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/pnx1302eh-557

```
void reconstruct (unsigned char *back,
                  unsigned char *forward,
                          char *idct,
                  unsigned char *destination)
{
    int i, temp;

    for (i = 0; i < 64; i += 1)
    {
        temp = ((back[i] + forward[i] + 1) >> 1) + idct[i];

        if (temp > 255)
            temp = 255;
        else if (temp < 0)
            temp = 0;

        destination[i] = temp;
    }
}
```

Figure 4-4. Straightforward code for MPEG frame reconstruction.

A straightforward coding of the reconstruction algorithm might look as shown in . This implementation shares many of the undesirable properties of the first example of byte-matrix transposition. The code accesses memory a byte at a time instead of a word at a time, which wastes 75% of the available bandwidth. Also, in light of the many quad-byte-parallel operations introduced in it seems inefficient to spend three separate additions and one shift to process a single eight-bit pixel. Perhaps even more unfortunate for a VLIW processor like PNX1300 is the branch-intensive code that performs the saturation testing; eliminating these branches could reap a significant performance gain.

Since MPEG decoding is the kind of task for which PNX1300 was created, there are two custom operations—quadavg and dspuquadaddui—that exactly fit this important MPEG kernel (and other kernels). These custom operations process four pairs of 8-bit pixel values in parallel. In addition, dspuquadaddui performs saturation tests in hardware, which eliminates any need to execute explicit tests and branches.

For readers familiar with the details of MPEG algorithms, the use of eight-bit IDCT values later in this example may be confusing. The standard MPEG implementation calls for nine-bit IDCT values, but extensive analysis has shown that values outside the range [–128..127] occur so rarely that they can be considered unimportant. Pursuant to this observation, the IDCT values are clipped into the eight-bit range [–128..127] with saturating arithmetic before the frame reconstruction code runs. The assumption that this saturation occurs permits some of PNX1300's custom operations to have clean, simple definitions.

The first step in seeing how custom operations can be of value in this case, is to unroll the loop by a factor of four. The unrolled code is shown in . This creates code that is parallel with respect to the four pixel computations. As it is easily seen in the code, the four groups of computations (one group per pixel) do not depend on each other.

After some experience is gained with custom operations, it is not necessary to unroll loops to discover situations where custom operations are useful. Often, a good programmer with knowledge of the function of the custom operations can see by simple inspection opportunities to exploit custom operations.

To understand how quadavg and dspuquadaddui can be used in this code, we examine the function of these custom operations.

The quadavg custom operation performs pixel averaging on four pairs of pixels in parallel. Formally, the operation of quadavg is as follows:

```
quadavg rscr1 rsrc2 -> rdest
```

takes arguments in registers rsrc1 and rsrc2, and it computes a result into register rdest. rsrc1 = [abcd], rsrc2 = [wxyz], and rdest = [pqrs] where a, b, c, d, w, x, y, z, p, q, r, and s are all unsigned eight-bit values. Then, quadavg computes the output vector [pqrs] as follows:

```
p = (a + w + 1) >> 1
q = (b + x + 1) >> 1
r = (c + y + 1) >> 1
s = (d + z + 1) >> 1
```

The pixel averaging in is evident in the first statement of each of the four groups of statements. The rest of the code—adding idct[i] value and performing the saturation test—can be performed by the dspuquadaddui operation. Formally, its function is as follows:

```
dspuquadaddui rsrc1 rsrc2 -> rdest
```

takes arguments in registers rsrc1 and rsrc2, and it computes a result into register rdest. rsrc1 = [efgh], rsrc2 = [stuv], and rdest = [ijkl] where e, f, g, h, i, j, k, and l are unsigned 8-bit values; s, t, u, and v are signed 8-bit values. Then, dspuquadaddui computes the output vector [ijkl] as follows:

```
i = uclipi(e + s, 255)
j = uclipi(f + t, 255)
k = uclipi(g + u, 255)
l = uclipi(h + v, 255)
```

The uclipi operation is defined in this case as it is for the separate PNX1300 operation of the same name described in . Its definition is as follows:

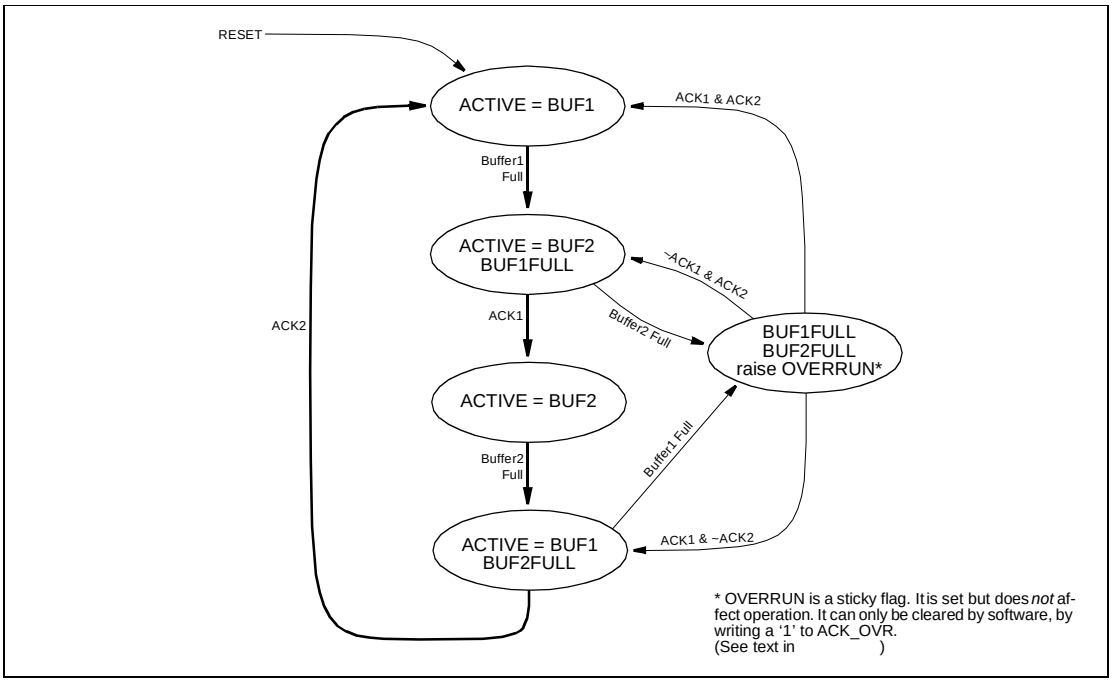


Figure 6-16. VI raw mode major states.

