PATRIOT
SCIENTIFIC CORPORATION

PSC1000™ Microprocessor Reference Manual

Patriot Scientific Corporation
10989 Via Frontera
San Diego, CA 92127
1 (619) 674 5000 voice
1 (619) 674 5005 fax
www.ptsc.com

99-0370001

PSC1000™ Microprocessor

Reference Manual

ADVANCE INFORMATION

Patriot Scientific Corporation
10989 Via Frontera
San Diego, CA 92127
1 (619) 674 5000 voice
1 (619) 674 5005 fax
www.ptsc.com

ADVANCE INFORMATION

DISCLAIMER

Patriot Scientific Corporation (PSC) reserves the right to make changes to its products or specifications at any time, or to discontinue any product, without notice. PSC advises its customers to obtain the latest product information available before designing-in or purchasing its products. PSC assumes no responsibility for the use of any circuitry described other than the circuitry embodied in a PSC product. PSC makes no representations that the circuitry described herein is free from patent infringement or other rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent, patent rights or other rights, of PSC.

Information within this document is subject to change without notice, but was believed to be accurate at the time of publication. No warranty of any kind, including but not limited to implied warranties of merchantability or fitness for a particular application, are stated or implied. PSC and the author assume no responsibility for any errors or omissions, and disclaims responsibility for any consequences resulting from the use of the information included herein.

PATRIOT
SCIENTIFIC CORPORATION

PSC1000™ Microprocessor Reference Manual

ADVANCE INFORMATION

by George William Shaw

Printing Date: 1999 March 25

Text, tables, and illustrations by George W. Shaw
Edited by Jeffrey Conroy

For company and product information, access www.ptsc.com. Patriot Scientific Corporation is publicly traded over the counter, symbol PTSC.

ShBoom and PSC1000 are trademark of Patriot Scientific Corporation. Any other brands and products used within this document are trademarks or registered trademarks of their respective owners.

The technology discussed in this document may be covered by one or more of the following US patents: 5,440,749; 5,530,890; 5,604,915; 5,659,703; 5,784,584. Other US and Foreign patents pending.

LIFE SUPPORT POLICY

Patriot Scientific Corporation's (PSC) products are not authorized for use as critical components in life-support appliances, devices or systems. Such use requires a specific written agreement signed by the appropriate PSC officer. Life-support devices or systems are devices or systems which (a) are intended for surgical implant into the body or (b) support or sustain life and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in significant injury to the user. A critical component is any component of a life- support device or system whose failure to perform can be reasonably expected to cause the failure of the life-support device or system, or to affect its safety or effectiveness. Use of PSC products in such applications is understood to be fully at the risk of the customer.

Contents

ADVANCE INFORMATION

ADVANCE INFORMATION

Figures

ADVANCE INFORMATION

Tables

ADVANCE INFORMATION

ADVANCE INFORMATION

Documentation Typography and Nomenclature

References to software commands, CPU instructions, registers, register fields, and package pins are in a different font than body text to minimize confusion and to distinguish them from the surrounding text. Specifically:

Processor instructions are in lowercase (e.g., “The `mloop` repeats `refresh` and `delay`, …”).

Registers or register fields are also in lowercase (e.g., “`msra` contains data used during…”). Contextually, use of a register or register field name can also imply its contents, (e.g. “…must contain the sum of `mgebtdobe` and `mgebtcase`”). When referring to a register or register field whose function is identical among its variants, X is used to hold the place of the identifying alpha or numeric character within the name (e.g. `ioXebt`).

Package pins are in uppercase (e.g., “…the timing for the $\overline{\text{CAS}}$ inactive portion, also referred to as CAS precharge…”). When referring to a pin whose function is identical among its variants, x is used to hold the place of the identifying alpha or numeric character within the name (e.g., $\overline{\text{CASx}}$). The over bar or a prefix “–” on signal names indicates the signal is active in its low state; otherwise, signals are active high or the active state is not relevant (e.g. $\overline{\text{RAS}}$ and –RAS refer to the same signal).

To avoid confusion regarding the width in bits of a “word”, the term “cell” is used to denote the full processor data element size of 32 bits.

PRODUCT PREVIEW indicates that the product is in the conceptual or design phase of development, and that the document represents the design goals for the product, which may change without notice before the product goes into production.

ADVANCE INFORMATION indicates that the product is in the sampling or pre-production phase of development and that data and specifications are preliminary and subject to change without notice.

PSC1000 Microprocessor

PATRIOT SCIENTIFIC CORPORATION

32-BIT RISC PROCESSOR

ADVANCE INFORMATION

Features

♦ Low-System-Cost 32-Bit RISC Microprocessor
♦ Runs Java™ at Native Speed
♦ Multiple Language Support
♦ Dual-Processor Architecture
- Microprocessing Unit (MPU)
  High-performance zero-operand dual-stack architecture
- Virtual Peripheral Unit (VPU)
  Performs timing, time-synchronous data transfers, bit outputs, DRAM refresh, emulates peripherals

♦ 4-Gigabyte Physical Address Space
♦ Internal Clock Multiplier
- 2X CPU clock, 4X Bus timing

♦ 4-Group Memory/Bus Interface
- Supports any combination of EPROM, SRAM, DRAM, VRAM
- Programmable memory and I/O timing

♦ Virtual Memory Support
♦ 8-Level Interrupt Controller
♦ 8-Level Direct Memory Access Controller
♦ 16 I/O bits
♦ 52 General-Purpose 32-Bit Registers
♦ “Glueless” System Interface
♦ Big Endian Byte Ordering
♦ Small, Low-Cost, 100-Pin TQFP Package

General Description

The PSC1000 microprocessor is a highly integrated 32-bit RISC processor that offers high performance and low power consumption at low system cost for a wide range of embedded applications. It is a highly integrated 32-bit RISC processor with a peak performance of one instruction per CPU-clock cycle. The 32-bit registers and data paths fully support 32-bit addresses and data types. The processor addresses up to four gigabytes of physical memory, and supports virtual memory with the use of external mapping logic.

As an implementation of the ShBoom™ Microprocessor architecture, the PSC1000 CPU architectural philosophy is that of simplification and efficiency of use. A zero-operand design eliminates most operand bits and the decoding time and instruction space they require. Instructions are shrunk to eight bits, significantly increasing instruction bandwidth and reducing program size. By not using pipeline or superscalar execution, the resulting control simplicity increases execution speed to issue *and* complete an instruction in a single clock cycle—as often as every clock cycle—without a conventional instruction cache. To ensure a low-cost chip, a data cache and its cost are also eliminated in favor of efficient register caches.

The stack architectures of the PSC1000 microprocessor and the Java Virtual Machine are very similar. This results in only a relatively simple byte code translator (20K) being required to produce executable native code from Java byte code, rather than a full Just-in-Time (JIT) compiler (200–400K). The result is *much* faster initial execution of Java programs and significantly smaller memory requirements. Further, most modern languages are implemented on a stack model. The features that allow the PSC1000 to run Java efficiently apply similarly to other languages such as C, Forth and Postscript..

The PSC1000 CPU operates up to four groups of programmable bus configurations from as fast as two CPU clocks to as slow as 82 CPU clocks, allowing any desired mix of high-speed and low-speed memory. Minimum system cost is reduced, thus allowing the system designer to trade system cost for performance as needed.

By incorporating many on-chip system functions and a “glueless” bus interface, support chips are eliminated, further lowering system cost. The CPU includes an MPU, a Virtual Peripheral Unit, a DMA controller, an interrupt controller, bit inputs, bit outputs, and a programmable memory interface. It can operate with 32-bit-wide or 8-bit-wide memory and devices, and includes hardware debugging support. A minimum system consists of a PSC1000 CPU, an 8-bit-wide EPROM, an oscillator, and optionally one x8 or two x16 memories—a total of 4 or 5 active components. The small die, which contains only 137 500 transistors, produces a high-performance, low-cost CPU, and a high level of integration produces a high-performance, low-cost system.

PATRIOT SCIENTIFIC CORPORATION

PSC1000 Microprocessor

32-BIT RISC PROCESSOR

ADVANCE INFORMATION

FEATURES

MICROPROCESSING UNIT (MPU)

- Zero-operand dual-stack architecture
- Very similar to Java Virtual Machine
- 12.5-ns instruction cycle
- 52 General-Purpose 32-Bit Registers
  - 16 global data registers (`g0–g15`)
  - 16 local registers (`r0–r15`) double as return stack cache
    - `r0` is an index register with predecrement and postincrement
    - Automatic local-register stack spill and refill
  - 18 operand stack cache registers (`s0–s17`)
    - `s0` is an address register
    - Automatic operand stack spill and refill
  - Index register (`x`) with predecrement and postincrement
  - Count register (`ct`)
- Stack paging traps
- Cache-management instructions
- MPU communicates with DMA and VPU via global registers
- Hardware single- and double-precision IEEE floating-point support
- Fast multiply
- Fast bit-shifter
- Hardware single-step and breakpoint
- Virtual-memory support
- Posted write
- Power-fail status bit
- Instruction-space-saving 8-bit opcodes

DIRECT MEMORY ACCESS CONTROLLER (DMAC)

- Eight prioritized DMA channels
- Fixed or revolving DMA priorities
- Byte, four-byte or cell DMA devices
- Single or back-to-back DMA requests
- Transfer rates to 200 MB/second
- Programmable timing for each channel
- Interrupt MPU on transfer boundary/count reached
- Terminate DMA on transfer boundary/count reached
- Channels can be configured as event counters
- DMA communicates with MPU and VPU via global registers

VIRTUAL PERIPHERAL UNIT (VPU)

- Executes instruction stream independent of MPU
- Deterministic execution
- Performs timing, time-synchronous data transfers, bit-output operations, DRAM refresh
- Emulates peripherals like serial I/O, A to D, D to A, PWM, timers
- Eight transfer channels
- Byte, four-byte or cell device transfers
- Programmable timing for each channel
- Interrupt MPU on transfer boundary/count reached
- Set/reset output bits
- Set MPU interrupt
- Test and branch on input bit
- Looping instructions
- Load transfer address, direction, interrupt on boundary
- VPU communicates with DMA and MPU via global registers or memory
- Channels can be configured as timers
- Instruction-space-saving 8-bit opcodes

INPUT-OUTPUT/INTERRUPTS

- Eight bit inputs
  - Bits can be configured as zero-persistent
  - Register- and bit-addressable
- Eight bit outputs
  - Register- and bit-addressable
- I/O bits available on pins or multiplexed on bus
- Eight prioritized and vectored interrupts

PROGRAMMABLE MEMORY INTERFACE (MIF)

- Programmable bus interface timing to 1/4 external clock
- Four independently configurable memory groups:
  - Any combination of 32-bit and 8-bit devices
  - Any combination of EPROM, SRAM, DRAM, VRAM
  - Almost any DRAM size/configuration
  - Fast-page mode access for each DRAM group
  - Glueless support for one memory bank per group
  - 1.25 gates per memory bank for decoding up to 16 memory banks (four per memory group)
- Virtual-memory support
- DRAM refresh support (via VPU)
- VRAM support includes DSF, $\overline{\text{OE}}$, $\overline{\text{WE}}$, $\overline{\text{CAS}}$ before $\overline{\text{RAS}}$ control

Table 1. Signal Descriptions

SYMBOL	TYPE	DESCRIPTION
cV_{SS}	PWR	Ground for core logic and all output driver pre-drivers.
cV_{CC}	PWR	Power for core logic and all output driver pre-drivers.
$ctrlV_{SS}$	PWR	Ground for control signal output drivers (DSF, OUT[7:0], all RASes, all CASes, $\overline{\text{DOB}}$, $\overline{\text{OE}}$, $\overline{\text{xWE}}$).
$ctrlV_{CC}$	PWR	Power for control signal output drivers (DSF, OUT[7:0], all RASes, all CASes, $\overline{\text{DOB}}$, $\overline{\text{OE}}$, $\overline{\text{xWE}}$).
adV_{SS}	PWR	Ground for AD[31:0] output drivers.
adV_{CC}	PWR	Power for AD[31:0] output drivers.
CLK	I	**EXTERNAL OSCILLATOR:** The CPU operating frequency is twice the external oscillator frequency.
$\overline{\text{RESET}}$	I A( )	**RESET:** Asserting $\overline{\text{RESET}}$ causes the entire CPU to be initialized and the MPU and VPU to begin execution at their hardware reset locations. If $\overline{\text{RESET}}$ is not held low during power-up, the signal also is input on AD8 during $\overline{\text{RAS}}$ active and $\overline{\text{CAS}}$ inactive, and $\overline{\text{RESET}}$ is ignored.
DSF	O I(L)	**DEVICE SPECIAL FUNCTION:** Set on VRAM memory cycles during $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$ accesses by the MPU to control VRAM function.
$\overline{\text{MFLT}}$	I S($\overline{\text{RAS}}$)	**MEMORY FAULT:** Asserted by external memory-management hardware before $\overline{\text{RAS}}$ active to invalidate the current MPU bus cycle and cause the MPU to trap if the configuration bit pkgmflt is set. The signal also is input on AD8 at $\overline{\text{RAS}}$ fall during $\overline{\text{CAS}}$ inactive, if the bit pkgmflt is clear.
$\overline{\text{IN}}$[7:0]	I A( )	**INPUTS:** Asserted by external hardware to request an interrupt or DMA, or to input a bit, when the configuration bit pkgio is set. The bits alternatively are input on AD[7:0] during $\overline{\text{RAS}}$ active and $\overline{\text{CAS}}$ inactive, if the bit pkgio is clear.
OUT[7:0]	O I(H)	**OUTPUTS:** Bit outputs writable from the VPU or MPU. These bits are also available on AD[7:0] during $\overline{\text{RAS}}$ inactive.
$\overline{\text{RAS}}$	O I(L)	**ROW ADDRESS STROBE:** A control signal asserted to define row address valid and deasserted only when another row address cycle is required.
RAS	O, I(H)	Inverted $\overline{\text{RAS}}$.
$\overline{\text{CAS}}$	O I(H)	**COLUMN ADDRESS STROBE:** A control signal asserted to define column address valid and deasserted at the end of the current bus cycle.
CAS	O, I(L)	Inverted $\overline{\text{CAS}}$.

ADVANCE INFORMATION

Table 1. Signal Descriptions (continued)

SYMBOL	TYPE	DESCRIPTION
$\overline{\text{MGS0…3}}$/ $\overline{\text{RAS0…3}}$	O I(L)	**MEMORY GROUP SELECTS/ROW ADDRESS STROBES:** In multiple memory bank (MMB) mode (configuration bit mmb is set), the strobes are active during all bus cycles for the entire bus cycle. In single memory bank (SMB) mode, they are similar to $\overline{\text{RAS}}$.
$\overline{\text{CAS0-3}}$	O I(H)	**COLUMN ADDRESS STROBES:** Similar to $\overline{\text{CAS}}$, to assert a column address cycle on the specified memory bank within the current memory group.
$\overline{\text{OE}}$	O I(H)	**OUTPUT ENABLE:** Active when the current bus transaction is a read from memory. The configuration bit oed is set or cleared during the CPU reset startup process.
$\overline{\text{EWE}}$	O I(H)	**EARLY WRITE ENABLE:** Active when the current bus transaction is a write to memory. Active time at either start of cycle or $\overline{\text{CAS}}$ fall is programmable for each memory group.
$\overline{\text{LWE}}$	O I(H)	**LATE WRITE ENABLE:** Active when the current bus transaction is a write to memory and for VRAM control. Active time either at or after $\overline{\text{DOB}}$ active is programmable for each memory group.

AD[31:0]	I/O
	S($\overline{\text{DOB}}$)
	S($\overline{\text{RAS}}$)
	A( )
	I(Z)
		ADDRESS DATA BUS: Multiplexed address, data, I/O and control bus.
		For data.
		For alternate memory fault on AD8.
		For alternate reset on AD8. See $\overline{\text{RESET}}$.

