 → r100`	r100 ← 0
r10 = 0, r40 = 0x100	`IF r10 ueqli(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 ueqli(63) r40 → r100`	r100 ← 0
r60 = 0x07f	`ueqli(127) r60 → r120`	r120 ← 1

Unsigned compare greater or equal with immediate

ugeqi

SYNTAX

```
[ IF rguard ] ugeqi(n) rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
  if (unsigned)r*src1* >= (unsigned)*n* **then**
    r*dest* ← 1
  else
    r*dest* ← 0
}

ATTRIBUTES

Function unit	alu
Operation code	36
Number of operands	1
Modifier	7 bits
Modifier range	0..127
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

DESCRIPTION

The `ugeqi` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than or equal to the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The `ugeqi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ugeqi(2) r30 → r80`	r80 ← 1
r30 = 3	`ugeqi(3) r30 → r90`	r90 ← 1
r30 = 3	`ugeqi(4) r30 → r100`	r100 ← 0
r10 = 0, r40 = 0x100	`IF r10 ugeqi(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 ugeqi(63) r40 → r100`	r100 ← 1
r60 = 0x80000000	`ugeqi(127) r60 → r120`	r120 ← 1

Sum of absolute values of unsigned 8-bit differences

ume8uu

SYNTAX

```
[ IF rguard ] ume8uu rsrc1 rsrc2 → rdest
```

ATTRIBUTES

Function unit	dspalu
Operation code	26
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

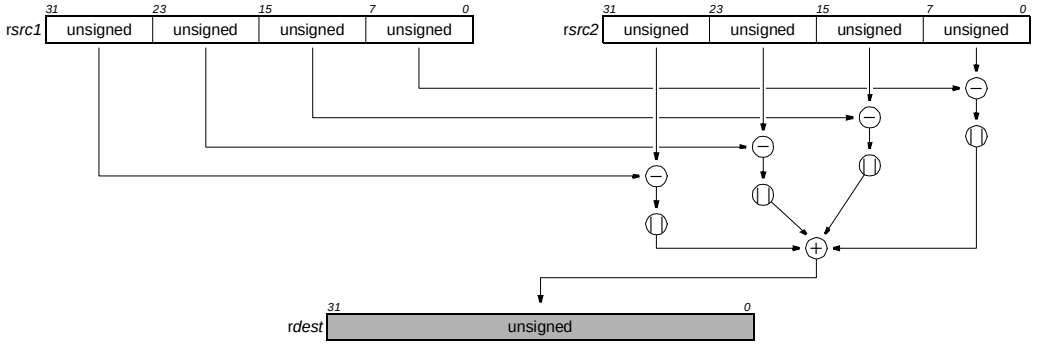
FUNCTION

if r*guard* **then**
  r*dest* ← abs_val(zero_ext8to32(r*src1*<31:24>) – zero_ext8to32(r*src2*<31:24>)) +
    abs_val(zero_ext8to32(r*src1*<23:16>) – zero_ext8to32(r*src2*<23:16>)) +
    abs_val(zero_ext8to32(r*src1*<15:8>) – zero_ext8to32(r*src2*<15:8>)) +
    abs_val(zero_ext8to32(r*src1*<7:0>) – zero_ext8to32(r*src2*<7:0>))

SEE ALSO

DESCRIPTION

As shown below, the ume8uu operation computes four separate differences of the four pairs of corresponding unsigned 8-bit bytes of r*src1* and r*src2*. The absolute values of the four differences are summed and the result is written to r*dest*. All computations are performed without loss of precision.



The ume8uu operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r80 = 0x0a14f6f6, r30 = 0x1414ecf6	`ume8uu r80 r30 → r100`	r100 ← 0x14
r10 = 0, r80 = 0x0a14f6f6, r30 = 0x1414ecf6	`IF r10 ume8uu r80 r30 → r70`	no change, since guard is false
r20 = 1, r90 = 0x64649c9c, r40 = 0x649c649c	`IF r20 ume8uu r90 r40 → r110`	r110 ← 0x70
r40 = 0x649c649c, r90 = 0x64649c9c	`ume8uu r40 r90 → r120`	r120 ← 0x70
r50 = 0x80808080, r60 = 0x7f7f7f7f	`ume8uu r50 r60 → r125`	r125 ← 0x4

MMIO Register Name	Offset (in hex)	Accessibility		Description
		DSPCPU	External PCI Initiators	
IC_LOCK_SIZE	10 0218	R/W	R/W	Size of address range that will be locked into the instruction cache
PLL_RATIOS	10 0300	R/—	R/—	Sets ratios of external and internal clock frequencies