and SIZE (in number of samples), it is safe to enable capture by setting CAPTURE_ENABLE. Note that SIZE is in samples and must be a multiple of 64, hence setting a minimum buffer size of 64 bytes for raw8 mode and 128 bytes for raw10 modes. At this point, buffer1 is the active capture buffer. Data is captured in buffer1 until capture is disabled or until SIZE samples have been captured. After every sample, a running address pointer is incremented by the sample size (one or two bytes). If SIZE samples have been captured, capture continues (without missing a sample) in buffer2. At the same time, BUF1FULL is asserted. This causes an interrupt on the DSPCPU, if enabled by BUF1FULL INTERRUPT ENABLE.

Buffer2 is now the active capture buffer and behaves as described above. In normal operation, the DSPCPU will respond to the BUF1FULL event by assigning a new BASE1 and (optionally) SIZE and performing an ACK1. If the DSPCPU fails to assign a new buffer1 and performs an ACK1 before buffer2 also fills up, the OVERRUN condition is raised and capture stops. Capture continues upon receipt of an ACK1, ACK2, or both, regardless of the OVERRUN state. The buffer in which capture resumes is as indicated in . The OVERRUN condition is 'sticky' and can only be cleared by software, by writing a '1' to the ACK_OVR bit in the VI_CTL register.

If insufficient bandwidth is allocated from the internal data highway, the VI internal buffers may overflow. This leads to assertion of the HIGHWAY BANDWIDTH ERROR condition. One or more data samples are lost. Capture resumes at the correct memory address as soon as the internal buffer is written to memory. The HBE error condition is sticky. It remains asserted until it is cleared by writing a '1' to HIGHWAY BANDWIDTH ERROR ACK. Refer to

Note that VI hardware uses copies of the BASE and SIZE registers once capture has started. Modifications of BASE or SIZE, therefore, have no effect until the start of the next use of the corresponding buffer.

Note also that the VI_BASE1 and VI_BASE2 addresses must be 64-byte aligned (the six LSBs are always '0').

6.6 MESSAGE-PASSING MODE

In this mode, VI receives 8-bit message data over the VI_DATA[7:0] pins. The message data is written in packed form (four 8-bit message bytes per 32-bit word) to SDRAM. Message data capture starts on receipt of a START event on VI_DATA[8]. Message data is received until EndOfMessage (EOM) is received on VI_DATA[9] or the receive buffer is full. Note that the VI_SIZE MMIO register determines the buffer size, and hence maximum message length. It should not be changed without a VI (soft) reset.

illustrates an example of an 8-byte message transfer. The first byte (D0) is sampled on the rising edge of the VI_CLK clock after a valid START was sampled on the preceding rising clock edge. The last byte (D7) is

1. SDRAM buffers must start on a 64-byte boundary.

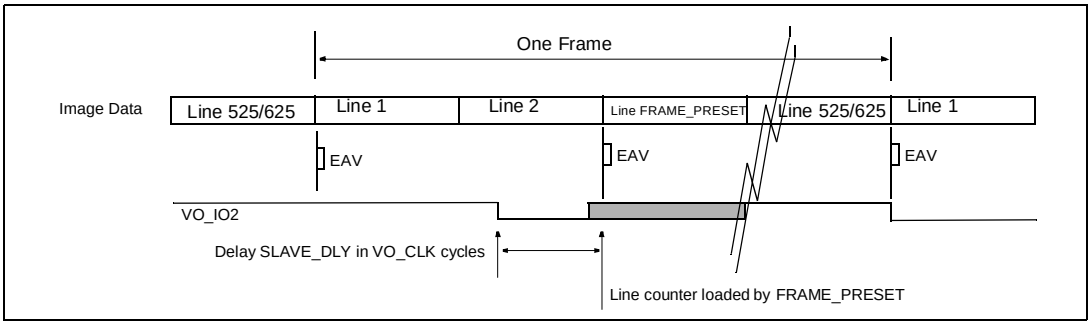


Figure 7-16. Genlock mode.

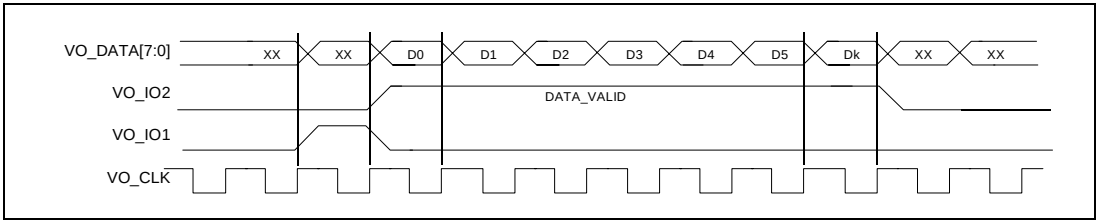


Figure 7-17. Data-streaming valid data signals.

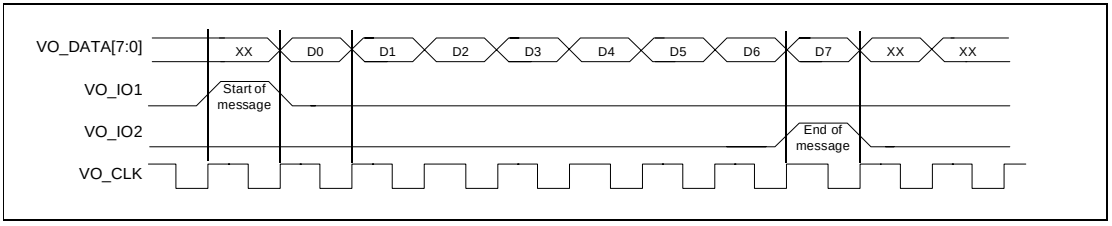


Figure 7-18. Message-passing START and END signals.

the first byte of the first line just after the VO_IO2 active signal.