Notes:

I =	Input-Only Pins	A( ) =	Asynchronous inputs	I(H) =	high value on reset
O =	Output-Only Pins	S(sym) =	Synchronous inputs must meet setup and hold requirements relative to symbol.	I(L) =	low value on reset
I/O =	Bidirectional Pins			I(Z) =	high impedance on reset
PWR =	Power Pin				

ADVANCE INFORMATION

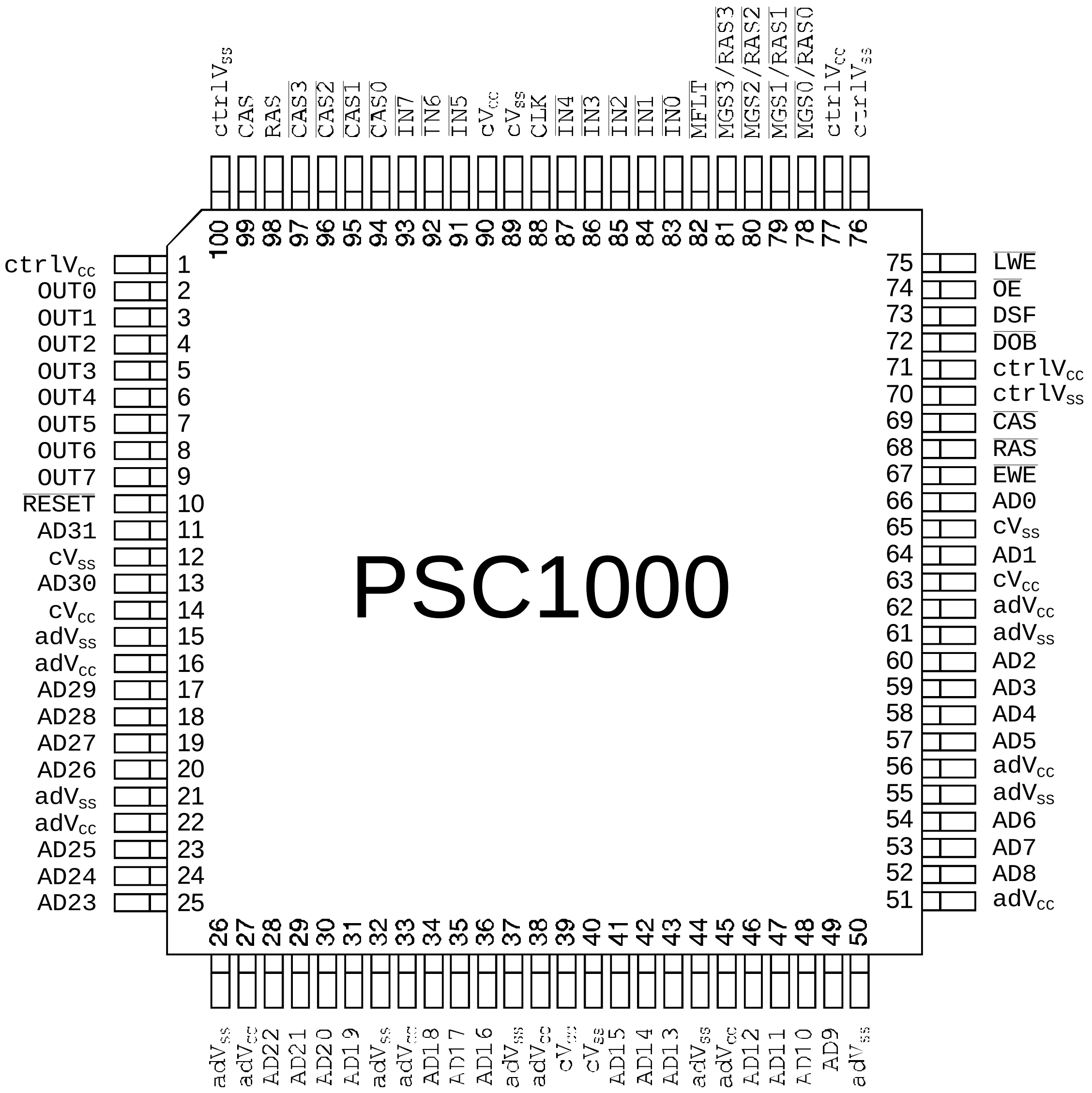


Figure 1. 100-Pin Thin Quad Flat Package (TQFP)

ADVANCE INFORMATION

Table 2. Pin Assignments, 100-Pin TQFP

PIN NO.	PIN NAME	TYPE	PIN NO.	PIN NAME	TYPE	PIN NO.	PIN NAME	TYPE
1	ctrlV_{cc}	PWR	35	AD17	I/O	69	$\overline{CAS}$	O
2	OUT0	O	36	AD16	I/O	70	ctrlV_{ss}	PWR
3	OUT1	O	37	adV_{ss}	PWR	71	ctrlV_{cc}	PWR
4	OUT2	O	38	adV_{cc}	PWR	72	$\overline{DOB}$	O
5	OUT3	O	39	cV_{cc}	PWR	73	DSF	O
6	OUT4	O	40	cV_{ss}	PWR	74	$\overline{OE}$	O
7	OUT5	O	41	AD15	I/O	75	$\overline{LWE}$	O
8	OUT6	O	42	AD14	I/O	76	ctrlV_{ss}	PWR
9	OUT7	O	43	AD13	I/O	77	ctrlV_{cc}	PWR
10	$\overline{RESET}$	I	44	adV_{ss}	PWR	78	$\overline{MGS0/RAS0}$	O
11	AD31	I/O	45	adV_{cc}	PWR	79	$\overline{MGS1/RAS1}$	O
12	cV_{ss}	PWR[1]	46	AD12	I/O	80	$\overline{MGS2/RAS2}$	O
13	AD30	I/O	47	AD11	I/O	81	$\overline{MGS3/RAS3}$	O
14	cV_{cc}	PWR[1]	48	AD10	I/O	82	$\overline{MFLT}$	I
15	adV_{ss}	PWR	49	AD9	I/O	83	$\overline{IN0}$	I
16	adV_{cc}	PWR	50	adV_{ss}	PWR	84	$\overline{IN1}$	I
17	AD29	I/O	51	adV_{cc}	PWR	85	$\overline{IN2}$	I
18	AD28	I/O	52	AD8	I/O	86	$\overline{IN3}$	I
19	AD27	I/O	53	AD7	I/O	87	$\overline{IN4}$	I
20	AD26	I/O	54	AD6	I/O	88	CLK	I
21	adV_{ss}	PWR	55	adV_{ss}	PWR	89	cV_{ss}	PWR[2]
22	adV_{cc}	PWR	56	adV_{cc}	PWR	90	cV_{cc}	PWR[2]
23	AD25	I/O	57	AD5	I/O	91	$\overline{IN5}$	I
24	AD24	I/O	58	AD4	I/O	92	$\overline{IN6}$	I
25	AD23	I/O	59	AD3	I/O	93	$\overline{IN7}$	I
26	adV_{ss}	PWR	60	AD2	I/O	94	$\overline{CAS0}$	O
27	adV_{cc}	PWR	61	adV_{ss}	PWR	95	$\overline{CAS1}$	O
28	AD22	I/O	62	adV_{cc}	PWR	96	$\overline{CAS2}$	O
29	AD21	I/O	63	cV_{cc}	PWR[1]	97	$\overline{CAS3}$	O
30	AD20	I/O	64	AD1	I/O	98	RAS	O
31	AD19	I/O	65	cV_{ss}	PWR[1]	99	CAS	O
32	adV_{ss}	PWR	66	AD0	I/O	100	ctrlV_{ss}	PWR
33	adV_{cc}	PWR	67	$\overline{EWE}$	O			
34	AD18	I/O	68	$\overline{RAS}$	O			

Notes:

1. PWR pin is near clock driver.
2. PWR pin is near PLL.

I	= Input-Only Pin	I/O	= Bidirectional Pins
O	= Output-Only Pin	PWR	= Power Pins

Table 3. PSC1000 Microprocessor Ordering Information

Description	CPU Clock Frequency (MHz)	Package Type	Stock Number
PSC1000-BAXTC	80	TQFP	31-0100371

ADVANCE INFORMATION

PSC1000 Microprocessor

PATRIOT SCIENTIFIC CORPORATION

32-BIT RISC PROCESSOR

Purpose

This reference manual describes the architecture, hardware interface, and programming of the PSC1000 Microprocessor. The Patriot PSC1000 microprocessor is one of a family of low-power, low-cost, stack-architecture processors targeted specifically for embedded applications. As stack-architecture processors, the PSC1000 family are ideal for applications that must run Java™ at native speeds. These include laser printers, ignition controllers, network routers, personal digital assistants, set-top cable controllers, video games, pagers, cell phones, and many other applications. But since C++ is semantically similar to Java, the PSC1000 family also run C and C++ efficiently, as well as stack-architecture languages such as Forth and Postscript™.

This data book provides the information required to design products that use the PSC1000 CPU, including functional capability, electrical characteristics and ratings, and package definitions, as well as the information required to program both the MPU and VPU.

Overview

The PSC1000 Microprocessor is an implementation of the ShBoom™ Microprocessor architecture. It is a highly integrated 32-bit RISC processor that executes at a peak performance of one instruction per CPU-clock cycle. The CPU is designed specifically for use in those embedded applications for which power consumption, MPU performance, and system cost are deciding selection factors.

The PSC1000 CPU instruction sets are hardwired, allowing most instructions to execute in a single cycle, without the use of pipelines or superscalar architecture. A "flow-through" design allows the next instruction to start before the prior instruction completes, thus increasing performance.

The PSC1000 MPU contains 52 general-purpose registers, including 16 global data registers, an index register, a count register, a 16-deep addressable register/return stack, and an 18-deep operand stack. Both stacks contain an index register in the top element, are cached on chip, and, when required, automatically spill to and refill from external memory. The stacks minimize the data movement typical of register-based architectures, and also minimize memory accesses during procedure calls, parameter passing, and variable assignments. Additionally, the MPU contains a mode/status register, two stack pointers, and 41 locally addressed registers for I/O, control, configuration, and status.

KEY FEATURES

Run Java at Native Speed: The stack architectures of the PSC1000 microprocessor and the Java Virtual Machine are very similar. This results in only a relatively simple byte code translator (20K) being required to produce executable native code from Java byte code, rather than a full Just-in-Time (JIT) compiler (200–400K) as is required for common processor architectures. The result is *much* faster initial execution of Java programs and significantly smaller memory requirements. Additionally, hundreds of kilobytes of memory are saved due to the reduced size of the translator itself.

Multiple Language Support: Most modern languages are implemented on a stack model. The features that allow the PSC1000 to run Java efficiently apply similarly to other languages such as C, C++, Forth and Postscript.

Dual-Processor Architecture: The CPU contains both a high-performance, zero-operand, dual-stack architecture microprocessing unit (MPU), and an virtual peripheral unit (VPU) that executes instructions to transfer data, measure time, test inputs, set outputs, and emulate peripherals such as serial ports and A to D or D to A converters.

Zero-Operand Architecture: Many RISC architectures waste valuable instruction space—often 15 bits or more per instruction—by specifying three possible operands for every instruction. Zero-operand (stack) architectures eliminate these operand bits, thus allowing much shorter instructions—typically one-fourth the size—and thus a higher instruction-execution bandwidth and smaller program size. Stacks also

ADVANCE INFORMATION

minimize register saves and loads within and across procedures, thus allowing shorter instruction sequences and faster-running code.

Fast, Simple Instructions: Instructions are simpler to decode and execute than those of conventional RISC processors, allowing the PSC1000 MPU and VPU to issue *and* complete instructions in a single clock cycle, as often as every CPU-clock cycle.

Four-Instruction Buffer: Using 8-bit opcodes, the CPU obtains up to four instructions from memory each time an instruction fetch or pre-fetch is performed. These instructions can be repeated without rereading them from memory. This maintains high performance when connected directly to DRAM, without the expense of a cache.

Local and Global Registers: Local and global registers minimize the number of accesses to data memory. The local-register stack automatically caches up to sixteen registers, and the operand stack up to eighteen registers. As stacks, any allocated data space efficiently nests and unnests across procedure calls. The sixteen global registers provide storage for shared data.

Posted Write: Decouples the processor from data writes to memory, allowing the processor to continue executing after a write is posted.

Programmable Memory/Bus Interface: Allows the use of lower-cost memory and system components in price-sensitive systems. The interface supports many types of EPROM/SRAM/DRAM/VRAM directly, including fast-page mode on up to four groups of DRAM devices. On-chip support of RAS cycle $\overline{OE}$ and $\overline{WE}$, CAS-before-RAS, and the DSF signal allow use of VRAM without additional external hardware. Programmable bus timing and driver power allow the designer a range of solutions to system design challenges in order to match the time, performance, and budget requirements for each project.

Clock Multiplier: Internally doubles and quadruples the external clock. An on-chip PLL circuit eliminates typical stringent oscillator specifications, thus allowing the use of lower-cost oscillators.

Fully Static Design: A fully static design allows running the clock from DC up to rated speed. Lower clock speeds can be used to drastically cut power consumption.

Hardware Debugging Support: Both breakpoint and single-step capability aid in debugging programs.

Virtual Memory: Supported through the use of external mapping SRAMs and support logic.

Floating-Point Support: Special instructions implement efficient single- and double-precision IEEE floating-point arithmetic.

Direct Memory Access Controller: Supports up to eight prioritized levels at data rates of up to the equivalent of one byte per CPU clock cycle.

Interrupt Controller: Supports up to eight prioritized levels with interrupt responses as fast as eight CPU-clock cycles.

Eight Bit Inputs and Eight Bit Outputs: I/O bits are available for MPU and VPU application use, thus reducing the requirement for external hardware.

ADVANCE INFORMATION

Central Processing Unit

The PSC1000 CPU architectural philosophy is that of simplification and efficiency of use: implement the simplest solution that adequately solves the problem and provides the best utilization of existing resources. In hardware, this typically equates to using fewer transistors, and fewer transistors means a lower-cost, and often lower-power, CPU.

Early RISC processors reduced transistor counts compared to CISC processors, and gained their cost and performance improvements therein. Today, interconnections between transistors dominate the silicon of many CPUs. The PSC1000 MPU architectural philosophy results in, along with fewer transistors, the minimization of interconnections compared to register-based MPUs.

Resources

The PSC1000 CPU contains ten major functional areas: microprocessing unit (MPU), virtual peripheral unit (VPU), global registers, direct memory access controller (DMAC), interrupt controller (INTC), on-chip resources, bit inputs, bit outputs, programmable memory interface (MIF), and clock. In part, the

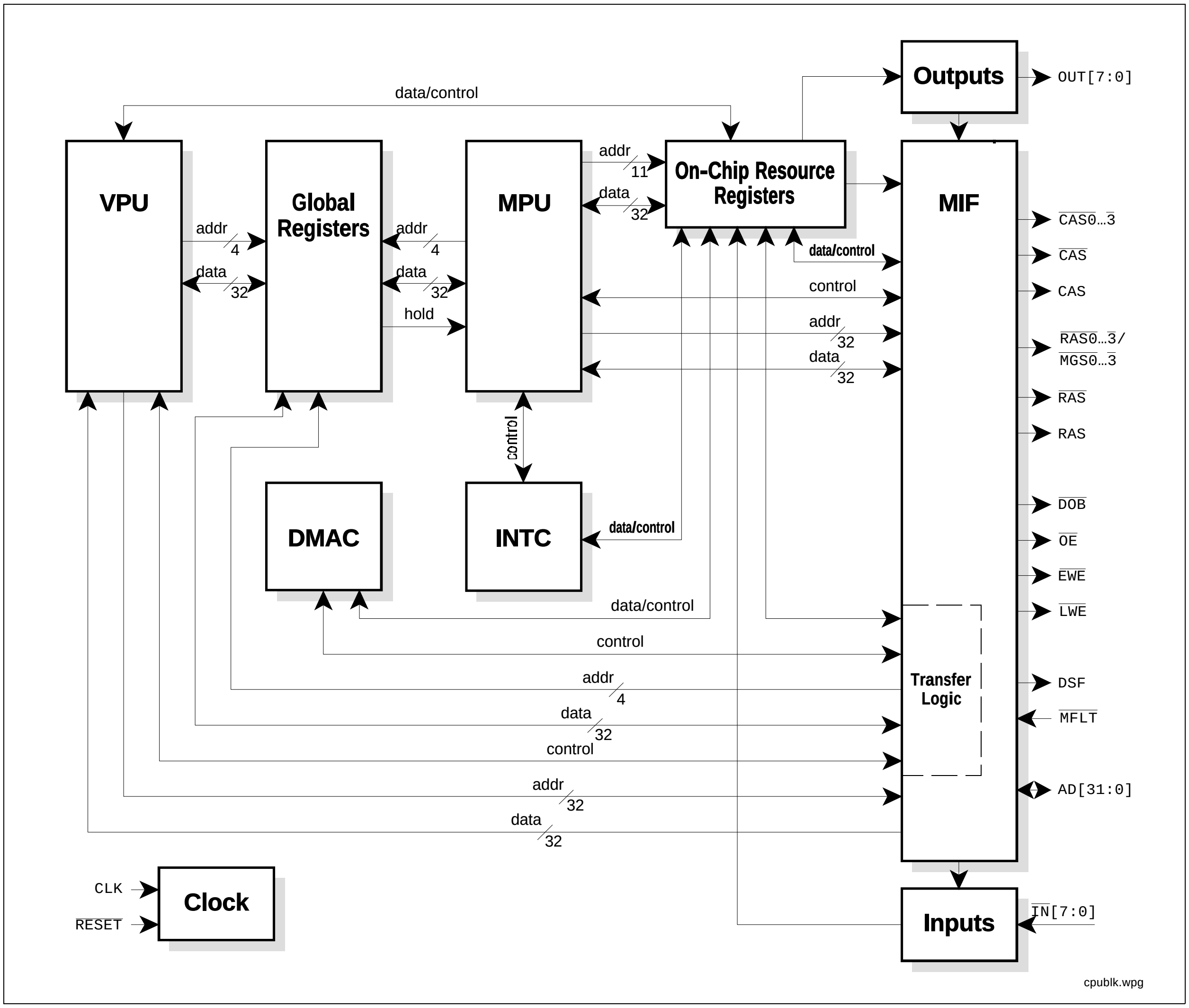


Figure 2. CPU Block Diagram

PSC1000 CPU gains its capability and small silicon size from the resource sharing within and among these areas. See Figure 2. For example:

- The global registers are shared by the MPU, the VPU, and the transfer logic within the MIF. They are used by the MPU for data storage and control communication with the DMAC and the VPU; by the VPU for transfer information, loop counts, and delay counts; and by the DMAC for transfer information. Further, the transfer information is used by the transfer logic in the MIF which is shared by the VPU and DMAC.

- The MIF is shared by the MPU, the VPU, the DMAC, the bit outputs, and the bit inputs for access to the system bus. Bus transaction requests are arbitrated and prioritized by the MIF to ensure temporally deterministic execution of the VPU.

- The bit inputs are made available to the system through the On-Chip Resource Registers. They are shared by the INTC and the DMAC for service requests, are available to the MPU and the VPU for programmed input, and are bit-addressable.

- The DMAC transfer-termination logic is significantly reduced by using specific termination conditions and close coupling with the MPU for intelligent termination action.

- The INTC is shared by the bit inputs, the VPU, and the DMAC (through the MIF transfer logic) for interrupt requests to the MPU.

- The bit outputs are made available to the system through the On-Chip Resource Registers. They are shared by the MPU and the VPU for programmed output, and are bit-addressable.

Although the maximum usage case requiring a complex VPU program, many interrupt sources, many input bits, many output bits, all available DMA channels, and maximum MPU computational ability might leave a shortage of resources, such applications are not typical. The sharing of resources among functional units increases CPU capability and flexibility, and significantly reduces transistor count, package pin count, and thus silicon size and cost. The ability to select among available resources, compared to the fixed resource set of other CPUs, allows the PSC1000 CPU to be used for a wider range of applications.