BLOCK_POWER_DOWN	10 3428	R/W	R/W	Powers up and down individual blocks
Video In				
VI_STATUS	10 1400	R/—	R/—	Status of video-in unit
VI_CTL	10 1404	R/W	R/W	Sets operation and interrupt modes for video in
VI_CLOCK	10 1408	R/W	R/W	Sets clock source (internal/external), frequency
VI_CAP_START	10 140c	R/W	R/W	Sets capture start x and y offsets
VI_CAP_SIZE	10 1410	R/W	R/W	Sets capture size width and height
VI_BASE1 VI_Y_BASE_ADR	10 1414	R/W	R/W	Capture modes: sets base address ofY-value array Message/raw modes: sets base address of buffer 1
VI_BASE2 VI_U_BASE_ADR	10 1418	R/W	R/W	Capture modes: sets base address of U-value array Message/raw modes: sets base address of buffer 2
VI_SIZE VI_V_BASE_ADR	10 141c	R/W	R/W	Capture modes: sets base address ofV-value array Message/raw modes: sets size of buffers
VI_UV_DELTA	10 1420	R/W	R/W	Capture modes: address delta for adjacent U, V lines
VI_Y_DELTA	10 1424	R/W	R/W	Capture modes: address delta for adjacent Y lines
Video Out				
VO_STATUS	10 1800	R/—	R/—	Status of video-out unit
VO_CTL	10 1804	R/W	R/W	Sets operation and interrupt modes for video out
VO_CLOCK	10 1808	R/W	R/W	Sets video-out clock frequency
VO_FRAME	10 180c	R/W	R/W	Sets frame parameters (preset, start, length)
VO_FIELD	10 1810	R/W	R/W	Sets field parameters (overlap, field-1 line, field-2 line)
VO_LINE	10 1814	R/W	R/W	Sets field parameters (starting pixel, frame width)
VO_IMAGE	10 1818	R/W	R/W	Sets image parameters (height, width)
VO_YTHR	10 181c	R/W	R/W	Sets threshold for YTR interrupt, image v/h offsets
VO_OLSTART	10 1820	R/W	R/W	Sets overlay image parameters (start line/pixel, alpha)
VO_OLHW	10 1824	R/W	R/W	Sets overlay image parameters (height, width)
VO_YADD	10 1828	R/W	R/W	Sets Y-component/buffer-1 starting address
VO_UADD	10 182c	R/W	R/W	Sets U-component/buffer-2 starting address
VO_VADD	10 1830	R/W	R/W	Sets V-component address/buffer-1 length
VO_OLADD	10 1834	R/W	R/W	Sets overlay image address/buffer-2 length
VO_VUF	10 1838	R/W	R/W	Sets start-of-line-to-start-of-line address offsets (U, V)
VO_YOLF	10 183c	R/W	R/W	Sets start-of-line-to-start-of-line addr. offsets (Y, overlay)
EVO_CTL	10 1840	R/W	R/W	Sets operations for enhance video out
EVO_MASK	10 1844	R/W	R/W	Sets YUV mask values foe the chroma-key process
EVO_CLIP	10 1848	R/W	R/W	Sets output clip values
EVO_KEY	10 184c	R/W	R/W	Sets YUV chroma-key values
EVO_SLVDLY	10 1850	R/W	R/W	Sets delay cycles for genlock mode
Audio In				
AI_STATUS	10 1c00	R/—	R/—	Status of audio-in unit
AI_CTL	10 1c04	R/W	R/W	Sets operation and interrupt modes for audio in
AI_SERIAL	10 1c08	R/W	R/W	Sets clock ratios and internal/external clock generation
AI_FRAMING	10 1c0c	R/W	R/W	Sets format of serial data stream