7.11 DATA TRANSFER TIMING

In data-streaming and message-passing modes, the EVO supplies a stream of 8-bit data. No data selection or data interpretation is done, and data is transferred at the rate of one byte per VO_CLK. Data is clocked out on the positive edge of VO_CLK.

When data-streaming mode is enabled and EVO_ENABLE = 1 and SYNC_STREAMING = 1, the VO_IO2 signal indicates a data-valid condition. This signal is asserted when the EVO starts outputting valid data (that is, data-streaming mode is enabled and video out is running), and is de-asserted when data-streaming mode is disabled. As shown in , the data-valid signal on VO_IO2 is asserted just before the first valid byte is present on VO_DATA[7:0], and is de-asserted just after the last valid byte was sent, or if an HBE error is signaled. All transitions of VO_IO2 occur on the rising edge of VO_CLK. The VO_IO1 signal generates a pulse one VO_CLK cycle before the first valid data is sent. The transitions of VO_IO1 occur on the rising edge of VO_CLK and last for one VO_CLK cycle.

In message-passing mode, the EVO issues signals on VO_IO1 and VO_IO2 to indicate the start and end of messages.

When message passing is started by setting VO_CTL. VO_ENABLE, the EVO sends a Start condition on VO_IO1. When the EVO has transferred the contents of the buffer, it sends an End condition on VO_IO2, sets BFR1_EMPTY, and interrupts the DSPCPU. The EVO stops, and no further operation takes place until the DSPCPU sets VO_ENABLE again to start another message, or until the DSCPU initiates other EVO operation. The timing for these signals is shown in .

7.12 IMAGE DATA MEMORY FORMATS

7.12.1 Video Image Formats

The EVO accepts memory-resident video image data in three formats: YUV 4:2:2 co-sited, YUV 4:2:2 interspersed, and YUV 4:2:0. These formats are shown in through .

Table 7-7. VO_CTL register fields

Field	Description
YTR_ACK	Acknowledge Y threshold. Writing a '1' to this bit clears the YTR flag and resets its interrupt condition. YTR signals the CPU to set new pointers for the next field. If YTR_ACK is not received by the time the active image area for the next field starts, the URUN flag is set. Data transfer continues with the old pointer values.
BFR1_INTEN BFR2_INTEN HBE_INTEN URUN_INTEN YTR_INTEN	Enable interrupt conditions. Enable corresponding interrupts to be generated when the BFR1_EMPTY, BFR2_EMPTY, HBE, URUN (underrun/end of transfer), and YTR (end of field/buffer) flags are set, respectively. Note: BFR2_INTEN, URUN_INTEN, YTR_INTEN must be 0 in message passing mode.
LTL_END	Little-endian. Specifies that data in SDRAM is stored in little-endian format. This only affects the overlay packed-image format interpretation in video-refresh modes. Refer to for details on byte ordering.
VO_ENABLE	Enable the EVO to send image data or message data to its output Note: This bit should not be simultaneously asserted with the RESET bit. The correct sequence to reset and enable the EVO is as follows. • Set all VO_CTL control fields as desired, writing VO_CTL with RESET = 1, VO_ENABLE = 0. • Retain all desired values of control fields, but rewrite VO_CTL with RESE T =0, VO_ENABL E =0. • Finally, still retaining all desired control fields, rewrite VO_CTL with RESET = 0, VO_ENABLE = 1. Setting VO_ENABLE in video-refresh modes starts the EVO sending image data beginning with the first pixel in the image. Setting VO_ENABLE in data-streaming and message-passing modes starts the EVO sending data beginning with the first byte in Buffer 1. In video-refresh and data-streaming modes, VO_ENABLE remains set until cleared by the CPU. In message-passing mode, VO_ENABLE is cleared when BFR1_EMPTY is set, indicating the end of message transfer. Note: De-asserting VO_ENABLE in video-refresh modes causes SDRAM reads to stop, but sync framing and BFR1_EMPTY generation and interrupts remain fully operational. The transmitted active image data is undefined in this case. To fully halt video output, a software reset is required.

7.16.3 VO-Related Registers

The VO-related registers and their fields are shown in . Their fields are unchanged from the TM-1000, however their function may vary depending upon the PNX1300 features that are selectively enabled by EVO_CTL (see).

Table 7-8. VO register fields

Register	Field	Description
VO_CLOCK	FREQUENCY	VO_CLK frequency. See DDS equation in , and PLL description in .
VO_FRAME	FRAME_LENGTH	Total number of lines per frame; the ending value of the Frame Line Counter; typically 525 or 625. Note: the Frame Line Counter counts from 1 to 525 or 625, consistent with CCIR 656 line numbering.
	FIELD_2_START	Start line number in the Frame Line Counter; where the second field of the frame begins . If non-interlaced pictures are desired, then the same value is programmed for Field 1 and Field 2. Field 1 becomes Fram e1 and Field2 becomes Fr ame2 .
	FRAME_PRESET	Value loaded into the Frame Line Counter when frame timing edge is received on VO_IO2.
VO_FIELD	F1_VIDEO_LINE	Line number in the Frame Line Counter of the first active video line of Field1 of the frame .
	F2_VIDEO_LINE	Line number in the Frame Line Counter of the first active video line of Fiel d2 of the frame If non-interlaced pictures are desired, this is programmed to the same value as F1_VIDEO_LINE
	F1_OLAP	Overlap of the SAV and EAV codes from Field 1 to Field 2. Overlap is defined as the delay in lines from start of blanking for Field 2 until SAV and EAV codes for Fiel d2 are emitted Typical values are +2 for 525/60 and +2 for 625/50.
	F2_OLAP	Overlap in lines of the SAV and EAV code from Fiel d2 to Fiel d1. Overlap is defined as the delay in lines from start of blanking for Field 1 until the SAV and EAV codes for Field 1 are emitted. Typical values are +3 for 525/60 and –2 for 625/50. The negative value means Field 1 blanking actually starts two lines before end of Fi eld2 of previous frame . This overlap is described in , and illustrated in .
VO_LINE	FRAME_WIDTH	Total line length in pixels including blanking. Also the ending value for the Frame Pixel Counter. Lines always begin with a horizontal blanking interval, and the image starts after the blanking interval and runs to the end of the line.
	VIDEO_PIXEL_START	Pixel number in Frame Pixel Counter of starting pixel of active video area within the line. Note: Must be even.