Clock Speed

The clock speed of a CPU is not a predictor of its performance. For example, the PowerPC 604, running at about half the speed of the DEC Alpha 21064A, achieves about the same SPECint95 benchmark performance. In this respect, the PSC1000 CPU is more like the DEC Alpha than the PowerPC. However, the PSC1000 CPU is based on a significantly different design philosophy than either of these CPUs.

Most processors historically have forced the system designer to maintain a balanced triangle among CPU execution speed, memory bandwidth, and I/O bandwidth. However, as system clock rate increases, typically so does bus speed, cache memory speed, and system interface costs. Typically, too, so do CPU cost, as often thousands of transistors are added to maintain this balance.

The PSC1000 CPU lets the system designer select the performance level desired, while maintaining low system cost. This may tilt the triangle slightly, but cost is not part of the classical triangle-balancing equation. The PSC1000 CPU's programmable memory interface permits a wide range of memory speeds to be used, allowing systems to use slow or fast memory as required. Slow memory clearly degrades system performance, but the fast internal clock speed of the PSC1000 CPU causes internal operations to be completed quickly. Thus the multi-cycle multiply and divide instructions always execute quickly, without the silicon expense of a single-cycle multiply unit. Although higher performance can sometimes be gained by dedicating large numbers of transistors to functions such as these, silicon cost also increases, and increased cost did not fit the design goals for this version of the PSC1000 CPU.

ADVANCE INFORMATION

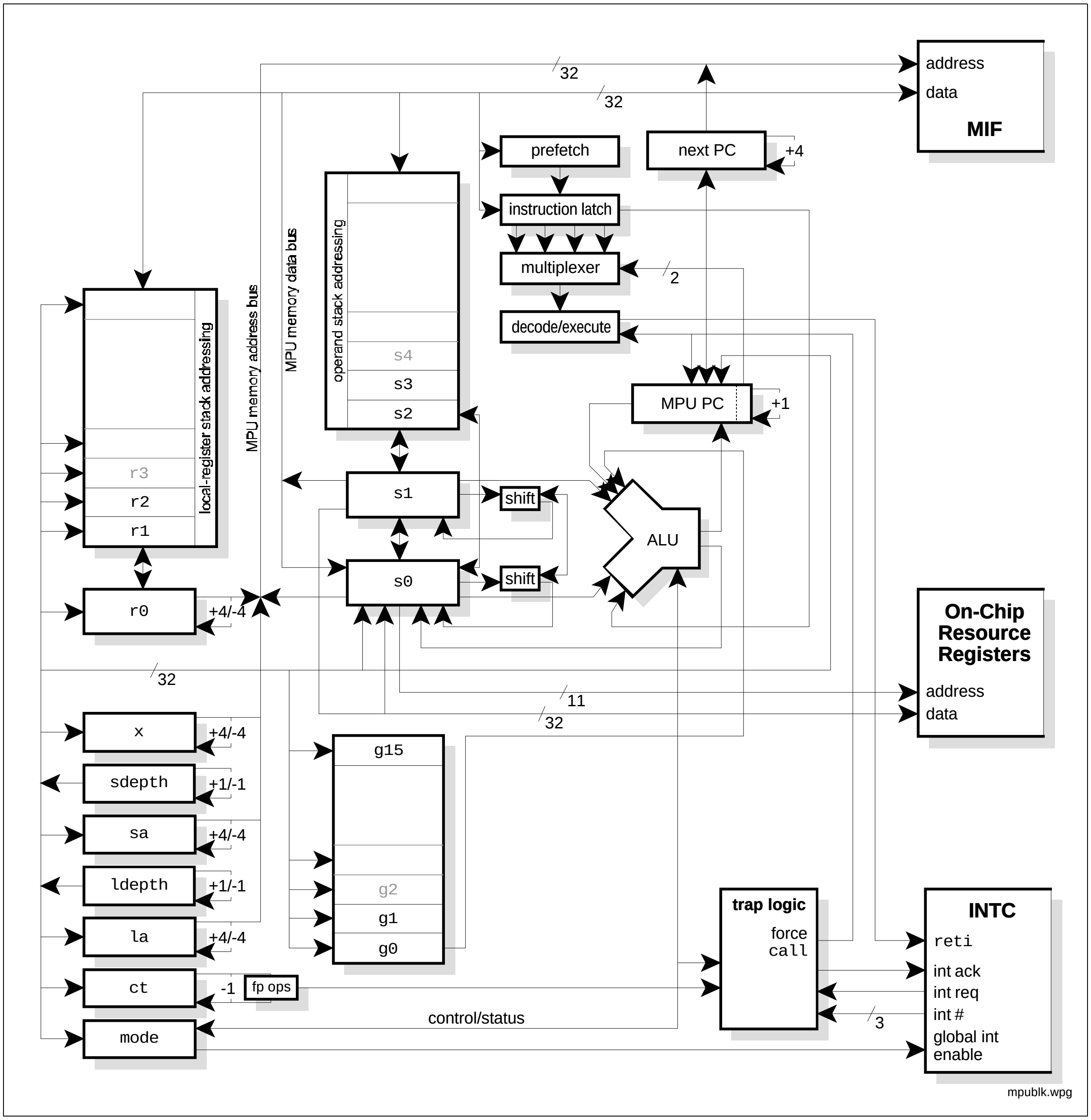


Figure 3. MPU Block Diagram

Microprocessing Unit

The MPU supports the ShBoom™ architectural philosophy of simplification and efficiency of use through its basic design in several interrelated ways.

Whereas most RISC processors use pipelines and superscalar execution to execute at high clock rates, the PSC1000 MPU uses neither. By having a simpler architecture, the PSC1000 MPU issues *and* completes most instructions in a single clock cycle. There are no pipelines to fill and none to flush during changes in program flow. Though more instructions are sometimes required to perform the same procedure in PSC1000 MPU code, the MPU operates at a higher clock frequency than other processors of similar silicon size and technology, thus giving comparable performance at significantly reduced cost.

A microprocessor's performance is often limited by how quickly it can be fed instructions from memory. The MPU reduces this bottleneck by using 8-bit instructions so that up to four instructions (an *instruction group*) can be obtained during each memory access. Each instruction typically takes one CPU-clock cycle to execute, thus requiring four CPU-clock cycles to execute the instruction group. Because a memory access can complete in four (or even fewer) CPU-clock cycles, the next instruction group can be available when execution of the previous group completes. This makes it possible to feed instructions to the processor at maximum instruction-execution bandwidth without the cost and complexity of an instruction cache.

Table 4. Instruction Bandwidth Comparison

g5 = g1 - (g2 + 1) + g3 - (g4 * 2)

Typical RISC MPU		**PSC1000 MPU**	
		push	g1
		push	g2
add	#1,g2,g5	inc	#1
sub	g1,g5,g5	sub	
		push	g3
add	g5,g3,g5	add	
		push	g4
shl	g4,#1,temp	shl	#1
		sub	
sub	g5,temp,g5	pop	g5
20 bytes		10 bytes	

Example of twice the instruction bandwidth available on the PSC1000 MPU

The zero-operand (stack) architecture makes 8-bit instructions possible. The stack architecture eliminates the requirement to specify source and destination operands in every instruction. By not using opcode bits on every instruction for operand specification, a much greater bandwidth of functional operations—up to four times as high—is possible. Table 4 depicts an example PSC1000 MPU instruction sequence that demonstrates twice the

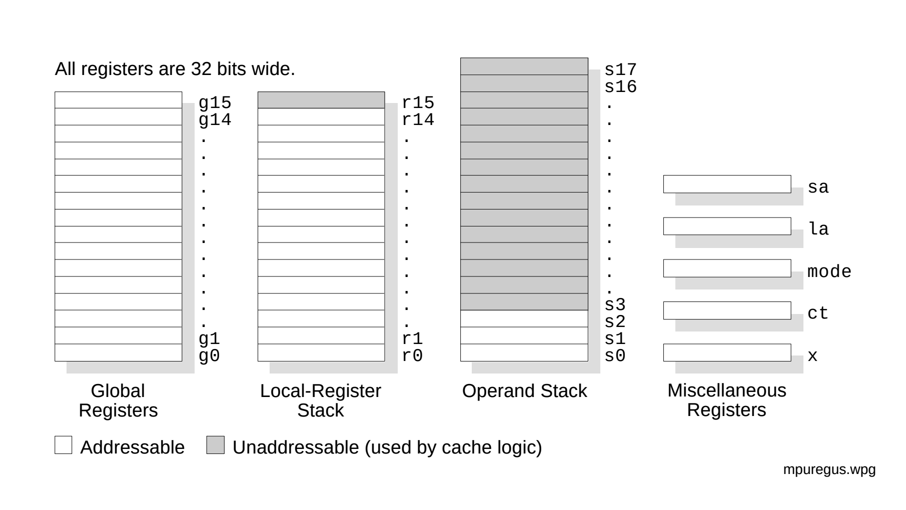


Figure 4. MPU Registers

typical RISC MPU instruction bandwidth. The instruction sequence on the PSC1000 MPU requires one-half the instruction bits, and the uncached performance benefits from the resulting increase in instruction bandwidth.

Stack MPUs are thus simpler than register-based MPUs, and the PSC1000 MPU has two hardware stacks to take advantage of this: the operand stack and the local-register stack. The simplicity is widespread and is reflected in the efficient ways stacks are used during execution.

The ALU processes data from primarily one source of inputs—the top of the operand stack. The ALU is also used for branch address calculations. Data bussing is thus greatly reduced and simplified. Intermediate results typically “stack up” to unlimited depth and are used directly when needed, rather than requiring specific register allocations and management. The stacks are individually cached and spill and refill automatically, eliminating software overhead for stack manipulation typical in other RISC processors. Function parameters are passed on, and consumed directly off of, the operand stack, eliminating the need for most stack frame management. When additional local storage is required, the local-register stack supplies registers that efficiently nest and unnest across functions. As stacks, the stack register spaces are only allocated for data actually stored, maximizing storage utilization and bus bandwidth when registers are spilled or refilled—unlike architectures using fixed-size register windows. Stacks speed context switches, such as interrupt servicing, because registers do not need to be explicitly saved before use—additional stack space is allocated as required. The stacks thus reduce the number of explicitly addressable registers otherwise required, and speed execution by reducing data location specification and movement. Stack storage is inherently local, so the global registers supply non-local register resources when required.

Eight-bit opcodes are too small to contain much associated data. Additional bytes are necessary for immediate values and branch offsets. However, variable-length instructions usually complicate decoding and complicate and lengthen the associated data access paths. To simplify the problem, byte literal data is taken only from the rightmost byte of the instruction group, regardless of the location of the byte literal opcode within the group. Similarly, branch offsets are taken as all bits to the right of the branch opcode, regardless of the opcode position. For 32-bit literal data, the data is taken from a subsequent memory cell. These design choices ensure that the required data is always right-justified for placement on the internal data busses, reducing interconnections and simplifying and speeding execution.

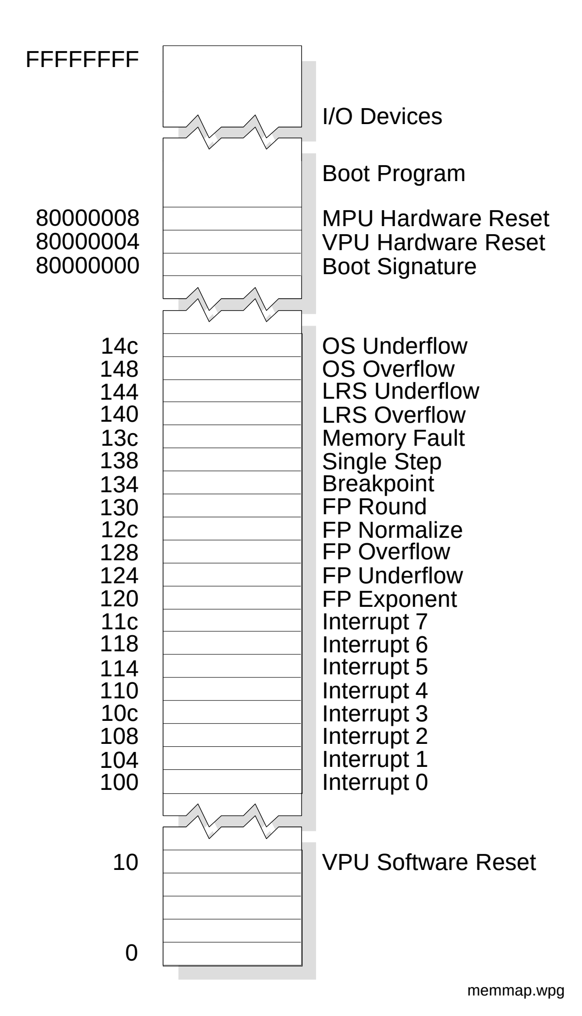


Figure 5. CPU Memory Map

Since most instructions decode and execute in a single clock cycle, the same ALU that is used for data operations is also available, and is used, for branch address calculations. This eliminates an entire ALU often required for branch offset calculations.

Rather than consume the chip area for a single-cycle multiply-accumulate unit, the higher clock speed of the MPU reduces the execution time of conventional multi-cycle multiply and divide instructions. For efficiently multiplying by constants, a fast multiply instruction multiplies only by the specified number of bits.

Rather than consume the chip area for a barrel shifter, the counted bit-shift operation is “smart” to first shift by bytes, and then by bits, to minimize the cycles required. The shift operations can also shift double cells (64 bits), allowing bit-rotate instructions to be easily synthesized.

Although floating-point math is useful, and sometimes required, it is not heavily used in embedded applications. Rather than consume the chip area for a floating-point unit, MPU instructions to efficiently perform the most time-consuming aspects of basic IEEE floating-point math operations, in both single and double precision, are supplied. The operations use the “smart” shifter to reduce the cycles required.

Byte read and write operations are available, but cycling through individual bytes is slow when scanning for byte values. These types of operations are made more efficient by instructions that operate on all of the bytes within a cell at once.

Address Space

The MPU fully supports a linear four-gigabyte address space for all program and data operations. I/O devices are selected by mapping them into memory addresses. By convention, the uppermost address bits select I/O device addresses decoded in external hardware. This convention leaves a contiguous linear program and data space of two gigabytes with a sparse address space above two gigabytes. It also allows simultaneous addressing of an I/O device and a memory address for I/O channel transfers. See *Memory and Device Addressing*, page 105.

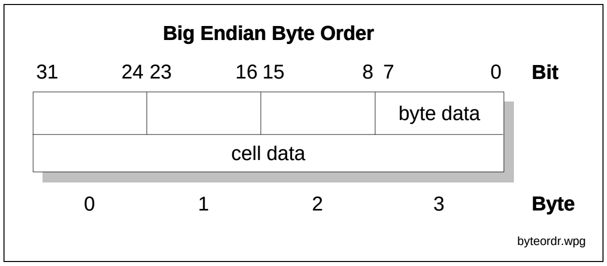


Figure 6. Byte Order

Several instructions or operations expect addresses aligned on four-byte (cell) boundaries. These addresses are referred to as *cell-aligned*. Only the upper 30 bits of the address are used to locate the data; the two least-significant address bits are ignored but appear externally. Within a cell, the high order byte is located at the low byte address. The next lower-order byte is at the next higher address, and so on. For example, the value 0x12345678 would exist at byte addresses in memory, from low to high address, as 12 34 56 78. See Figure 6.

Registers and Stacks

The register set contains 52 general-purpose registers, a mode/status register, two stack pointers, and 41 local address-mapped on-chip resource registers used for I/O, configuration, and status. See Figure 4, and Figure 24, page 129.

The operand stack contains eighteen registers and operates as a push-down stack, with direct access to the top three registers (`s0–s2`). These registers and the remaining registers (`s3–s17`) operate together as a stack cache. Arithmetic, logical, and data-movement operations, as well as intermediate result processing, are performed on the operand stack. Parameters are passed to procedures and results are returned from procedures on the stack, without the requirement of building a stack frame or necessarily moving data between other registers and the frame. As a true stack, registers are allocated only as required, resulting in efficient use of available storage. The external operand stack is addressed by register `sa`.

The local-register stack contains sixteen registers and operates as a push-down stack with direct access to the first fifteen registers (`r0–r14`). Theses registers and

ADVANCE INFORMATION

the remaining register (`r15`) operate together as a stack cache. As a stack, they are used to hold subroutine return addresses and automatically nest local-register data. The external operand stack is addressed by register `la`.

Both cached stacks automatically spill to memory and refill from memory, and can be arbitrarily deep. Additionally, `s0` and `r0` can be used for memory access. See *Stacks and Stack Caches* on page 28.

The use of stack-cached operand and local registers improves performance by eliminating the overhead required to save and restore context (when compared to processors with only global registers available). This allows for very efficient interrupt and subroutine processing.

In addition to the stacks are sixteen global registers and three other registers. The global registers (`g0`–`g15`) are used for data storage, as operand storage for the MPU multiply and divide instructions (`g0`), and for the VPU. Since these registers are shared, the MPU and the VPU can also communicate through them. Remaining are `mode`, which contains mode and status bits; `x`, which is an index register (in addition to `s0` and `r0`); and `ct`, which is a loop counter and also participates in floating-point operations.

Programming Model

For those familiar with the Java Virtual Machine, American National Standard Forth (ANS Forth), Postscript, or Hewlett-Packard calculators that use postfix notation, commonly known as Reverse Polish Notation (RPN), programming the PSC1000 MPU is in many ways be very familiar.

An MPU architecture can be classified as to the number of operands specified within its instruction format. Typical 16-bit and 32-bit CISC and RISC MPUs are usually two- or three-operand architectures, whereas smaller microcontrollers are often one-operand architectures. In each instruction, two- and three-operand architectures specify a source and destination, or two sources and a destination, whereas one-operand architectures specify only one source and have an implicit destination, typically the accumulator. Architectures are also usually not pure. For example, one-operand architectures often have two-operand instructions to specify both a source and destination for data movement between registers.

The PSC1000 MPU is a zero-operand architecture, known as a *stack computer*. Operand sources and destinations are assumed to be on the top of the operand stack, which is also the accumulator. An operation such as `add` uses both source operands from the top of the operand stack, adds them, and returns the result to the top of the operand stack, thus causing a net reduction of one in the operand stack depth. See Figure 7.

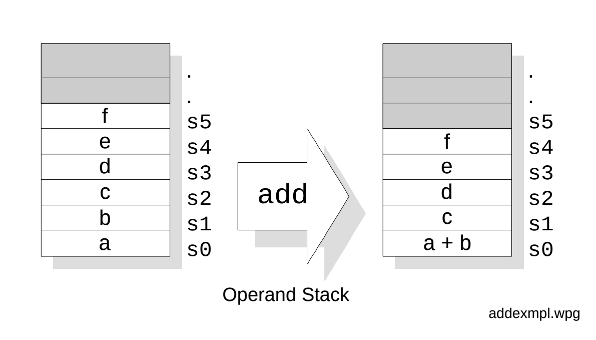


Figure 7. add **Execution Example**

Most ALU operations behave similarly, using two source operands and returning one result operand to the operand stack. A few ALU operations use one source operand and return one result operand to the operand stack. Some ALU and other operations also require a non-stack register, and a very few do not use the operand stack at all.

Non-ALU operations are also similar. Loads (memory reads) either use an address on the operand stack or in a specified register, and place the retrieved data on the operand stack. Stores (memory writes) use either an address on the operand stack or in a register, and use data from the operand stack. Data movement operations push data from a register onto the operand stack, or pop data from the stack into a register.

Once data is on the operand stack it can be used for any instruction that expects data there. The result of an add, for instance, can be left on the stack indefinitely, until used by a subsequent instruction. See Table 4. Instructions are also available to reorder the data in the top few cells of the operand stack so that prior results can be accessed when required. Data can also be removed from the operand stack and placed in local or global registers to minimize or eliminate later reordering of stack elements. Data can even be popped from the operand stack and restacked by pushing it onto the local-register stack.

Computations are usually most efficiently performed by executing the most deeply nested computations first, leaving the intermediate results on the operand stack, and then combining the intermediate results as the computation unnests. If the nesting of the computation is complex, or if the intermediate results are to be used some time later after other data will have been added to the operand stack, the intermediate results can be removed from the operand stack and stored in global or local registers.

Global registers are used directly and maintain their data indefinitely. Local registers are registers within the local-register stack cache and, as a stack, must first be allocated. Allocation can be performed by popping data from the operand stack and pushing it onto the local-register stack one cell at a time. It can also be preformed by allocating a block of uninitialized stack registers at one time; the uninitialized registers are then initialized by popping data, one cell at a time, into the registers in any order. The allocated local registers can be deallocated by pushing data onto the operand stack by popping it off of the local register stack one cell at a time, and then discarding from the operand stack the data that is not required. Alternatively, the allocated local registers can be deallocated by first saving any data required from the registers, and then deallocating a block of registers at one time. The method selected depends on the number of registers required and whether the data on the operand stack is in the required order.

Registers on both stacks are referenced relative to the tops of the stacks and are thus local in scope. What was accessible in r0, for example, after one cell has been push onto the local-register stack, is accessible as r1; the newly pushed value is accessible as r0.

Parameters are passed to and returned from subroutines on the operand stack. An unlimited number of parameters can be passed and returned in this manner. An unlimited number of local-register allocations can also be made. Parameters and allocated local registers thus conveniently nest and unnest across subroutines and program basic blocks.

Subroutine return addresses are pushed onto the local-register stack and thus appear as r0 on entry to the subroutine, with the previous r0 accessible as r1, and so on. As data is pushed onto the stacks and the available register space fills, registers are spilled to memory when required. Similarly, as data is removed from the stacks and the register space empties, the registers are refilled from memory as required. Thus from the program's perspective, the stack registers are always available.

Instruction Set Overview

Table 5 lists the MPU instructions; Table 39, Table 39, page 84, 85, and Table 40, Table 40, page 86, 87, list the mnemonics and opcodes. All instructions consist of eight bits, except for those that require immediate data. This allows up to four instructions (an instruction group) to be obtained on each instruction fetch, thus reducing memory-bandwidth requirements compared to typical RISC machines with 32-bit instructions. This characteristic also allows looping on an instruction group (a micro-loop) without additional instruction fetches from memory, further increasing efficiency. Instruction formats are depicted in Figure 8.

Table 5. MPU Instruction Set

ARITHMETIC/SHIFT

ADD
ADD with carry
ADD ADDRESS
SUBTRACT
SUBTRACT with borrow
INCREMENT
DECREMENT
NEGATE
SIGN EXTEND BYTE
COMPARE
MAXIMUM
MULTIPLY SIGNED
MULTIPLY UNSIGNED
FAST MULTIPLY SIGNED
DIVIDE UNSIGNED
SHIFT LEFT/RIGHT
DOUBLE SHIFT LEFT/RIGHT
INVERT CARRY

MISCELLANEOUS

CACHE CONTROL
FRAME CONTROL
STACK DEPTH
NO OPERATION
ENABLE/DISABLE INTERRUPTS

CONTROL TRANSFER

BRANCH
BRANCH ON ZERO
BRANCH INDIRECT
CALL
CALL INDIRECT
DECREMENT AND BRANCH
SKIP
SKIP ON CONDITION
MICRO-LOOP
MICRO-LOOP ON CONDITION
RETURN
RETURN FROM INTERRUPT

FLOATING POINT

TEST EXPONENT
EXTRACT EXPONENT
EXTRACT SIGNIFICAND
REPLACE EXPONENT
DENORMALIZE
NORMALIZE RIGHT/LEFT
EXPONENT DIFFERENCE
ADD EXPONENTS
SUBTRACT EXPONENTS
ROUND

LOGICAL

AND
OR
XOR
NOT AND
TEST BYTES
EQUAL ZERO

DEBUGGING

STEP
BREAKPOINT

DATA MANAGEMENT

LOAD
STORE
STORE INDIRECT, pre-dec/post-inc
PUSH REGISTER/STACK
POP REGISTER/STACK
EXCHANGE
REVOLVE
SPLIT
REPLACE BYTE
PUSH LITERAL
STORE ON-CHIP RESOURCE
LOAD ON-CHIP RESOURCE

ADVANCE INFORMATION

Table 6. ALU Instructions

add	add pc	adda	addc
and	cmp	dec #1	dec #4
dec ct,#1	divu	eqz	iand
inc #1	inc #4	mulfs	muls
mulu	mxm	neg	notc
or	sexb	shift	shiftd
shl #1	shl #8	shr #1	shr #8
shld #1	shrd #1	sub	subb
testb	xor		

Table 7. Code Examples: Rotate

```
; Rotate single cell left by specified number of bits
; ( n1 #bits -- n2 )

rotate_left::

        push    #0      ; space for bits
        xcg             ; get count
        shiftd
        or              ; combine parts
        ...

; Rotate single cell right by specified number of bits
; ( n1 #bits -- n2 )

rotate_right::

        push    #0      ; space for bits
        rev
        rev

        shl     #1      ; make a negative
        notc            ; sign magnitude
        shr     #1      ; number

        shiftd
        or
        ...
```

ALU Operations

Almost all ALU operations occur on the top of the operand stack in `s0` and, if required, `s1`. A few operations also use `g0`, `ct`, or `pc`.

Only one ALU status bit, `carry`, is maintained and is stored in `mode`. Since there are no other ALU status bits, all other conditional operations are performed by testing `s0` on the fly. `eqz` is used to reverse the zero/non-zero state of `s0`. Most arithmetic operations modify `carry` from the result produced out of bit 31 of `s0`. The instruction `add pc` is available to perform `pc`-relative data references. `adda` is available to perform address arithmetic without changing `carry`. Other operations modify `carry` as part of the result of the operation.

`s0` and `s1` can be used together for double-cell shifts, with `s0` containing the more-significant cell and `s1` the less-significant cell of the 64-bit value. Both single-cell and double-cell shifts transfer a bit between `carry` and bit 31 of `s0`. Code depicting single-cell rotates constructed from the double-cell shift is given in Table 7.

All ALU instruction opcodes are formatted as 8-bit values with no encoded fields.

ADVANCE INFORMATION

Table 8. Branch, Loop and Skip Instructions

`br`	`br []`	`bz`	`call`
`call []`	`dbr`	`mloop`	`mloopc`
`mloopn`	`mloopnc`	`mloopnn`	`mloopnz`
`mloopz`	`ret`	`reti`	`skip`
`skipc`	`skipn`	`skipnc`	`skipnn`
`skipnz`	`skipz`		

Branches, Skips, and Loops

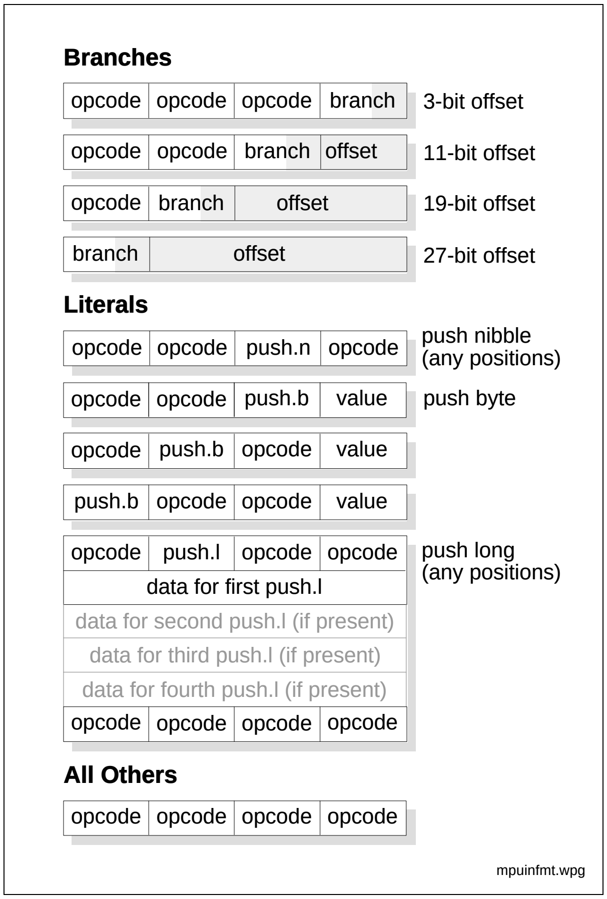
The instructions `br`, `bz`, `call` and `dbr` are variable-length. The three least-significant bits in the opcode and all of the bits in the current instruction group to the right of the opcode are used for the relative branch offset. See Figure 8 and Table 9. Branch destination addresses are cell-aligned to maximize the range of the offset and the number of instructions that are executed at the destination. If an offset is not of sufficient size for the branch to reach the destination, the branch must be moved to an instruction group where more offset bits are available, or a register indirect branch, `br []` or `call []`, can be used. Register indirect branches use an absolute byte-aligned address from `s0`. The instruction `add pc` can be used if a computed `pc`-relative branch is required.

The `mloop_` instructions are referred to as *micro-loops*. If specified, a condition is tested, and then `ct` is decremented. If a termination condition is not met, execution continues at the beginning of the current instruction group. Micro-loops are used to re-execute short instruction sequences without re-fetching the instructions from memory. See Table 14.

Table 9. MPU Branch Ranges

Offset Bits	Offset Range in Bytes
3	-16/+12
11	-4096/+4092
19	-1048576/+1048572
27	-268435456/+268435452

Note:
Encoded offset is in cells. Offset is added to the address of the beginning of the cell containing the branch to compute the destination address.



Branches

opcode | opcode | opcode | branch — 3-bit offset

opcode | opcode | branch | offset — 11-bit offset

opcode | branch | offset — 19-bit offset

branch | offset — 27-bit offset

Literals

opcode | opcode | push.n | opcode — push nibble (any positions)

opcode | opcode | push.b | value — push byte

opcode | push.b | opcode | value

push.b | opcode | opcode | value

opcode | push.l | opcode | opcode — push long (any positions)

data for first push.l

data for second push.l (if present)

data for third push.l (if present)

data for fourth push.l (if present)

opcode | opcode | opcode | opcode

All Others

opcode | opcode | opcode | opcode

Figure 8. MPU Instruction Formats

Other than branching on zero with `bz`, conditional branching is performed with the `skip_` instructions. They terminate execution of the current instruction group and continue execution at the beginning of the next instruction group. They can be combined with the `br`, `call`, `dbr`, and `ret` (or other instructions) to create additional flow-of-control operations.

Table 10. Literal Instructions

`push.b`	`push.l`	`push.n`

Literals

To maximize opcode bandwidth, three sizes of literals are available. The data for four-bit (nibble) literals, with a range of –7 to +8, is encoded in the four least-significant bits of the opcode; the numbers are encoded as two's-complement values with the value 1000 binary decoded as +8. The data for eight-bit(byte) literals, with a range of 0–255, is located in the right-most byte of the instruction group, regardless of the position of the opcode within the instruction group. The data for 32-bit (long, or cell) literals, is located in a cell following the instruction group in the instruction stream. Multiple `push.l` instructions in the same instruction group access consecutive cells immediately following the instruction group. See Figure 8.

Table 11. Data Movement Instructions

`pop ct`	`pop g`*`i`*	`pop r`*`i`*	`pop x`
`push ct`	`push g`*`i`*	`push r`*`i`*	`push s`*`i`*
`push x`			

Data Movement

Register data is moved by first pushing the register onto the operand stack, and then popping it into the destination register. Memory data is moved similarly. See *Loads and Stores*, above.

The opcodes for the data-movement instructions that access `g`*`i`* and `r`*`i`* are 8-bit values with the register number encoded in the four least-significant bits. All other data-movement instruction opcodes are formatted as 8-bit values with no encoded fields.

Table 12. Load and Store Instructions

```
ld [--r0] ld [--x]  ld [r0++] ld [r0]
ld [x++]  ld [x]    ld []     ld.b []
st [--r0] st [--x]  st [r0++] st [r0]
st [x++]  st [x]    st []     replb
```

Loads and Stores

r0 and x support register-indirect addressing and also register-indirect addressing with predecrement by four or postincrement by four. These modes allow for efficient memory reference operations. Code depicting memory move and fill operations is given in Table 14.

Register indirect addressing can also be performed with the address in s0. Other addressing modes can be implemented using adda. Table 13 depicts the code for a complex memory reference operation.

The memory accesses depicted in the examples above are cell-aligned, with the two least-significant bits of the memory addresses ignored. Memory can also be read at byte addresses with ld.b [] and written at byte addresses using x and replb. See *Byte Operations*.

Table 13. Code Example: Complex Addressing Mode

```
; addc [g0+g2+20],#8,[g0-g3-4]

        push    g0
        push    g2
        adda
        push.b  #20
        adda
        ld      []

        push.n  #8
        addc

        push    g0
        push    g3
        neg
        adda
        dec     #4
        st      []

; The carry into and out of addc is maintained.
```

Table 14. Code Examples: Memory Move and Fill

```
; Memory Move
; ( cell_source cell_dest cell_count -- )

move_cells::

        pop     ct          ; count
        pop     x           ; dest
        pop     lstack      ; source to r0

move_cell_loop::

        ld      [r0++]
        st      [x++]
        mloop   move_cell_loop

        push    lstack
        pop                 ; discard source
        ...

; Memory Fill
; ( cell_dest cell_count cell_value  -- )

fill_cells::
        xcg
        pop     ct          ; count
        xcg
        pop     x           ; dest

fill_cells_loop::
        push                ; keep fill value
        st      [x++]
        mloop   fill_cells_loop

        pop                 ; discard fill value
        ...
```

The MPU contains a one-level posted write. This allows the MPU to continue executing while the posted write is in progress and can significantly reduce execution time. Memory coherency is maintained by giving the posted write priority bus access over other MPU bus requests, thus writes are not indefinitely deferred. In the code examples in Table 14, the loop execution overhead is zero when using posted writes. Posted writes are enabled by setting mspwe.

All load and store instruction opcodes are formatted as 8-bit values with no encoded fields.

Table 15. Stack Data Management Instructions

`lframe`	`pop`	`pop lstack`	`push`
`push lstack`	`rev`	`sframe`	`xcg`

Stack Data Management

Operand stack data is used from the top of the stack and is generally consumed when processed. This can require the use of instructions to duplicate, discard, or reorder the stack data. Data can also be moved to the local-register stack to place it temporarily out of the way, or to reverse its stack access order, or to place it in a local register for direct access. See the code examples in Table 14.

If more than a few stack data management instructions are required to access a given operand stack cell, performance usually improves by placing data in a local or global register. However, there is a finite supply of global registers, and local registers, at some point, spill to memory. Data should be maintained on the operand stack only while it is efficient to do so. In general, if the program requires frequent access to data in the operand stack deeper than `s2`, that data, or other more accessible data, should be placed in directly addressable registers to simplify access.

To use the local-register stack, data can be popped from the operand stack and pushed onto the local-register stack, or data can be popped from the local-register stack and pushed onto the operand stack. This mechanism is convenient to move a few cells when the resulting operand stack order is acceptable. When moving more data, or when the data order on the operand stack is not as desired, `lframe` can be used to allocate or deallocate the required local registers, and then the registers can be written and read directly. Using `lframe` also has the advantage of making the required local-register stack space available by spilling the stack as a continuous sequence of bus transactions, which minimizes the number of RAS cycles required when writing to DRAM. The instruction `sframe` behaves similarly to `lframe`, and is primarily used to discard a number of cells from the operand stack.

All stack data management instruction opcodes are formatted as 8-bit values with no encoded fields.

Table 16. Stack Cache Management Instructions

`lcache`	`ldepth`	`pop la`	`pop sa`
`push la`	`push sa`	`scache`	`sdepth`

Stack Cache Management

Other than initialization, and possibly monitoring of overflow and underflow via the related traps, the stack caches do not require active management. Several instructions exist to efficiently manipulate the caches for context switching, status checking, and spill and refill scheduling.