Note that the buffers must be 64-byte aligned, and a multiple of 64 samples in size (the six LSBs of AI_BASE1, AI_BASE2 and AI_SIZE are always '0').

The DSPCPU is required to assign a new, empty buffer to BASE1 and perform an ACK1, before buffer 2 fills up. Capture continues in buffer 2, until it fills up. At that time, BUF2_FULL is asserted, and capture continues in the new buffer 1, etc.

Upon receipt of an ACK, the AI hardware removes the related interrupt request line assertion at the next DSPCPU clock edge. Refer to for the rules regarding ACK and interrupt re-enabling. The AI interrupt should always be operated in level-sensitive mode, since AI can signal multiple conditions that each need independent ACKs over the single internal SOURCE 11 request line.

In normal operation, the DSPCPU and AI hardware continuously exchange buffers without ever loosing a sample. If the DSPCPU fails to provide a new buffer in time, the OVERRUN error flag is raised. This flag is *not affected* by ACK1 or ACK2; it can only be cleared by an explicit ACK_OVR.

8.8 POWER DOWN AND SLEEPLESS

The AI unit enters power down state whenever PNX1300 is put in global power down mode, except if the SLEEPLESS bit in AI_CTL is set. In the latter case, the unit continues DMA operation and will wake up the DSPCPU whenever an interrupt is generated.

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

It is recommended that AI be stopped (by negating AI_CTL.CAP_ENABLE) before block level power down is started, or that SLEEPLESS mode is used when global power down is activated.

8.9 HIGHWAY LATENCY AND HBE

The AI unit uses internal buffering before writing data to SDRAM. The internal buffer consists of one stereo sample input holding register and 64 bytes of internal buffer memory. Under normal operation, the 64-byte buffer is written to SDRAM while the input register receives another sample. This normal operation is guaranteed to be maintained as long as the highway arbiter is set to guarantee a latency for the AI unit that matches the sampling interval. Given a sample rate f_s, and an associated sample interval T (in nsec), the arbiter should be set to have a latency of at most T-20 nsec. Refer to for information on arbiter programming. If the requested latency is not adequate, the HBE (Highway Bandwidth Error) condition may result. This error flag gets set when the input register is full, the 64-byte buffer has not yet been written to memory, and a new sample arrives.

shows the required arbiter latency settings for a number of common operating modes. The rightmost column illustrates the nature of the resulting 64-byte highway requests. Is not necessary to compute arbiter settings, but they may be used to compute bus availability in a given interval.

Table 8-10. AI highway arbiter latency requirement examples

CapMode	f_s (kHz)	T (nS)	max arbiter latency (nsec)	access pattern
stereo 16 bits/sample	44.1	22,676	22,656	1 request every 362,812 nsec
stereo 16 bits/sample	48.0	20,833	20,813	1 request every 333,333 nsec
stereo 16 bits/sample	96.0	10,417	10,397	1 request every 166,667 nsec

8.10 ERROR BEHAVIOR

If either an OVERRUN or HBE error occurs, input sampling is temporarily halted, and samples will be lost. In case of OVERRUN, sampling resumes as soon as the DSPCPU makes one or more new buffers available through an ACK1 or ACK2 operation. In the case of HBE, sampling will resume as soon as the internal buffer is written to SDRAM.

HBE and OVERRUN are 'sticky' error flags. They will remain set until an explicit ACK_HBE or ACK_OVR.

8.11 DIAGNOSTIC MODE

Diagnostic mode is entered by setting the DIAGMODE bit in the AI_CTL register. In diagnostic mode, the AI_SCK, AI_WS and AI_SD inputs of the serial-parallel converter are taken from the output pins of the PNX1300 AO unit. This mode can be used during the diagnostic phase of system boot to verify correct operation of most of the AI unit and AO unit logic circuitry.

Note that the inputs are truly taken from the PNX1300 AO external pins, i.e. if an external (board level) source is driving AO_SCK or AO_WS, diagnostic mode is not capable of testing Audio Out.

Special care must be taken to enable diagnostic mode. The recommended way of entering diagnostic mode is:

- setup the AO unit such that an AO_SCK is generated
- set DIAGMODE bit followed by a 5 (AI_SCK) cycle delay
- perform a software reset of the AI unit and immediately set the DIAGMODE bit back to '1'.

table. The simplest way to generate these values in common computer languages such as C is as follows:

1. Generate the Increment Value as a floating point number = Input Width / Output Width
2. Multiply the Increment Value by 65536
3. Convert the result to a Long Integer (32 bits). The upper 16 bits of the Long integer will be the Integer increment value, and the lower 16 bits will be the Fractional value
4. Store the 32-bit Long integer in the parameter table as the combined Integer and Fractional increment values

For YUV 4:2:2 or YUV 4:2:0 input data and RGB output data, the scaling factor for U and V must be twice the scaling factor for Y, unless YUV4:2:2 sequencing is used for speed. In YUV 4:2:2 or YUV 4:2:0 data, the horizontal components of U and V are half those of Y. The U and V must be upscaled by 2 to generate a YUV 4:4:4 format internally for YUV to RGB conversion. For YUV 4:1:1 input data, the U and V components must be upscaled by a factor of 4 to generate the required internal YUV 4:4:4 format.

The Start Fraction defines the starting value in the scaling counter for each line. It is a 16-bit, two's complement fractional value between -0.500 and 0.49999+. The Start Fraction allows the input data to be offset by up to half a pixel, referred to the input pixel grid. It is '0' for Y and for UV co-sited data, and is set to '-0.25' (C000) for interspersed to co-sited conversion of U and V data. The '-0.25' value effectively shifts the U and V data toward the start of the line by 1/4 pixel, the amount required for conversion.

The Alpha 1 and Alpha 0 values are 8-bit fields within the 16-bit Alpha field. These values are loaded into the Alpha 1 and Alpha 0 registers, resp., for use by RGB 15+α and YUV 4:2:2+α overlay formats in alpha blending.

The Overlay start and end pixels and lines define the start and end pixels and lines within the output image for the overlay. The first pixel of the overlay image will be blended with the pixel at the Overlay Start Pixel and Overlay Start Line in the output image.

14.6.11.3 Control word format

The Control word provides bit fields which affect the horizontal filtering operation. The format of the Control word is as follows.

Bits	Name	Function
15	Bypass	Normally set to 0 to enable filtering. Can be set to 1 to accomplish data move without filtering.
14	422SEQ	4:2:2 Sequence bit. Used with YUV 4:2:2 output
13	YUV420	YUV 4:2:0 input format
12	OEN	Overlay enable. Valid only for PCI output
11	PCI	PCI output enable. Otherwise SDRAM output
10	BEN	Bit mask enable. Valid only for PCI output
9	GETB	Large down scaling bit. Picks five input pixels nearest 5 output pixels and passes to filter. Equivalent to filter bypass + 5-tap filter of output pixels. LSB value = 0 for filtering.
8	OLLE	Overlay little endian enable
7-6	OFRM	Overlay format 0 = RGB 24+α 1 = RGB 15+α 2 = YUV 4:2:2+α
5	CHK	Chroma keying enable
4	LE	RGB output little endian enable
3-0	RGB	RGB Output Code 0 = YUV 4:2:2+α 1 = YUV 4:2:2 2 = RGB 24+α 3 = RGB 24 packed 4 = RGB 8A (RGB 233) 5 = RGB 8R (RGB 332) 6 = RGB15+α 7 = RGB 16

The 422SEQ bit controls the internal sequencing of the YUV to RGB operation. It is set to '1' when YUV 4:2:2 output is selected. When 422SEQ is '0', normal RGB output is assumed. In this mode, the input is YUV 4:2:2 or YUV 4:2:0, and the output is RGB. To generate the RGB output, the YUV 4:2:2 or YUV 4:2:0 input must be upscaled to YUV 4:4:4 before conversion to RGB. This means the scaling factor for U and V must be twice the scaling factor for Y. The internal sequencing of the filter in this case is UVY, UVY, UVY to generate RGB, RGB, RGB. For YUV 4:2:2 output formats, no upscaling of U and V is required. In this case, the 422SEQ bit is set to one, and the filter sequence is UVYY, UVYY, UVYY.

The 422SEQ bit can be set in RGB output mode to decrease the processing time for the image at the expense of color bandwidth and some corresponding decrease in picture quality. If the 422SEQ bit is set for RGB output, the filter will perform the UVYY sequence. In this case, the U and V components are not upscaled by 2, and the YUV to RGB converter updates its U and V components every other pixel. In the normal case (422SEQ=0), it takes 6 clock cycles to generate two RGB pixels. In the 422SEQ=1 case, it takes 4 clock cycles to generate two RGB pixels, reducing processing time by 33%.

The YUV420 bit indicates that the input data is in YUV 4:2:0 format. In YUV 4:2:0 format, the U and V components are half the width and half the height of the Y data. YUV 4:2:0 data is normally converted to YUV 4:2:2 data by a separate vertical upscaling by a factor of 2.0 for best quality. The YUV420 bit allows using YUV 4:2:0 data directly but with some quality degradation. When YUV420 is set, the ICP up scales the data vertically by line duplication. Each U and V input line is used twice. The sepa-