The `_depth` instructions can be used to determine the number of cells in the SRAM part of the stack caches. This value can be used to discard the values currently in the cache, to later restore the cache depth with `_cache`, or to compute the total on-chip and external stack depth.

The `_cache` instructions can be used to ensure either that data is in the cache or that space for data exists in the cache, so that spills and refills occur at preferential times. This allows more control over the caching process and thus a greater degree of determinism during the program execution process. Scheduling stack spills and refills in this way can also improve performance by minimizing the RAS cycles required due to stack memory accesses.

The `_frame` instructions can be used to allocate a block of uninitialized register space at the top of the SRAM part of a stack, or to discard such a block of register space when no longer required. They, like the `_cache` instructions, can be used to group stack spills and refills to improve performance by minimizing the RAS cycles required due to stack memory accesses.

See *Stacks and Stack Caches* on page 28 for more information.

All stack cache management instruction opcodes are formatted as 8-bit values with no encoded fields.

Table 17. Byte Operation Instructions

`ld.b []`	`replb`	`copyb`	`shl #8`
`shr #8`	`testb`		

Byte Operations

Bytes can be addressed and read from memory directly and can be addressed and written to memory with the code depicted in Table 18.

Instructions are available for manipulating bytes within cells. A byte can be replicated across a cell, the bytes within a cell can be tested for zero, and a cell can be shifted by left or right by one byte. Code examples depicting scanning for a specified byte, scanning for a null byte, and moving a null-terminated string in cell-sized units are given below.

All byte operation instruction opcodes are formatted as 8-bit values with no encoded fields.

Table 18. Code Example: Byte Store

```
; Byte store
; ( byte byte_addr -- )

byte_store::

        pop     x       ; address
        ld      [x]     ; get data
        replb           ; insert byte
        st      [x]     ; replace data
```

Table 19. Code Example: Null Character Search

```
; Null character search
; ( cell_source -- )

null_search::

        pop     x       ; address

        push.n  #0
        pop     ct      ; a very long loop

        ; loop terminates when null found or after
        ; a long time if not found.

null_search_loop::

        ld      [x++]
        testb
        pop
        mloopnc         null_search_loop
        ...
```

Table 20. Code Example: Null-Terminated String Move

```
; Move cell-aligned null-terminated string
; ( cell_source cell_dest -- )

null_move::

        pop     x       ; destination
        pop     lstack  ; source

        push.n  #0
        pop     ct      ; a very long loop

null_move_loop::

        ld      [r0++]
        testb           ; check for zero
        st      [x++]
        mloopnc null_move_loop

        push    lstack
        pop             ; discard source
        ...
```

Table 21. Code Example: Byte Search

```
; Byte search
; ( cell_source cell_count byte -- )

byte_search::

        xcg
        pop     ct      ; count

        xcg
        pop     x       ; source

        copyb

byte_search_loop::

        push            ; keep data pattern
        ld      [x++]
        xor

        testb
        pop

        skipnc
        dbr     byte_search_loop
        ; carry set if byte found

        pop             ; discard pattern
        ...
```

Table 22. Floating-Point Math Instructions

addexp	denorm	expdif	extexp
extsig	norml	normr	replexp
rnd	subexp	testexp	

Floating-Point Math

The instructions above are used to implement efficient single- and double-precision IEEE floating-point software for basic math functions (+, -, *, /), and to aid in the development of floating-point library routines. The instructions perform primarily the normalization, denormalization, exponent arithmetic, rounding and detection of exceptional numbers and conditions that are otherwise execution-time-intensive when programmed conventionally. See *Floating-Point Math Support* on page 33.

All floating-point math instruction opcodes are formatted as 8-bit values with no encoded fields.

Table 23. Debugging Instructions

`bkpt`	`step`

Debugging Features

Each of these instructions signals an exception and traps to an application-supplied execution-monitoring program to assist in the debugging of programs. See *Debugging Support* on page 36.

Both debugging instruction opcodes are formatted as 8-bit values with no encoded fields.

Table 24. On-Chip Resources Instructions

`ldo []`	`ldo.i []`	`sto []`	`sto.i []`

On-Chip Resources

These instructions allow access to the on-chip peripherals, status registers, and configuration registers. All registers can be accessed with the `ldo []` and `sto []` instructions. The first six registers each contain eight bits, which are also bit addressable with `ldo.i []` and `sto.i []`. See On-Chip Resource Registers on page 129.

All on-chip resource instruction opcodes are formatted as 8-bit values with no encoded fields.

Table 25. Miscellaneous Instructions

`di`	`ei`	`nop`	`pop mode`
`push mode`	`split`		

Miscellaneous

The disable- and enable-interrupt instructions are the only system control instructions; they are supplied to make interrupt processing more efficient. Other system control functions are performed by setting or clearing bits in `mode`, or in an on-chip resource register. The instruction `split` separates a 32-bit value into two cells, each containing 16 bits of the original value.

All miscellaneous instruction opcodes are formatted as 8-bit values with no encoded fields.

Stacks and Stack Caches

The stack caches optimize use of the stack register resources by minimizing the overhead required for the allocation and saving of registers during programmed or exceptional context switches (such as call subroutine execution and trap or interrupt servicing).

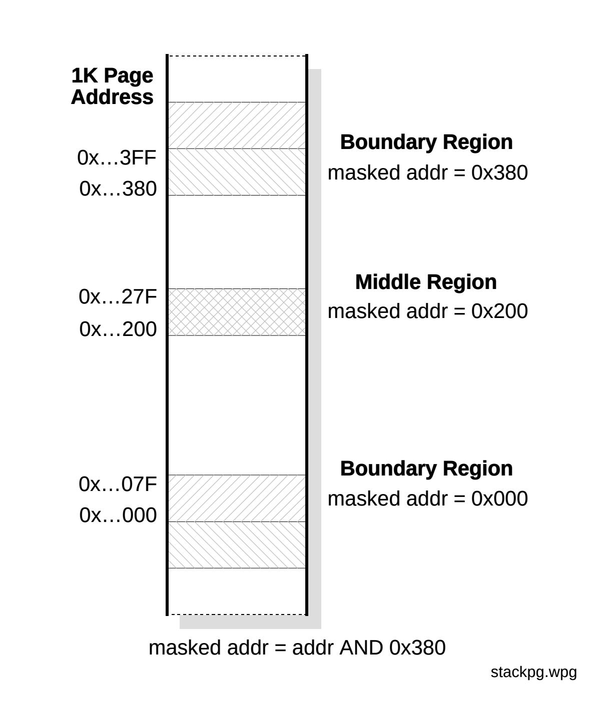


Figure 9. Stack Exception Regions

The local-register stack consists of an on-chip SRAM array that is addressed to behave as a conventional last-in, first-out queue. Local registers `r0–r15` are addressed internally relative to the current top of stack. The registers `r0–r14` are individually addressable and are always contiguously allocated and filled. If a register is accessed that is not in the cache, all the lower-ordinal registers are read in to ensure a contiguous data set.

The operand stack is constructed similarly, with the addition of two registers in front of the SRAM stack cache array to supply inputs to the ALU. These registers are designated `s0` and `s1`, and the SRAM array is designated `s2–s17`. Only registers `s0`, `s1` and `s2` are individually addressable, but otherwise the operand stack behaves similarly to the local-register stack. Whereas the SRAM array, `s2–s17`, can become "empty" (see below), `s0` and `s1` are always considered to contain data.

The stack caches are designed to always allow the current operation to execute to completion before an implicit stack memory operation is required to occur. No instruction explicitly pushes or explicitly pops more than one cell from either stack (except for stack management instructions). Thus to allow execution to completion, the stack cache logic ensures that there is always one or more cells full and one or more cells empty in each stack cache (except immediately after reset, see below) before instruction execution. If, after the execution of an instruction, this is not the case on either stack, the corresponding stack cache is automatically spilled to memory or refilled from memory to reach this condition before the next instruction is allowed to execute. Similarly, the instructions `_cache`, `_frame`, `pop sa`, and `pop la`, which explicitly change the stack cache depth, execute to completion, and then ensure the above conditions exist.

Thus `r15` or `s17` can be filled by the execution of an instruction, but they are spilled before the next instruction executes. Similarly, `r0` and `s2` can be emptied by the execution of an instruction, but they are filled before the next instruction executes.

The stacks can be arbitrarily deep. When a stack spills, data is written at the address in the stack pointer and then the stack pointer is decremented by four (postdecremented stack pointer). Conversely, when a stack refills, the stack pointer is incremented by four, and then data is read from memory (preincremented stack pointer). The stack pointer thus points to the next location to write and the stacks grow from higher to lower memory addresses. The stack pointer for the operand stack is `sa`, and the stack pointer for the local-register stack is `la`.

Since the stacks are dynamically allocated memory areas, some amount of planning or management is required to ensure the memory areas do not overflow or underflow. The simplest is to allocate a sufficiently large memory area so that overflow conditions won't occur. In this case, a correctly written program does not produce underflow. Alternatively, stack memory can be dynamically allocated or monitored through the use of stack-page exceptions.

Stack-Page Exceptions

Stack-page exceptions occur on any stack-cache memory access near the boundary of any 1024-byte memory page to allow overflow and underflow protection and stack memory management. To prevent thrashing stack-page exceptions near the margins of the page boundary areas, once a boundary area is accessed and the corresponding stack-page exception is signaled, the stack pointer must move to the middle region of the stack page before another stack-page exception can be signaled. See Figure 9.

Stack-page exceptions enable stack memory to be managed by allowing stack memory pages to be reallocated or relocated when the edges of the current stack page are approached. The boundary regions of the stack pages are located 32 cells from the ends of each page to allow even a `_cache` or `_frame` instruction to execute to completion and to allow for the corresponding stack cache to be emptied to memory. Using the stack-page exceptions requires that only 2 KB of addressable memory be allotted to each stack at any given time: the current stack page and the page near the most recently encroached boundary.

Each stack supports stack-page overflow and stack-page underflow exceptions. These exception conditions are tested against the memory address that is accessed when the corresponding stack spills or refills between the execution of instructions. `mode` contains bits that signal local-stack overflow, local-stack underflow, operand stack overflow and operand stack underflow, as well as the corresponding trap enable bits.

The stack-page exceptions have the highest priority of all of the traps. As this implies, it is important to consider carefully the stack effects of the stack trap handler code so that stack-page boundaries are not be violated during its execution.

Table 26. Code Example: Stack Initialization

```
init_stacks::

        ; Create a stack area below xx_base in
        ; memory. One cell is read in to initialize s2/r0.

        push.l  #os_base-8
        pop     sa          ; read os_base-4
        ; s0 and s1 are uninitialized

        push.l  #ls_base-8  ; allow dead zone
        pop     la          ; read ls_base-4
```

Stack Initialization

After CPU reset both of the MPU stacks should be considered uninitialized until the corresponding stack pointers are loaded, and this should be one of the first operations performed by the MPU.

After a reset, the stacks are abnormally empty. That is, `r0` and `s2` have not been allocated, and are allocated on the first push operation to, or stack pointer initialization of, the corresponding stack. However, popping the pushed cell causes that stack to be empty and require a refill. The first pushed cell should therefore be left on that stack, or the corresponding stack pointer should be initialized, before the stack is used further. See Table 26.

Stack Depth

The total number of cells on each stack can readily be determined by adding the number of cells that have spilled to memory and the number of cells in the on-chip caches. See Table 27.

Table 27. Code Example: Stack Depth

```
; Operand stack depth

os_depth::

        push.n  #-2
        scache
        pop                 ; ensure 3 spaces available

        .quad   3           ; keep up to push sa
                            ; uninterruptable
        sdepth

        push.l  #os_base-4
        push    sa
        sub                 ; compute memory used

        shr     #1
        shr     #1          ; convert to cells

        add                 ; total on-chip & off
        ...

ls_depth::                  ; "::" forces alignment
                            ; keep to push la
                            ; uninterruptable
        ldepth

        push.l  #ls_base-4
        push    la
        sub                 ; compute memory used

        shr     #1
        shr     #1          ; convert to cells

        add                 ; total on-chip & off
        ...
```

ADVANCE INFORMATION

Table 28. Code Example: Save Context

```
; Context switch: save context
;      Save off any gloabls required and flush stacks

save_context::

      ; Save globals as required
      push      g15
      push      g14
      …                         ; save any others required

; Flush stacks to memory

      ; add one cell to local-register stack so on-chip
      ; part can spill.
      push.b    #-14            ; count for _cache
      pop       lstack

      push      r0              ; count for lcache

      ; ensure no interrupts between flush and la read
      .quad     2
      lcache                    ; write out spillable area
      push      la              ; save pointer

      ; add three cells to stack so on-chip part can spill
      push
      push
      push      r0              ; count for scache

      ; ensure no interrupts between flush and sa read
      .quad     2
      scache                    ; write out all of spillable area
      push      sa

      push.l    #sp_save_area
      st        []              ; save off stack pointer

; Now load new context and continue
      ...
```

Stack Flush and Restore

When performing a context switch, it is necessary to spill the data in the stack caches to memory so that the stack caches can be reloaded for the new context. Attention must be given to ensure that the parts of the stack caches that are always maintained on-chip, `r0` and `s0–s2`, are forced into the spillable area of the stack caches so that they can be written to memory. Code examples are given for context switches that include flushing and restoring the caches in Table 28 and Table 29, respectively.

Table 29. Code Example: Restore Context

```
; Context switch: restore context
;         Restore stack pointer and globals.

restore_context::

         push.l    #sp_save_area
         pop       sa        ; restore it, s2 refills...
                             ; other refill when accessed

         pop
         pop                 ; bring s2 to s0
         pop       la        ; restore it, r0 refills…
                             ; other refill when accessed

         ; Restore globals as required
         ...                 ; restore last saved first
         pop       g14
         pop       g15       ; and first saved last

         ret                 ; return to suspended
                             ; execution
```

Table 30. Traps Dependent on System State

Stack Depth Change		Traps
Operand Stack	Local-Register Stack	
+n	0	Operand Stack Overflow
–n	0	Operand Stack Underflow
0	+1	Local Stack Overflow
0	–1	Local Stack Underflow
+1	-n	Local Stack Underflow Operand Stack Overflow Local Stack Underflow and Operand Stack Overflow
–1	+n	Local Stack Overflow Operand Stack Underflow Local Stack Overflow and Operand Stack Underflow
–1	–n	Local Stack Underflow Operand Stack Underflow Local Stack Underflow and Operand Stack Underflow

Notes:
1. +n > 0, –n < 0
2. If the instruction reads or writes memory or if a posted write is in progress, a memory fault can also occur.
3. If the instruction is single-stepped, a single-step trap also occurs.
4. If any trap occurs, a local-register stack overflow could also occur.

Exceptions and Trapping

Exception handling is precise and is managed by trapping to executable-code vectors in low memory. Each 32-bit vector location can contain up to four instructions. This allows servicing the trap within those four instructions or by branching to a longer trap routine. Traps are prioritized and nested to ensure proper handling. The trap names and executable vector locations are shown in Figure 5.

An exception is said to be signaled when the defined conditions exist to cause the exception. If the trap is enabled, the trap is then processed. Traps are processed by the trap logic, which causes a call subroutine to the associated executable-code-vector address. When multiple traps occur concurrently, the lowest-priority trap is processed first, but before the executable-code vector is executed, the next-higher-priority trap is processed, and so on, until the highest-priority trap is processed. The highest-priority trap's executable-code vector then executes. The nested executable-code-vector return addresses unnest as each trap handler executes `ret`, thus producing the prioritized trap executions.

Interrupts are disabled during trap processing and nesting, until an instruction that begins in byte one of an instruction group is executed. Interrupts do not nest with the traps since their request state is maintained in the INTC registers.

Table 31 lists the priorities of each trap. Traps that can occur explicitly due to the data processed or instruction executed are listed in Table 32. Traps that can occur due to the current state of the system, concurrently with the traps in Table 32, are listed in Table 30.

Table 31. Trap Priorities

Priority	Traps
1 (highest)	local-register stack overflow
2	operand stack overflow
3	local-register stack underflow
4	operand stack underflow
5	memory fault
6	floating-point exponent floating-point underflow floating-point overflow floating-point round
7	floating-point normalize
8	breakpoint
9 (lowest)	single step

Table 32. Traps Independent of System State

Instruction	Trap Combinations
`addexp`	Floating Point Underflow, Floating Point Overflow
`bkpt`	Breakpoint
`denorm`	Floating Point Normalize
`norml`	Floating Point Underflow, Floating Point Normalize, Floating Point Underflow and Floating Point Normalize
`normr`	Floating Point Overflow, Floating Point Normalize, Floating Point Overflow and Floating Point Normalize
`rnd`	Floating Point Round
`step`	Single Step
`subexp`	Floating Point Underflow, Floating Point Overflow
`testexp`	Floating Point Exponent

Floating-Point Math Support

The MPU supports single-precision (32-bit) and double-precision (64-bit) IEEE floating-point math software. Rather than a floating-point unit and the silicon area it would require, the MPU contains instructions to perform most of the time-consuming operations required when programming basic floating-point math operations. Existing integer math operations are used to supply the core add, subtract, multiply, and divide functions, while special instructions are used to efficiently manipulate the exponents and detect exception conditions. Additionally, a three-bit extension to the top one or two stack cells (depending on the precision) is used to aid in rounding and to supply the required precision and exception signaling operations.

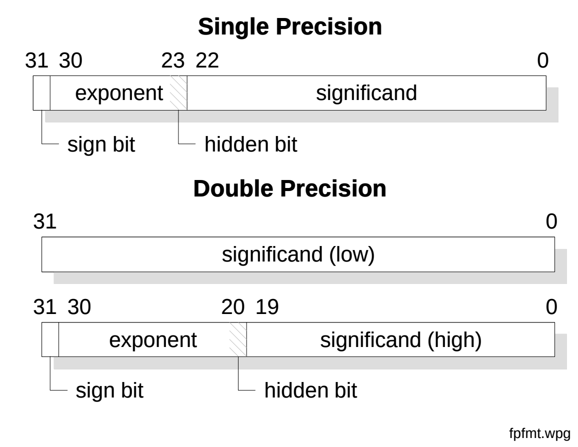


Figure 10. Floating-Point Number Formats

Data Formats

Though single- and double-precision IEEE formats are supported, from the perspective of the MPU, only 32-bit values are manipulated at any one time (except for double shifting). See Figure 10. The MPU instructions directly support the normalized data formats depicted. The related denormalized formats are detected by `testexp` and fully supportable in software.