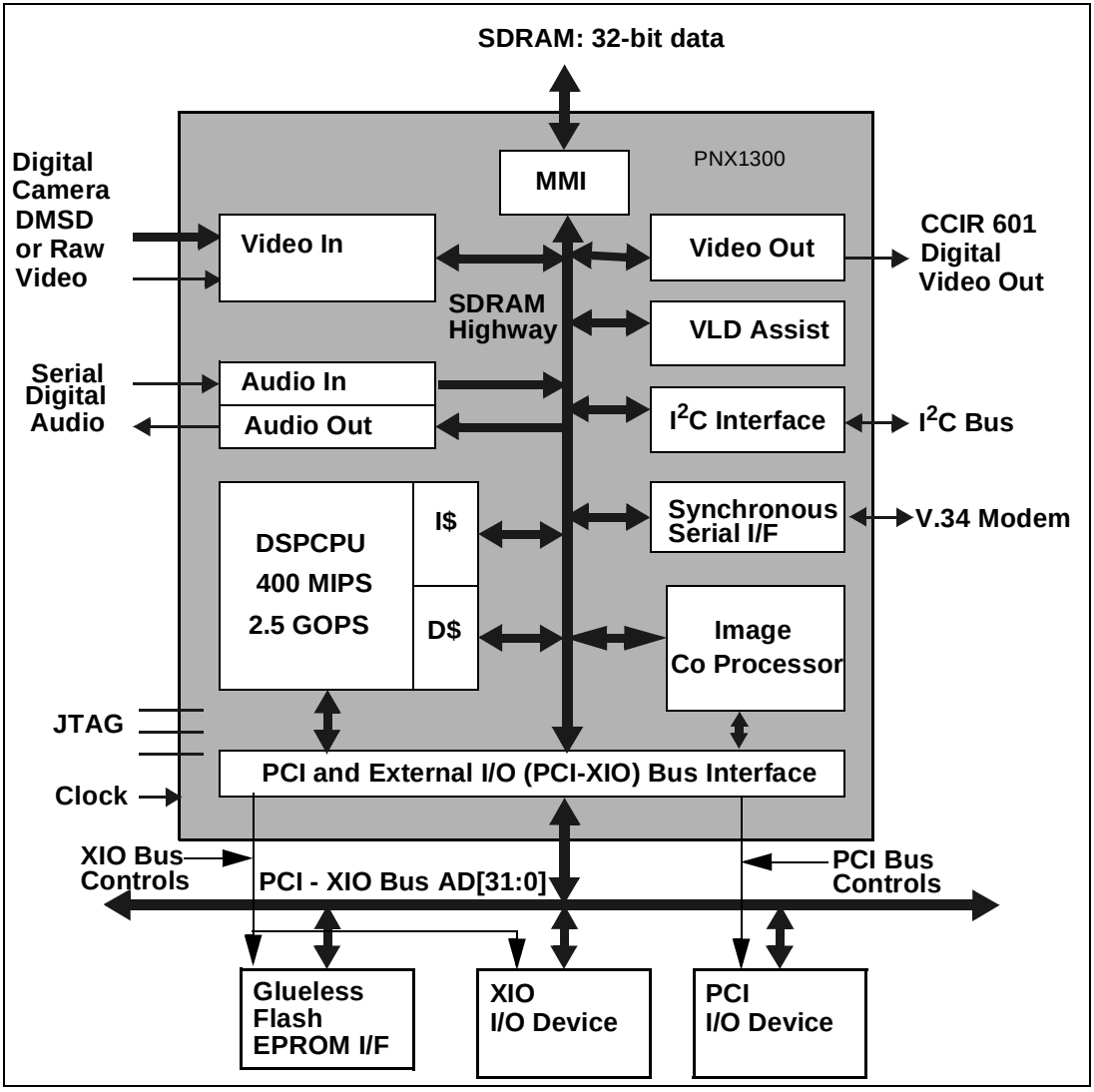


Figure 22-1. Partial PNX1300 chip block diagram

PCI transfer and is incremented for each PCI word transferred.

The XIO Bus does not generate parity during XIO Bus write transfers or check parity during XIO Bus read transfers. This allows the XIO Bus to interface to standard 8-bit devices without having to add parity-generation and check logic. While the XIO Bus is active, the XIO Bus logic inhibits parity checking and drives the PCI Parity and Parity Error pins so that they do not float.

Word transfer is used to transfer the bytes to and from the PCI bus for hardware simplicity. The primary intended use of the PCI-XIO Bus is for slow devices, ROMs, flash EPROMs and I/O. Because the PCI-XIO bus is so much slower than the PNX1300, there is time available for the PNX1300 to pack and unpack the words. In the case of ROMs and flash EPROMs, the data is typically compressed, requiring the PNX1300 CPU to both unpack and decompress the data.

The PCI-XIO Bus Controller logic reconfigures the byte enables as control signals for the attached XIO Bus chips during XIO Bus transfers. It also drives the PCI_TRDY# signal to the PCI Bus for each transfer. The PCI Bus byte enables are reconfigured to generate XIO Bus timing signals: Read (IORD), Write (IOWR) and Data Strobe (DS). These signals allow ROM, flash EPROM, 68K and x86 devices to be gluelessly interfaced to the XIO Bus. For a single device, the PCI_INTB# line is used as the global

Compute carry bit from unsigned add

carry

SYNTAX

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

FUNCTION

if r*guard* **then {**
  if (r*src1*+r*src2*) < 2^{32} **then**
    r*dest* ← 0
  else
    r*dest* ← 1
}

ATTRIBUTES

Function unit	alu
Operation code	45
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

DESCRIPTION

The `carry` operation computes the unsigned sum of the first and second arguments, r*src1*+r*src2*. If the sum generates a carry (if the sum is greater than 2^{32}-1), 1 is stored in the destination register, r*dest*; otherwise, r*dest* is set to 0.

The `carry` 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
r70 = 2, r30 = 0xfffffffc	`carry r70 r30 → r80`	r80 ← 0
r10 = 0, r70 = 2, r30 = 0xfffffffc	`IF r10 carry r70 r30 → r90`	no change, since guard is false
r20 = 1, r70 = 2, r30 = 0xfffffffc	`IF r20 carry r70 r30 → r100`	r100 ← 0
r60 = 4, r30 = 0xfffffffc	`carry r60 r30 → r110`	r110 ← 1
r30 = 0xfffffffc	`carry r30 r30 → r120`	r120 ← 1

Read clock cycle counter, least-significant word

cycles

SYNTAX

```
[ IF rguard ] cycles → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← CCCOUNT<31:0>

ATTRIBUTES

Function unit	fcomp
Operation code	154
Number of operands	0
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

DESCRIPTION

Refer to for a description of the CCCOUNT operation. The `cycles` operation copies the low 32 bits of the slave register of Clock Cycle Counter (CCCOUNT) to the destination register, r*dest*. The contents of the master counter are transferred to the slave CCCOUNT register only on a successful interruptible jump and on processor reset. Thus, if `cycles` and `hicycles` are executed without intervening interruptible jumps, the operation pair is guaranteed to be a coherent sample of the master clock-cycle counter. The master counter increments on all cycles (processor-stall and non-stall) if PCSW.CS = 1; otherwise, the counter increments only on non-stall cycles.

The `cycles` 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
CCCOUNT_HR = 0xabcdefff12345678	`cycles → r60`	r30 ← 0x12345678
r10 = 0, CCCOUNT_HR = 0xabcdefff12345678	`IF r10 cycles → r70`	no change, since guard is false
r20 = 1, CCCOUNT_HR = 0xabcdefff12345678	`IF r20 cycles → r100`	r100 ← 0x12345678

h_dspiabs

Clipped signed absolute value

SYNTAX