Status and Control Bits

`mode` contains 13 bits that set floating-point precision, rounding mode, exception signals, and trap enables. See Figure 11, page 39.

Table 33. GRS Extension Bit Manipulation Instructions

```
cleared by:
   testexp  replexp

shifted into by:
   denorm    normr    shift      shiftd
   shr #1    shr #8   shrd #1

shifted out of by:
   norml

tested by:
   rnd

read by:
   push mode

written by:
   pop mode
```

GRS Extension Bits

To maintain the precision required by the IEEE standard, more significand bits are required than are held in the IEEE format numbers. These extra bits are used to hold bits that have been shifted out of the right of the significand. They are used to maintain additional precision, to determine if any precision has been lost during processing, and to determine whether rounding should occur. The three bits appear in `mode` so they can be saved, restored and manipulated. Individually, the bits are named `guard_bit`, `round_bit` and `sticky_bit`. Several instructions manipulate or modify the bits. See Table 33.

When `denorm` and `normr` shift bits into the GRS extension, the source of the bits is always the least-significant bits of the significand. In single-precision mode the GRS extension bits are taken from `s0`, and in double-precision mode the bits are taken from `s1`. For conventional right shifts, the GRS extension bits always come from the least significant bits of the shift (i.e., `s0` if a single shift and `s1` if a double shift). The instruction `norml` is the only instruction to shift bits out of the GRS extension; it shifts into `s0` in single-precision mode and into `s1` in double-precision mode. Conventional left shifts always shift in zeros and do not affect the GRS extension bits.

Table 34. Rounding-Mode Actions

Sign of `ct`	G	R	S	Action
Round to nearest or even				
x	0	x	x	do nothing
x	1	0	0	increment `s0`, clear bit 0 of `s0`
x	1	any 1		increment `s0`
Round toward negative infinity				
0	x	x	x	do nothing
1	0	0	0	do nothing
1	any 1			increment `s0`
Round toward positive infinity				
0	0	0	0	do nothing
0	any 1			increment `s0`
1	x	x	x	do nothing
Round toward zero				
x	x	x	x	do nothing

Rounding

The GRS extension maintains three extra bits of precision while producing a floating-point result. These bits are used to decide how to round the result to fit the destination format. If one views the bits as if they were just to the right of the binary point, then `guard_bit` has a position value of one-half, `round_bit` has a positional value of one-quarter, and `sticky_bit` has a positional value of one-eighth. The rounding operation selected by `fp_round_mode` uses the GRS extension bits and the sign bit of `ct` to determine how rounding occurs. If `guard_bit` is zero the value of GRS extension is below one-half. If `guard_bit` is one the value of GRS extension is one-half or greater. Since the GRS extension bits are not part of the destination format they are discarded when the operation is complete. This information is the basis for the operation of the instruction `rnd`.

Most rounding adjustments by `rnd` involve doing nothing or incrementing `s0`. Whether this is rounding down or rounding up depends on the sign of the floating-point result that is in `ct`. If the GRS extension bits are non-zero, then doing nothing has the effect of "rounding down" if the result is positive, and "rounding up" if the result is negative. Similarly, incrementing the result has the effect of "rounding up" if the result is positive and "rounding down" if the result is negative. If the GRS extension bits are zero then the result was exact and rounding is not required. See Table 34.

In practice, the significand (or the lower cell of a double-precision significand) is in `s0,` and the sign and exponent are in `ct`. `carry` is set if the increment from `rnd` carried out of bit 31 of `s0`; otherwise, `carry` is cleared. This allows `carry` to be propagated into the upper cell of a double-precision significand.

Exceptions

To speed processing, exception conditions detected by the floating-point instructions set exception signaling bits in `mode` and, if enabled, trap. The following traps are supported:

•	Exponent	signaled from `testexp`
•	Underflow	signaled from `norml`, `addexp`, `subexp`
•	Overflow	signaled from `normr`, `addexp`, `subexp`
•	Normalize	signaled from `denorm`, `norml`, `normr`
•	Rounded	signaled from `rnd`

Exceptions are prioritized when the instruction completes and are processed with any other system exceptions or traps that occur concurrently. See *Exceptions and Trapping*, page 32.

- Exponent Trap: Detects special-case exponents. If the tested exponent is all zeros or all ones, `carry` is set and the exception is signaled. Setting `carry` allows testing the result without processing a trap.
- Underflow Trap: Detects exponents that have become too small due to calculations or decrementing while shifting.
- Overflow Trap: Detects exponents that have become too large due to calculations or incrementing while shifting.

Table 35. Code Example: Floating-Point Multiply

```
; Floating-Point Multiply
; ( r1 r2 -- product )
          ...
          testexp
          addexp

          pop       ct          ; save sign & exp sum

; A 24-bit x 24-bit multiply makes a 47 to 48-bit product,
; leaving 16-bits in the high cell. If we multiply 32-bit x
;  24-bit we get a 56-bit product with 24-bits in the high
; part, which is what we want.

; make into a 32-bit multiplier
          shl       #8
          pop       g0

          shl       #1
          push.n    #0

          mulu
          xcg
          pop                   ; discard low part

          normr
          rnd
          normr

          push      ct
          replexp
          ...
```

- Normalize Exception: Detects bits lost due to shifting into the GRS extension. The exception condition is tested at the end of instruction execution and is signaled if any of the bits in the GRS extension are set. Testing at this time allows normal right shifts to be used to set the GRS extension bits for later floating-point instructions to test and signal.
- Rounded Exception: Detects a change in bit zero of `s0` due to rounding.

Hardware Debugging Support

The MPU contains both a breakpoint instruction, `bkpt`, and a single-step instruction, `step`. The instruction `bkpt` executes the breakpoint trap and supplies the address of the `bkpt` opcode to the trap handler. This allows execution at full processor speed up to the breakpoint, and then execution in a program-controlled manner following the breakpoint. `step` executes the instruction at the supplied address, and then executes the single-step trap. The single-step trap can efficiently monitor execution on an instruction-by-instruction basis.

Breakpoint

The instruction `bkpt` performs an operation similar to a call subroutine to address 0x134, except that the return address is the address of the `bkpt` opcode. This behavior is required because, due to the instruction `push.l`, the address of a call subroutine cannot always be determined from its return address.

Commonly, `bkpt` is used to temporarily replace an instruction in an application at a point of interest for debugging. The trap handler for `bkpt` typically restores the original instruction, displays information for the user, and waits for a command. Or, the trap handler could be implemented as a conditional breakpoint to check for a termination condition (such as a register value or the number of executions of this particular breakpoint), continuing execution of the application until the condition is met. The advantage of `bkpt` over `step` is that the applications executes at full speed between breakpoints.

Single-Step

The instruction `step` is used to execute an application program one instruction at a time. It acts much like a return from subroutine, except that after executing one instruction at the return address, a trap to address 0x138 occurs. The return address from the trap is the address of the next instruction. The trap handler for `step` typically displays information for the user, and waits for a command. Or, the trap handler could instead check for a termination condition (such as a register value or the number of executions of this particular location), continuing execution of the application until the condition is met.

`step` is processed and prioritized similarly to the other exception traps. This means that all traps execute before the step trap. The result is that `step` cannot directly single-step through the program code of other trap handlers. The instruction `step` is normally considered to be below the operating-system level, thus operating-system functions such as stack-page traps must execute without its intervention.

Higher-priority trap handlers can be single-stepped by re-prioritizing them in software. Rather than directly executing a higher-priority trap handler from the corresponding executable trap vector, the vector would branch to code to rearrange the return addresses on the return stack to change the resulting execution sequence of the trap handlers. Various housekeeping tasks must also be performed, and the various handlers must ensure that the stack memory area boundaries are not violated by the re-prioritized handlers.

Virtual-Memory Support

The MPU supports virtual memory through the use of external mapping logic that translates logical to physical memory addresses. During MPU RAS memory cycles, the CPU-supplied logical row address is translated by an external SRAM to the physical row address and a memory page-fault bit. The memory page-fault bit is sampled during the memory cycle to determine if the translated page in memory is valid or invalid. Sufficient time exists in the normal RAS precharge portion of DRAM memory cycles to map the logical pages to physical pages with no memory-cycle-time overhead.

An invalid memory page indication causes the memory-fault exception to be signaled and, if enabled, the trap to be executed to service the fault condition. Posted-write faults are completed in the trap routine; other types of faulting operations are completed by returning from the trap routine to re-execute them. Whether the fault is from a read or write operation is indicated by `mflt_write`. The fault address and data (if a write) are stored in `mfltaddr` and `mfltdata`. Memory-fault traps are enabled by `mflt_trap_en`. See the code example on page 37.

Table 36. Code Example: Memory-Fault Service Routine

```
; Memory-fault trap handler

memflt_handler::

          push      mode
          di

          ; Get data (if any) and fault address.

          push.l    #mfltdata   ; must be read first
          ldo       []
          push.l    #mfltaddr   ; must be read last
          ldo       []

; Now go and get the faulted page from disk
;  into memory, update the mapping SRAM, etc.
; ( mode data addr -- mode data addr )
          ...

; If memory fault occurred while attempting a
;  posted write, perform the write in the handler.

          ; check if fault was read or write
          push      s2                    ; duplicate mode
          push.l    #mflt_write
          and

          bz        discard_location      ; write fault?

          push.l    #miscc
          ldo       []

          push.b    #mspwe
          and                             ; posted write?

          .quad     3
          skipz     stack,discard_location
          st        []                    ; complete it
          push                            ; maintain 2 items

discard_location::

          pop                             ; discard “address”
          pop                             ; discard “data”

          ; Reset exception-signal bit.

          push.l    #mflt_exc_sig
          iand
          pop       mode

; For non-posted-write faults, the load/store/pre
;-fetch retries on return.

          ret
```

Table 37. VRAM Commands

Description	At falling edge of:				
	$\overline{\text{RAS}}$				$\overline{\text{CAS}}$
	$\overline{\text{CAS}}$	$\overline{\text{OE}}$	$\overline{\text{WE}}$	DSF	DSF
RAM read/write	H	H	H	L	L
color register set	H	H	H	H	-
masked write	H	H	L	L	L
flash write	H	H	L	H	-
read transfer	H	L	H	L	-
split read transfer	H	L	H	H	-
block write	H	H	H	L	H
masked block write	H	H	L	L	H
set bit-blt mode	L	-	L	-	-

Video RAM Support

Video RAMS (VRAMs) are DRAMs that have a second port that provides serial access to the DRAM array. This allows video data to be serially clocked out of the memory to the display while normal MPU accesses occur. To prevent DRAM array access contentions, the MPU periodically issues *read transfer* requests, which copy the selected DRAM row to the serial transfer buffer. To eliminate read transfer synchronization problems, many VRAMs have split transfer buffers, which allow greater timing flexibility for the MPU's read transfer operations. The MPU instructs the VRAM to perform a read transfer or a split read transfer by encoding the command on the state of the VRAM $\overline{\text{OE}}$, $\overline{\text{WE}}$, and DSF (device special function) during the time $\overline{\text{RAS}}$ falls. These operations are encoded by writing `vram` and performing an appropriate read or write to the desired VRAM memory address. See Figure 32, page 137.

Some VRAMs have more advanced operations—such as line fills, block fills, and bit-blts—which are encoded with other combinations of $\overline{\text{WE}}$, $\overline{\text{OE}}$, DSF, $\overline{\text{RAS}}$, and $\overline{\text{CAS}}$. A basic set of operations and commands is common among manufacturers, but the commands for more advanced functions vary.

Register mode

mode contains a variety of bits that indicate the status and execution options of the MPU. Except as noted, all bits are writable. The register is shown in Figure 11.

mflt_write
After a memory-fault exception is signaled, indicates that the fault occurred due to a memory write.

guard_bit
The most-significant bit of a 3-bit extension below the least-significant bit of s0 (s1, if fp_precision is set) that is used to aid in rounding floating-point numbers.

round_bit
The middle bit of a 3-bit extension below the least-significant bit of s0 (s1, if fp_precision is set) that is used to aid in rounding floating-point numbers.

sticky_bit
The least-significant bit of a 3-bit extension below the least-significant bit of s0 (s1, if fp_precision is set) that is used to aid in rounding floating-point numbers. Once set due to shifting or writing the bit directly, the bit stays set even though zero bits are shifted right through it, until it is explicitly cleared or written to zero.

mflt_trap_en
If set, enables memory-fault traps.

mflt_exc_sig
Set if a memory fault is detected.

ls_boundary
Set if ls_ovf_exc_sig or ls_unf_exc_sig becomes set as the result of a stack spill or refill. Cleared when the address in la, as the result of a stack spill or refill, has entered the middle region of a 1024-byte memory page, and when la is written. Used by the local-register stack trap logic to prevent unnecessary stack overflow and underflow traps when repeated local-register stack spills and refills occur near a 1024-byte memory page boundary. Not writable.

ls_unf_trap_en
If set, enables a local-register stack underflow trap to occur after a local-register stack underflow exception is signaled.

ls_unf_exc_sig
Set if a local-register stack refill occurs, ls_boundary is clear, and the accessed memory address is in the last thirty-two cells of a 1024-byte memory page.

ls_ovf_trap_en
If set, enables a local-register stack overflow trap to occur after a local-register stack overflow exception is signaled.

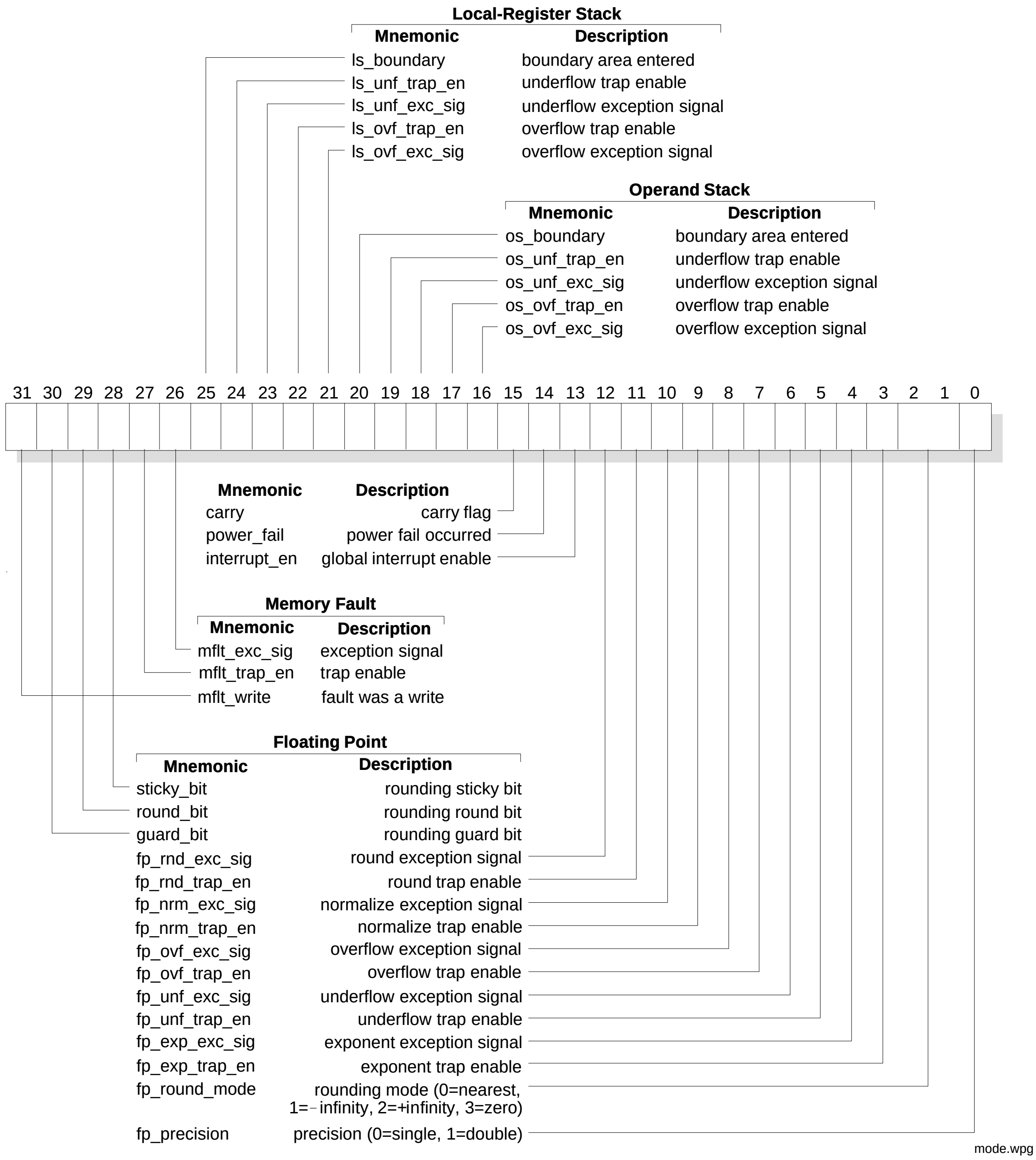
ls_ovf_exc_sig
Set if a local-register stack spill occurs, ls_boundary is clear, and the accessed memory address is in the first thirty-two cells of a 1024-byte memory page.

os_boundary
Set if os_ovf_exc_sig or os_unf_exc_sig becomes set as the result of a stack spill or refill. Cleared when the address in sa, as the result of a stack spill or refill, has entered the middle region of a 1024-byte memory page, and when sa is written. Used by the operand stack trap logic to prevent unnecessary stack overflow and underflow traps when repeated operand stack spills and refills occur near a 1024-byte memory page boundary. Not writable.

os_unf_trap_en
If set, enables an operand stack underflow trap to occur after an operand stack underflow exception is signaled.

os_unf_exc_sig
Set if an operand stack refill occurs, os_boundary is clear, and the accessed memory address is in the last thirty-two cells of a 1024-byte memory page.

os_ovf_trap_en
If set, enables an operand stack overflow trap to occur after an operand stack overflow exception is signaled.



Local-Register Stack

Mnemonic	Description
ls_boundary	boundary area entered
ls_unf_trap_en	underflow trap enable
ls_unf_exc_sig	underflow exception signal
ls_ovf_trap_en	overflow trap enable
ls_ovf_exc_sig	overflow exception signal

Operand Stack

Mnemonic	Description
os_boundary	boundary area entered
os_unf_trap_en	underflow trap enable
os_unf_exc_sig	underflow exception signal
os_ovf_trap_en	overflow trap enable
os_ovf_exc_sig	overflow exception signal

Mnemonic	Description
carry	carry flag
power_fail	power fail occurred
interrupt_en	global interrupt enable

Memory Fault

Mnemonic	Description
mflt_exc_sig	exception signal
mflt_trap_en	trap enable
mflt_write	fault was a write

Floating Point

Mnemonic	Description
sticky_bit	rounding sticky bit
round_bit	rounding round bit
guard_bit	rounding guard bit
fp_rnd_exc_sig	round exception signal
fp_rnd_trap_en	round trap enable
fp_nrm_exc_sig	normalize exception signal
fp_nrm_trap_en	normalize trap enable
fp_ovf_exc_sig	overflow exception signal
fp_ovf_trap_en	overflow trap enable
fp_unf_exc_sig	underflow exception signal
fp_unf_trap_en	underflow trap enable
fp_exp_exc_sig	exponent exception signal
fp_exp_trap_en	exponent trap enable
fp_round_mode	rounding mode (0=nearest, 1=−infinity, 2=+infinity, 3=zero)
fp_precision	precision (0=single, 1=double)

Figure 11. Register `mode`

`os_ovf_exc_sig`
Set if an operand stack spill occurs, `os_boundary` is clear, and the accessed memory address is in the first thirty-two cells of a 1024-byte memory page.

`carry`
Contains the carry bit from the accumulator. Saving and restoring `mode` can be used to save and restore `carry`.

`power_fail`
Set during power-up to indicate that a power failure has occurred. Cleared by any write to `mode`. Otherwise, not writable.

`interrupt_en`
If set, interrupts are globally enabled. Set by the instruction `ei`, cleared by `di`.

`fp_rnd_exc_sig`
If set, a previous execution of `rnd` caused a change in the least significant bit of `s0` (`s1`, if `fp_precision` is set).

`fp_rnd_trap_en`
If set, enables a floating-point round trap to occur after a floating-point round exception is signaled.

`fp_nrm_exc_sig`
If set, one or more of the `guard_bit`, `round_bit` and `sticky_bit` were set after a previous execution of `denorm`, `norml` or `normr`.

`fp_nrm_trap_en`
If set, enables a floating-point normalize trap to occur after a floating-point normalize exception is signaled.

`fp_ovf_exc_sig`
If set, a previous execution of `normr`, `addexp` or `subexp` caused the exponent field to increase to or beyond all ones.

`fp_ovf_trap_en`
If set, enables a floating-point overflow trap to occur after a floating-point overflow exception is signaled.

`fp_unf_exc_sig`
If set, a previous execution of `norml`, `addexp` or `subexp` caused the exponent field to decrease to or beyond all zeros.

`fp_unf_trap_en`
If set, enables a floating-point underflow trap to occur after a floating-point underflow exception is signaled.

`fp_exp_exc_sig`
If set, a previous execution of `testexp` detected an exponent field containing all ones or all zeros.

`fp_exp_trap_en`
If set, enables a floating-point exponent trap to occur after a floating-point exponent exception is signaled.

`fp_round_mode`
Contains the type of rounding to be performed by the MPU instruction `rnd`.

`fp_precision`
If clear, the floating-point instructions operate on stack values in IEEE single-precision (32-bit) format. If set, the floating-point instructions operate on stack values in IEEE double-precision (64-bit) format.

MPU Reset

After reset, the VPU begins executing at address 0x80000004, before the MPU begins execution. The VPU must be programmed to execute `delay` before the MPU can access the bus and begin execution. Once the VPU executes `delay`, the MPU begins executing at address 0x80000008. Details of various startup configurations are detailed in *Processor Startup*, page 181.

Interrupts

The CPU contains an on-chip prioritized interrupt controller that supports up to eight different interrupt levels from twenty-four interrupt sources. Interrupts can be received through the bit inputs, from I/O-channel transfers, from the VPU, or can be forced in software by writing to `ioin`. For complete details of interrupts and their servicing, see *Interrupt Controller*, page 107.

Bit Inputs

The CPU contains eight general-purpose bit inputs that are shared with the INTC and DMAC as requests for those services. The bits are taken from $\overline{\texttt{IN}}$`[7:0]`, or, if so configured, are sampled from `AD[7:0]` on the bus. Sampling from the bus can allow the use of smaller, less-expensive packages for the CPU; it can also reduce PWB area requirements through reuse of the AD bus rather than routing a separate bit-input bus. See *Bit Inputs*, page 111

Bit Outputs

The CPU contains eight general-purpose bit outputs that can be written by the MPU or VPU. The bits are output on $\overline{\texttt{OUT}}$`[7:0]` and are also available on `AD[7:0]` during $\overline{\text{RAS}}$ inactive. Taking the bits from the bus can allow the use of smaller, less-expensive packages for the CPU; it can also reduce PWB area requirements through reuse of the AD bus rather than routing a separate bit-output bus. See *Bit Outputs*, page 115.

Table 38. Instructions That Hold-off Pre-fetch

`bkpt`	`br`	`bz`	`call`	`dbr`	`ld`†
`mloopx`	`push.l`	`ret`	`reti`	`st`†	`step`

† See text.

Instruction Pre-fetch

The MPU issues bus requests ordered to optimize execution. To keep executing instructions as much as possible, the next group of instructions are fetched while the current group executes. This is referred to as *instruction pre-fetch*. Instruction pre-fetch begins as soon as an instruction group begins to execute unless it is held off. Pre-fetch is held off if the executing instruction group contains one of the instruction in Table 38. `ld` and `st` only hold off pre-fetch if they occur as the first instruction in the executing instruction group. Knowing which instruction hold-off prefetch Is useful when programming bus configuration information.

Posted-Write

The MPU supports a one-level posted write. This allows MPU execution to continue unimpeded after the write is posted. To maintain memory coherency, posted writes have the highest priority of all MPU bus requests. This guarantees that memory reads following a posed write will always retrieve the most up-to-date data.

On-Chip Resources

The non-MPU hardware features of the CPU are generally accessed by the MPU through a set of 41 registers located in their own address space. Using a separate address space simplifies implementation, preserves opcodes, and prevents cluttering the normal memory address space with peripherals. Collectively known as the On-Chip Resources, these registers allow access to the bit inputs, bit outputs, INTC, DMAC, MIF, system configuration, and some functions of the VPU. These registers and their functions are referenced throughout this manual and are described in detail in *On-Chip Resource Registers*, page 129.

Instruction Reference

As a stack-based MPU architecture, the PSC1000 MPU instructions have documentation requirements similar to other stack-based systems, such as the Java Virtual Machine (JVM) and American National Standard Forth (ANS Forth). Not surprisingly, many of the JVM and ANS Forth operations are instructions on the PSC1000 MPU. As a result, the JVM and ANS Forth stack notation used for language documentation is useful for describing PSC1000 MPU instructions. The basic notation adapted for the PSC1000 MPU is:

(input_operands -- output_operands)
(L: input_operands -- output_operands)

where “--” indicates the execution of the instruction. “Input_operands” and “output_operands” are lists of values on the operand stack (the default) or local register stack (preceded by “L:”). These are similar, though not always identical, to the source and destination operands that can be represented within instruction mnemonics. The value held in the top-of-stack register (`s0` or `r0`) is always on the right of the operand list with the values held in the higher ordinal registers appearing to the left (e.g., `s2 s1 s0`). The only items in the operand lists are those that are pertinent to the instruction; other values may exist under these on the stacks. All of the input_operands are considered to be popped off the stack, the operation performed, and the output_operands pushed on the stack. For example, a notational expression of:

n1 n2 -- n3

represents two input operands, n1 and n2, and one output operand, n3. For the instruction `add`, n1 (taken from `s1`) is added to n2 (taken from `s0`), and the result is n3 (left in `s0`). If the name of a value on the left of either diagram is the same as the name of a value on the right, then the value was required, but unchanged. The name represents the operand type. Numeric suffixes are added to indicate different or changed operands of the same type. The values may be bytes, integers, floating-point numbers, addresses, or any other type of value that can be placed in a single 32-bit cell.

addr	address
byte	character or byte (upper 24 bits zero)
n	integer or 32 arbitrary bits
other text	integer or 32 arbitrary bits

ANS Forth defines other operand types and operands that occupy more than one stack cell; those are not used here.

Note that typically all stack action is described by the notation and is not explicitly described in the text. If there are multiple possible outcomes then the outcome options are on separate lines and are to be considered as individual cases. If other registers or memory variables are modified, then that effect is documented in the text.

Also on the stack diagram line is an indication of the effect on `carry`, if any, as well as the opcode and execution time at the right margin.

A timing with an “M” indicates the specified number of bus requests and bus transactions (memory cycles) for the instruction to complete. Bus requests require two CPU-clock cycles and bus transaction times are as programmed and described in *Programmable Memory Interface*, page 117, and *Bus Operation*, page 157.The value used for “M” includes both the bus request and bus transaction times.

Timings do not include implied memory cycles such as stack spills and refills required to maintain the state of the stack caches. Any operation that pushes or pops a stack, or references a local register could cause a memory cycle. Operations that wait on the completion of instruction pre-fetch are labeled “Mprefetch.” These are distinct in that pre-fetch occurs in parallel with execution so the wait time is probably not a full memory cycle.

ANS Forth Word Equivalents

Those PSC1000 instructions that are exact equivalents of ANS Forth words are indicated in the body text for the instruction. Many additional ANS Forth words simply require a short instruction sequence, but these are not indicated.

Java Byte Code Equivalents

Those PSC1000 instructions that are exact equivalents of Java byte codes are indicated in the body text for the PSC1000 instruction. Many additional Java byte codes simply require a short instruction sequence, though the most complex byte codes require a subroutine call. For detailed information contact Patriot Scientific.

add

```
add               ( n1 n2 -- n3 )                      carry±      1100 0000
                                                                        0xC0
                                                                 1 CPU-clock
```

Add *n1* and *n2* giving the sum *n3*. `carry` is set if there is a carry out of bit 31 of the sum and cleared otherwise.

Equivalent to Java byte code `iadd`.

Equivalent to ANS Forth word `+`.

```
add pc            ( n1 -- n2 )                         carry±      1011 1011
                                                                        0xBB
                                                                 1 CPU-clock
```

Add the value of `pc` (the byte-aligned address of the `add  pc` opcode) to *n1* giving the sum *n2*. `carry` is set if there is a carry out of bit 31 of the sum and cleared otherwise.

adda

Add Address

```
adda              ( n1 n2 -- n3 )                                  1110 1000
                                                                        0xE8
                                                                 1 CPU-clock
```

Add *n1* and *n2* giving the sum *n3*. `carry` is unaffected.

addc

Add with Carry

```
addc              ( n1 n2 -- n3 )                      carry±      1100 0010
                                                                        0xC2
                                                                 1 CPU-clock
```

Add *n1* and *n2* and `carry` giving the sum *n3*. `carry` is set if there is a carry out of bit 31 of the sum, otherwise `carry` is cleared.

ADVANCE INFORMATION

addexp

Add Exponents

```
addexp           ( n1 n2 -- n3 n4 n5 )                           1101 0010
                                                                      0xD2
                                                              2 CPU-clocks
                 ( L: -- addr )  only when trap processed   4+M CPU-clocks
```

Perform the following:

Exponent_Field(*n5*) = Exponent_Field(*n1*) - BIAS + Exponent_Field(*n2*)
Sign_Bit(*n5*) = Sign_Bit(*n1*) XOR Sign_Bit(*n2*)

BIAS is 127 (0x3F800000 in position) for single precision and 1023 (0x3FF00000 in position) for double precision, as selected by `fp_precision`.

Compute as described above. Clear the exponent field bits and sign bit and set the hidden bit of *n1* and *n2*, giving *n3* and *n4*, respectively. *n5* is the result of the computation. After completion, if the exponent-field calculation result equaled or exceeded the maximum value of the exponent field (exponent field result ≥ 255 for single, exponent field result ≥ 2047 for double) an overflow exception is signaled. If the exponent-field calculation result is less than or equal to zero an underflow exception is signaled. When an exception is signaled, the exponent field of *n5* contains as many bits of the computed exponent as it will hold.

and

Bitwise AND

```
and              ( n1 n2 -- n3 )                     carry clear  1110 0001
                                                                      0xE1
                                                               1 CPU-clock
```

Perform a bitwise AND of *n1* and *n2* giving the result *n3*.

Equivalent to Java byte code `iand`.

Equivalent to the ANS Forth word AND.

bkpt

Breakpoint

```
bkpt                ( -- )                         0011 1100
                    ( L: -- addr )                      0x3C
                                               1+M CPU-clocks
```

Perform a call subroutine to the breakpoint trap location, 0x134. *addr* is the address of the `bkpt` instruction. Typically the breakpoint service routine replaces the `bkpt` opcode at *addr* with the original opcode, performs whatever debugging function desired, and `ret` to *addr*.

Equivalent to Java byte code `breakpoint`.

_b_

Branch if Condition

br *offset* (--) 0000 0xxx
Branch Unconditionally 0x0?
M CPU-clocks

Transfer execution to *offset* cells from the beginning of the current instruction group.

The instruction adds the two's-complement cell offset encoded within and following the br opcode to pc, and transfers execution to the resulting cell-aligned address.

Equivalent to Java byte codes goto, goto_w.

Equivalent to the run-time for the ANS Forth words AGAIN, AHEAD, ELSE.

br [] (*addr* --) 0100 1011
Branch Indirect 0x4B
M CPU-clocks

Replace the value in pc with *addr* to transfer execution to *addr*. Note that *addr* is an absolute byte-aligned address and not an offset.

bz *offset* (*n* --) 0001 0xxx
Branch if Zero 0x1?
M CPU-clocks

If *n* is zero, transfer execution to *offset* cells from the beginning of the instruction group; otherwise, continue execution at the next instruction group.

If *n* is zero the instruction adds the two's-complement cell offset encoded within and following the bz opcode to pc, and transfers execution to the resulting cell-aligned address. If *n* is non-zero execution continues with the next instruction group.

Equivalent to Java byte codes ifeq, ifnull.

Equivalent to the run-time for the ANS Forth words IF, UNTIL, WHILE.

```
dbr offset       ( -- )                                0001 1xxx
Decrement CT and Branch                                     0x1?
                                                    M CPU-clocks
```

Decrement `ct` by one. If `ct,` is non-zero transfer execution to *offset* cells from the beginning of the current instruction group; otherwise, continue execution with the next instruction group.

The instruction decrements `ct` by one. If the resulting `ct` is non-zero the instruction then adds the two's-complement cell offset encoded within and following the `dbr` opcode to `pc`, and transfers execution to the resulting cell-aligned address. If the resulting `ct` is zero execution continues with the next instruction group.

_cache

Fill/Empty Stack Cache

The cache instructions are used to optimize program execution, or to make program execution more deterministic. Stack cache spills and refills can be caused to occur at preferential times, and to occur in bursts to optimize memory access. Executing the instruction with both *n* and *n*−14 (*n*>0) ensures that an exact number of items are in the stack cache. Pushing dummy values onto the stack (one value for the local-register stack, three values for the operand stack) and then executing the instruction with *n* = −14 causes all previously held data to be spilled to memory.

lcache (*n* --) 0100 1101

0x4D

1 or (1M to 14M) CPU-clocks

If *n* > 0, ensure that at least *n* cells can be removed from the local-register stack without causing local-register stack cache refills. Cells are refilled from memory into the cache if required. (1 ≤ *n* ≤ 14).

If *n* < 0 (two's complement), ensure that at least |*n*| cells can be added to the local-register stack without causing local-register stack cache spills. Cells are spilled from the stack cache to memory if required. (−14 ≤ *n* ≤ −1).

If n = 0 the local-register stack cache is unchanged.

scache (*n* -- *n*) 0100 0101

0x45

1 or (1M to 14M) CPU-clocks

If *n* > 0, ensure that at least *n* cells can be removed from the operand stack without causing operand stack cache refills. Cells are refilled from memory into the cache if required. (1 ≤ n ≤ 14).

If *n* < 0 (two's complement), ensure that at least |*n*| cells can be added to the operand stack without causing operand stack cache spills. Cells are spilled from the stack cache to memory if required. (−14 ≤ n ≤ −1)

If *n* = 0 the operand stack cache is unchanged.

ADVANCE INFORMATION

call

Call Subroutine

```
call offset        ( -- )                                    0000 1xxx
                   ( L: -- addr )                                 0x0?
Call Subroutine                                       1+M CPU-clocks
```

Transfer execution to *offset* cells from the beginning of the current instruction group. *addr* is the cell-aligned address of the next instruction group.

The instruction pushes *addr* on the local-register stack and then adds the two's-complement cell *offset* encoded within and following the `call` opcode to `pc,` and transfers execution to the resulting cell-aligned address. The *offset* is in the same form and follows the same rules as those for branches.

```
call []            ( addr1 -- )                              0100 1110
                   ( L: -- addr2 )                                0x4E
Call Subroutine Indirect                              1+M CPU-clocks
```

Replace the value in pc with *addr1* to transfer execution there. *addr2* is the byte-aligned address of the next instruction following `call []`. Note that *addr1* is an absolute address and not an offset.

cmp

Compare

```
cmp                ( n1 n2 -- n1 n2 )          carry±        1100 1011
                                                                  0xCB
                                                           1 CPU-clock
```

Compare *n2* and *n1* as signed values. Set `carry` if *n1* < *n2*, otherwise clear `carry`.

copyb

Copy Byte Across Cell

```
copyb              ( n1 -- n2 )                              1101 0000
                                                                  0xD0
                                                           1 CPU-clock
```

n2 is the result of copying the lowest byte of *n1* into each of the higher byte positions. For example, 0x12345678 becomes 0x78787878.