```
[ IF rguard ] h_dspiabs r0 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc2 >= 0 then
      rdest ← rsrc2
   else if rsrc2 = 0x80000000 then
      rdest ← 0x7fffffff
   else
      rdest ← –rsrc2
}
```

ATTRIBUTES

Function unit	dspalu
Operation code	65
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

DESCRIPTION

The `h_dspiabs` operation computes the absolute value of rsrc2, clips the result into the range [0x0..0x7fffffff], and stores the clipped value into r*dest*. All values are signed integers. This operation requires a zero as first argument. The programmer is advised to use the unary pseudo operation `dspiabs` instead.

The `h_dspiabs` 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 = 0xffffffff	`h_dspiabs r0 r30 → r60`	r60 ← 0x00000001
r10 = 0, r40 = 0x80000001	`IF r10 h_dspiabs r0 r40 → r70`	no change, since guard is false
r20 = 1, r40 = 0x80000001	`IF r20 h_dspiabs r0 r40 → r100`	r100 ← 0x7fffffff
r50 = 0x80000000	`h_dspiabs r0 r50 → r80`	r80 ← 0x7fffffff
r90 = 0x7fffffff	`h_dspiabs r0 r90 → r110`	r110 ← 0x7fffffff

ifixieee

Convert floating-point to integer using PCSW rounding mode

SYNTAX

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

FUNCTION

if r*guard* **then {**
  r*dest* ← (long) ((float)r*src1*)
}

ATTRIBUTES

Function unit	falu
Operation code	121
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

DESCRIPTION

The `ifixieee` operation converts the single-precision IEEE floating-point value in r*src1* to a signed integer and writes the result into r*dest*. Rounding is according to the IEEE rounding mode bits in PCSW. If r*src1* is denormalized, zero is substituted before conversion, and the IFZ flag in the PCSW is set. If `ifixieee` causes an IEEE exception, such as overflow or underflow, 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 `ifixieeeflags` operation computes the exception flags that would result from an individual `ifixieee`.

The `ifixieee` 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)	`ifixieee r30 → r100`	r100 ← 3
r35 = 0x40247ae1 (2.57)	`ifixieee r35 → r102`	r102 ← 3, INX flag set
r10 = 0, r40 = 0xff4fffff (–3.402823466e+38)	`IF r10 ifixieee r40 → r105`	no change, since guard is false
r20 = 1, r40 = 0xff4fffff (–3.402823466e+38)	`IF r20 ifixieee r40 → r110`	r110 ← 0x80000000 (-2^{31}), INV flag set
r45 = 0x7f800000 (+INF))	`ifixieee r45 → r112`	r112 ← 0x7fffffff (2^{31}-1), INV flag set
r50 = 0xbfc147ae (-1.51)	`ifixieee r50 → r115`	r115 ← -2, INX flag set
r60 = 0x00400000 (5.877471754e-39)	`ifixieee r60 → r117`	r117 ← 0, IFZ set
r70 = 0xffffffff (QNaN)	`ifixieee r70 → r120`	r120 ← 0, INV flag set
r80 = 0xffbfffff (SNaN)	`ifixieee r80 → r122`	r122 ← 0, INV flag set

mergemsb

Merge most-significant byte

SYNTAX

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

FUNCTION

if r*guard* **then {**
r*dest*<7:0> ← r*src2*<23:15>
r*dest*<15:8> ← r*src1*<23:15>
r*dest*<23:16> ← r*src2*<31:24>
r*dest*<31:24> ← r*src1*<31:24>
}

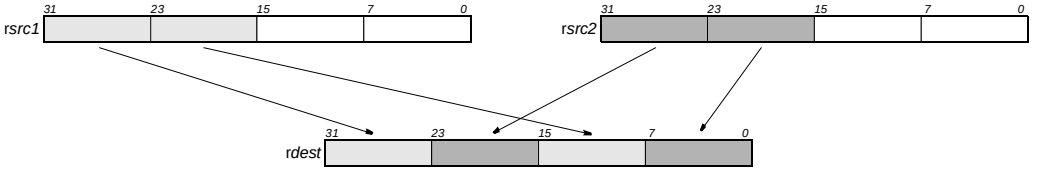
ATTRIBUTES

Function unit	alu
Operation code	58
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

DESCRIPTION

As shown below, the `mergemsb` operation interleaves the two pairs of most-significant bytes from the arguments r*src1* and r*src2* into r*dest*. The second-most-significant byte from r*src2* is packed into the least-significant byte of r*dest*; the second-most-significant byte from r*src1* is packed into the second-least-significant byte of r*dest*; the most-significant byte from r*src2* is packed into the second-most-significant byte of r*dest*; and the most-significant byte from r*src1* is packed into the most-significant byte of r*dest*.



The `mergemsb` 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 = 0x12345678, r40 = 0xaabbccdd	`mergemsb r30 r40 → r50`	r50 ← 0x12aa34bb
r10 = 0, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r10 mergemsb r40 r30 → r60`	no change, since guard is false
r20 = 1, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r20 mergemsb r40 r30 → r70`	r70 ← 0xaa12bb34

prefd

prefetch with displacement

SYNTAX

```
[ IF rguard ] prefd(d) rsrc1
```

FUNCTION

if r*guard* **then {**
cache_block_mask = ~(cache_block_size - 1)
data_cache <- mem[(rsrc1 + d) & cache_block_mask]
}

ATTRIBUTES

Function unit	dmemspec
Operation code	209
Number of operands	1
Modifier	7 bits
Modifier range	–256..252 by 4
Latency	-
Issue slots	5

SEE ALSO

DESCRIPTION

The prefd operation loads the one full cache block size of memory value from the address computed by ((rsrc1+d) & 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 prefd operation will not be executed when the cache is already occupied with 2 cache misses, when the operation is issued.

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

EXAMPLES

Initial Values	Operation	Result
r10 = 0xabcd, cache_block_size = 0x40	`prefd(0xd) r10`	Loads a cache line for the address space from 0xabc0 to 0x0xabff from the main memory. If the data is already in the cache, the operation is not executed.
r10 = 0xabcd, r11 = 0, cache_block_size = 0x40	`IF r11 prefd(0xd) r10`	since guard is false, prefd operation is not executed
r10 = 0xabff, r11 = 1, cache_block_size = 0x40	`IF r11 prefd(ox1) r10`	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.

16-bit store

pseudo-op for h_st16d(0)

st16

SYNTAX

```
[ IF rguard ] st16 rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
   if PCSW.bytesex = LITTLE_ENDIAN then
      bs ← 1
   else
      bs ← 0
   mem[rsrc1 + (1 ⊕ bs)] ← rsrc2<7:0>
   mem[rsrc1 + (0 ⊕ bs)] ← rsrc2<15:8>
}
```

ATTRIBUTES

Function unit	dmem
Operation code	30
Number of operands	2
Modifier	No
Modifier range	—
Latency	n/a
Issue slots	4, 5

SEE ALSO

DESCRIPTION

The `st16` operation is a pseudo operation transformed by the scheduler into an `h_st16d(0)` with the same arguments. (Note: pseudo operations cannot be used in assembly files.)

The `st16` operation stores the least-significant 16-bit halfword of r*src2* into the memory locations pointed to by the address in r*src1*. This store operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

If `st16` is misaligned (the memory address in r*src1* is not a multiple of 2), the result of `st16` is undefined, and the MSE (Misaligned Store Exception) bit in the PCSW register is set to 1. Additionally, if the TRPMSE (TRaP on Misaligned Store Exception) bit in PCSW is 1, exception processing will be requested on the next interruptible jump.

The result of an access by `st16` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `st16` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the addressed memory locations (and the modification of cache if the locations are cacheable). If the LSB of r*guard* is 1, the store takes effect. If the LSB of r*guard* is 0, st16 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	`st16 r10 r80`	[0xd00] ← 0x22, [0xd01] ← 0x11
r50 = 0, r20 = 0xd01, r70 = 0xaabbccdd	`IF r50 st16 r20 r70`	no change, since guard is false
r60 = 1, r30 = 0xd02, r70 = 0xaabbccdd	`IF r60 st16 r30 r70`	[0xd02] ← 0xcc, [0xd03] ← 0xdd

uimm

Unsigned immediate

SYNTAX

`uimm(`*n*`)` → *rdest*

FUNCTION

r*dest* ← *n*

ATTRIBUTES

Function unit	const
Operation code	191
Number of operands	0
Modifier	32 bits
Modifier range	0..0xffffffff
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

DESCRIPTION

The `uimm` operation writes the unsigned 32-bit opcode modifier *n* into r*dest*. Note: this operation is not guarded.

EXAMPLES

Initial Values	Operation	Result
	`uimm(2)` → `r10`	r10 ← 2
	`uimm(0x100)` → `r20`	r20 ← 0x100
	`uimm(0xfffc0000)` → `r30`	r30 ← 0xfffc0000

ume8ii

Unsigned sum of absolute values of signed 8-bit differences

SYNTAX

```
[ IF rguard ] ume8ii rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**

r*dest* ← abs_val(sign_ext8to32(r*src1*<31:24>) – sign_ext8to32(r*src2*<31:24>)) +
abs_val(sign_ext8to32(r*src1*<23:16>) – sign_ext8to32(r*src2*<23:16>)) +
abs_val(sign_ext8to32(r*src1*<15:8>) – sign_ext8to32(r*src2*<15:8>)) +
abs_val(sign_ext8to32(r*src1*<7:0>) – sign_ext8to32(r*src2*<7:0>))

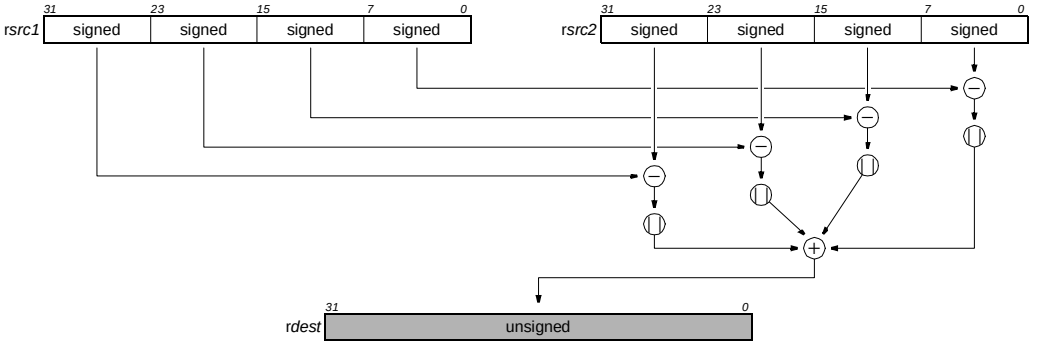
ATTRIBUTES

Function unit	dspalu
Operation code	64
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

DESCRIPTION

As shown below, the ume8ii operation computes four separate differences of the four pairs of corresponding signed 8-bit bytes of r*src1* and r*src2*; the absolute values of the four differences are summed, and the sum is written to r*dest*. All computations are performed without loss of precision.



The ume8ii 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	`ume8ii r80 r30 → r100`	r100 ← 0x14
r10 = 0, r80 = 0x0a14f6f6, r30 = 0x1414ecf6	`IF r10 ume8ii r80 r30 → r70`	no change, since guard is false
r20 = 1, r90 = 0x64649c9c, r40 = 0x649c649c	`IF r20 ume8ii r90 r40 → r110`	r110 ← 0x190
r40 = 0x649c649c, r90 = 0x64649c9c	`ume8ii r40 r90 → r120`	r120 ← 0x190
r50 = 0x80808080, r60 = 0x7f7f7f7f	`ume8ii r50 r60 → r125`	r125 ← 0x3fc

Index

Numerics

A