ADVANCE INFORMATION

dbr

See _b_.

dec

Decrement

```
dec #1          ( n1 -- n2 )                1100 1111
                                                 0xCF
                                          1 CPU-clock
```

Subtract one from *n1* leaving the result *n2*.

Equivalent to ANS Forth word `1-`.

```
dec #4          ( n1 -- n2 )                1100 1101
                                                 0xCD
                                          1 CPU-clock
```

Subtract four from *n1* leaving the result *n2*.

```
dec ct, #1      ( -- )                      1100 0001
                                                 0xC1
                                          1 CPU-clock
```

Subtract one from `ct`.

denorm

Denormalize

```
denorm          ( n1 -- n2 )  if single precision                 1100 0101
                ( n1 n2 -- n3 n4 )  if double precision                0xC5
                                                         1 to 13 CPU-clocks
                ( L: -- addr )  only when trap processed
                                                     3+M to 15+M CPU-clocks
```

Shift *n1* (or *n2n1* if double) right by the bit count in the exponent field of `ct`. Bits shift out of the right into the GRS extension. If any bit in the GRS extension is set, a normalize exception is signaled. The location of the exponent field depends on `fp_precision`. The exponent field of `ct` is decremented to zero.

Shifting is performed by bytes or bits to minimize CPU-clock cycles required. If the count in the exponent bits of `ct` is larger than the width in bits of the significand field + 3 (for the `guard_bit`, `round_bit` and the hidden bit), the `sticky_bit` is set and the other bits are cleared, and execution requires one CPU-clock cycle.

_depth

Depth of Stack

```
ldepth          ( -- n )                                          1001 1011
                                                                       0x9B
                                                                1 CPU-clock
```

n is exactly the number of cells that can be removed from the local-register stack without causing a local-register stack cache refill. (0 ≤ *n* ≤ 14).

```
sdepth          ( -- n )                                          1001 1111
                                                                       0x9F
                                                                1 CPU-clock
```

n is exactly the number of cells, before *n* was pushed, that could be removed from the operand stack without causing an operand stack cache refill. (0 ≤ *n* ≤ 14). If *n* = 14, then an operand stack cache spill occurred when *n* was pushed and only 13 cells remain, excluding *n*, that can be removed from the operand stack without causing an operand stack cache refill.

di

Disable Interrupts

```
di                ( -- )                                   1011 0111
                                                                0xB7
                                                         1 CPU-clock
```

Globally disable interrupts, clearing `interrupt_en`. The `ioie` bits are not changed.

divu

Divide Unsigned

```
divu              ( n1 n2 -- n3 n4 )                       1101 1110
                                                                0xDE
                                                       32 CPU-clocks
```

Divide the double value *n2n1* by the value in `g0` giving the quotient *n3* and remainder *n4*. All values are unsigned. If *n2* is greater than or equal to `g0` then the quotient will overflow. If `g0` is zero then *n3* equals *n1* and *n4* equals *n2*.

ei

Enable Interrupts

```
ei                ( -- )                                   1011 0110
                                                                0xB6
                                                         1 CPU-clock
```

Globally enable interrupts, setting `interrupt_en`. The `ioie` bits are not changed.

ADVANCE INFORMATION

eqz

Equal Zero

`eqz` (*n1* -- *n2*) 1110 0101
0xE5
1 CPU-clock

n2 is the logical inverse of *n1*. If *n1* is equal to zero *n2* is -1. If *n1* is non-zero *n2* is zero.

Equivalent to ANS Forth word `0=`.

expdif

Exponent Difference

`expdif` (*n1 n2* -- *n3 n4*) 1100 0100
0xC4
1 CPU-clock

Clear the upper half of `ct`. Subtract the exponent field of *n2* from the exponent field in *n1* placing the result in the exponent-field bits of `ct`. Clear the exponent-field bits and sign bit and set the hidden bit of *n1* and *n2* giving *n3* and *n4*, respectively. The locations of the exponent field and hidden bit depend on `fp_precision`.

extexp

Extract Exponent

`extexp` (*n1* -- *n2*) 1101 1011
0xDB
1 CPU-clock

Clear the significand bits of *n1* leaving the exponent-field bits and sign bit unchanged, giving *n2*. The locations of the exponent field and significand field depend on `fp_precision`.

ADVANCE INFORMATION

extsig

Extract Significand

```
extsig            ( n1 -- n2 )                              1101 1100
                                                                 0xDC
                                                          1 CPU-clock
```

Clear the exponent and sign bits of *n1* leaving the significand-field bits unchanged. Then set the hidden bit of *n1*, giving *n2*. The locations of the exponent field and significand field depend on `fp_precision`.

`_frame`

Allocate On-Chip Stack Frame

`lframe` (n --) 1011 1110

(L: $\ldots j_2\ j_1$ -- $\ldots j_2\ j_1\ x_n \ldots x_1$) ($n > 0$) 0xBE

1 or (1M to 15M) CPU-clocks

(L: $\ldots j_{n+1}\ j_n \ldots j_1$ -- $\ldots j_{n+2}\ j_{n+1}$) ($n < 0$)

1 or (1 to 15) CPU-clocks

(L: --) ($n = 0$) 1 CPU-clock

If $n > 0$, allocate n uninitialized cells, $x_n \ldots x_1$, at the top of the local-register stack cache. This causes `r0` to move to `r`n, `r1` to move to `r`(n+1), `r`i to move to `r`(n+i), etc. Those local registers for which (n+i) > 14 are written from the local-register stack cache to memory. ($1 \le n \le 15$).

If $n < 0$, discard $|n|$ cells, $j_n \ldots j_1$, from the top of the local-register stack cache. This causes `r0` through `r`($|n|$-1) to be discarded, `r`$|n|$ to become `r0`, `r`($|n|$+1) to become `r1`, etc. ($-15 \le n \le -1$). Each cell discarded that is not in the stack cache requires one CPU-clock cycle.

If $n = 0$, no cells are allocated or discarded.

`sframe` 1011 1111

0xBF

($\ldots j_2\ j_1\ m\ n$ -- $\ldots j_2\ j_1\ x_n \ldots x_1\ m\ n$) ($n > 0$)

1 or (1M to 15M) CPU-clocks

($\ldots j_{n+1}\ j_n \ldots j_1\ m\ n$ -- $\ldots j_{n+2}\ j_{n+1}\ m\ n$) ($n < 0$)

1 or (1 to 15) CPU-clocks

(n -- n) ($n = 0$) 1 CPU-clock

If $n > 0$, allocate n uninitialized cells, $x_n \ldots x_1$, in the operand stack cache after `s0` and `s1`. This causes `s2` to move to `s`(n+2), `s3` to move to `s`(n+3), `s`i to move to `s`(n+i), etc. Those stack cells for which (n+i) > 16 are written from the operand stack cache to memory. ($1 \le n \le 15$).

If $n < 0$, discard $|n|$ cells, $j_n \ldots j_1$, from within the operand stack cache after `s0` and `s1`. This causes `s2` through `s`($|n|$+1) to be discarded, `s`($|n|$+2) to become `s2`, `s`($|n|$+3) to become `s3`, etc. ($-15 \le n \le -1$). Each cell discarded that is not in the stack cache requires one CPU-clock cycle.

If $n = 0$, no cells are allocated or discarded.

ADVANCE INFORMATION

iand

Bitwise Invert then AND

iand	(*n1 n2 -- n3*)	clear carry	1110 1001 0xE9 1 CPU-clock

Clear the bits in *n1* that are set in *n2* leaving the result *n3*.

inc

Increment

inc #1	(*n1 -- n2*)		1100 1110 0xCE 1 CPU-clock

Add one to *n1* giving the sum *n2*.

Equivalent to ANS Forth word 1+.

inc #4	(*n1 -- n2*)		1100 1100 0xCC 1 CPU-clock

Add four to *n1* giving the sum *n2*.

lcache

See _cache.

ld

Load Indirect from Memory

ld [--r0] (-- *n*) 0100 0100
0x44
1+M CPU-clocks

Decrement the address in r0 by four. *n* is the value from the cell in memory at the new address in r0. The two least significant bits of the address are ignored and treated as zero.

ld [--x] (-- *n*) 0100 1010
0x4A
1+M CPU-clocks

Decrement the address in x by four. *n* is the value from the cell in memory at the new address in x. The two least significant bits of the address are ignored and treated as zero.

ld [r0++] (-- *n*) 0100 0110
0x46
M CPU-clocks

n is the value from the cell in memory at the address in r0. Increment r0 by four. The two least significant bits of the address are ignored and treated as zero.

ld [r0] (-- *n*) 0100 0010
0x42
M CPU-clocks

n is the value from the cell in memory at the address in r0. The two least significant bits of the address are ignored and treated as zero.

ld [x++] (-- *n*) 0100 1001
0x49
M CPU-clocks

n is the value from the cell in memory at the address in x. Increment x by four. The two least significant bits of the address are ignored and treated as zero.

ld [x] (-- *n*) 0100 0001
0x41
M CPU-clocks

n is the value from the cell in memory at the address in x. The two least significant bits of the address are ignored and treated as zero.

ADVANCE INFORMATION

ld [] (*addr* -- *n*) 0100 0000
0x40
M CPU-clocks

n is the value from the cell in memory at the address *addr*. The two least significant bits of the address are ignored and treated as zero.

Equivalent to ANS Forth words @, F@, SF@.

ld.b [] (*addr* -- *byte*) 0100 1000
0x48
M CPU-clocks

byte is the value from the byte in memory at the address *addr*.

Equivalent to ANS Forth word C@.

ldo

Load Indirect from On-Chip Resource

ldo [] (*addr* -- *n*) 1001 0110
0x96
1 CPU-clock

n is the value from the on-chip resource at *addr*. For valid values of *addr*, see *On-Chip Resource Registers*, page 129.

ldo.i [] (*bit_addr* -- *n*) 1001 0111
0x97
1 CPU-clock

n is all ones (-1) if the bit at the on-chip resource address *bit_addr* is one, otherwise *n* is zero. For valid values of *bit_addr*, see *On-Chip Resource Registers*, page 129.

ldepth

See _depth.

lframe

See _frame.

mloop_

Micro Loop on Condition

An mloop re-executes the current instruction group, beginning with the first instruction in the group, up to the mloop_ instruction, until a specified condition is not met or until ct is decremented to zero. When either termination condition occurs, execution continues with the instruction following the mloop_ opcode.

```
mloop            ( -- )                                    0011 1000
Micro Loop Unconditionally                                      0x38
                                                         1 CPU-clock
```

Decrement ct by one. If ct is non-zero transfer execution to the beginning of the current instruction group. If ct is zero continue execution with the instruction following mloop.

```
mloopc           ( -- )                                    0011 1001
Micro Loop if Carry                                             0x39
                                                         1 CPU-clock
```

Decrement ct by one. If ct is non-zero and carry is set transfer execution to the beginning of the current instruction group. If ct is zero or carry is clear continue execution with the instruction following mloopc.

```
mloopn
mloopnp          ( n -- n )                                0011 1010
Micro Loop if Negative/Not Positive                             0x3A
                                                         1 CPU-clock
```

Decrement ct by one. If ct is non-zero and *n* is negative (neither positive nor zero) transfer execution to the beginning of the current instruction group. If ct is zero or *n* is not negative (either positive or zero) continue execution with the instruction following mloopn or mloopnp.

```
mloopnc          ( -- )                                    0011 1101
Micro Loop if Not Carry                                         0x3D
                                                         1 CPU-clock
```

Decrement ct by one. If ct is non-zero and carry is clear transfer execution to the beginning of the current instruction group. If ct is zero or carry is set continue execution with the instruction following mloopnc.

```
mloopnn
mloopp           ( n -- n )                                0011 1110
Micro Loop if Not Negative/Positive                             0x3E
                                                         1 CPU-clock
```

Decrement ct by one. If ct is non-zero and *n* is not negative (either positive or zero) transfer execution to the beginning of the current instruction group. If ct is zero or *n* is negative (neither positive nor zero) continue execution with the instruction following mloopnn or mloopp.

ADVANCE INFORMATION

mloopnz (*n* -- *n*) 0011 1111
Micro Loop if Not Zero 0x3F
1 CPU-clock

Decrement `ct` by one. If `ct` is non-zero and *n* is not zero transfer execution to the beginning of the current instruction group. If `ct` is zero or *n* is zero continue execution with the instruction following `mloopnz`.

mloopz (*n* -- *n*) 0011 1011
Micro Loop if Zero 0x3B
1 CPU-clock

Decrement `ct` by one. If `ct` is non-zero and *n* is zero transfer execution to the beginning of the current instruction group. If `ct` is zero or *n* is not zero continue execution with the instruction following `mloopz`.

mulfs

Multiply Fast Signed

mulfs (*n1 n2* -- *n3 n4*) 1101 0110
0xD6
2 to 32 CPU-clocks

Multiply the bit-order-reversed value *n1* by the value in `g0` leaving the result *n4*. *n2* is usually zero and *n3* is garbage (see below). The number of significant bits in *n1* is indicated by the value in `ct`. All values are single-cell size and signed. `ct` is decremented to zero.

The program must supply *n1* in bit-order-reversed form (e.g., the binary value for decimal 13 is 01101 and bit-order reversed is 10110; note that the original high-order bit is zero as a sign bit and must be included.) The program must also load `ct` with the bit count and push a zero for *n2*. For the example number above, the count would be 5. *n3* is typically discarded.

n2 could be non-zero but its use in this form is questionable. The effect of *n2* on the result is that the value of *n2* shifted left by the bit count value in `ct` is added to the result, *n4*. *n3* contains the low cell of the value remaining after *n2n1* is shifted right by the number of bits in `ct`. Instruction execution time is limited to 65 CPU-clock cycles by the instruction expiration counter.

muls

Multiply Signed

```
muls              ( n1 n2 -- n3 n4 )                          1101 0101
                                                                   0xD5
                                                             32 CPU-clocks
```

Multiply *n1* by the value in `g0` and add *n2*, leaving the double result *n4n3*. All values are signed.

mulu

Multiply Unsigned

```
mulu              ( n1 n2 -- n3 n4 )                          1101 0111
                                                                   0xD7
                                                             32 CPU-clocks
```

Multiply *n1* by the value in `g0` and add *n2*, leaving the double result *n4n3*. All values are unsigned.

mxm

Maximum

```
mxm               ( n1 n2 -- n1 n2 )          carry set      1101 1111
             or   ( n1 n2 -- n2 n1 )          carry clear         0xDF
                                                              2 CPU-clocks
```

Compare *n2* and *n1* as signed values. Set `carry` if *n1* < *n2*, otherwise clear `carry`. Bring the larger of *n1* and *n2* to the top of stack. That is, if the resulting `carry` is set then *n2* is greater than *n1* and *n2* remains on top. If the resulting `carry` is clear then *n2* is less than or equal to *n1* and *n1* is exchanged with *n2*.

ADVANCE INFORMATION

neg

Two's-Complement Negation

```
neg                ( n1 -- n2 )                      1100 1001
                                                          0xC9
                                                   1 CPU-clock
```

n2 is the two's-complement negation of *n1*.

Equivalent to Java byte code `ineg`.

Equivalent to ANS Forth word `NEGATE`.

nop

No Operation

```
nop                ( -- )                            1110 1010
                                                          0xEA
                                                   1 CPU-clock
```

Do nothing.

Equivalent to Java byte code `nop`.

norml

Normalize Left

```
norml          ( n1 -- n2 )   if single precision                1100 0111
               ( n1 n2 -- n3 n4 )   if double precision               0xC7
                                                           1 to 13 CPU-clocks
               ( L: -- addr )   only when trap processed
                                                      3+M to 15+M CPU-clocks
               ( L: -- addr1 addr2 )  only when both traps processed
                                                    5+2M to 17+2M CPU-clocks
```

While the hidden bit and the seven bits to the right of it in *n1* (*n2* if double) are zero, repeat the following:
  Shift *n1* (or *n2n1* if double) left by eight bits and decrement the exponent field in `ct` by eight.
Then, while the hidden bit of *n1* (*n2* if double) is zero, repeat the following:
  Shift *n1* (or *n2n1* if double) left by one bit and decrement the exponent field in `ct` by one.

In both steps, bits shifted into bit zero of *n1* come from the GRS extension.

When the operation is complete, if shifting was required and the decremented field in `ct` reached or passed all zero bits during the processing, an underflow exception is signaled. If no shifting is required an underflow exception is not signaled. Then, if any bit in the GRS extension is set, a normalize exception is signaled. The location of the exponent field depends on `fp_precision`. If both traps are processed, the underflow trap has higher priority. Instruction execution time is limited to 65 CPU-clock cycles by the instruction expiration counter.

ADVANCE INFORMATION

normr

Normalize Right

normr (*n1* -- *n2*) if single precision 1100 0110
(*n1* *n2* -- *n3* *n4*) if double precision 0xC6
1 to 11 CPU-clocks
(L: -- *addr*) only when trap processed
3+M to 13+M CPU-clocks
(L: -- *addr1* *addr2*) only when both traps processed
5+2M to 15+2M CPU-clocks

While any bit except the first bit (the hidden bit) in the exponent field is non-zero, repeat the following:
Shift *n1* (or *n2n1* if double) right by one bit and increment the exponent field in ct by one. Bits shifted out of bit zero of *n1* shift into the GRS extension bits.

When the operation is complete, if shifting was required and the incremented field in ct reached or passed all one bits during the processing, an overflow exception is signaled. If no shifting is required an overflow exception is not signaled. Then, if the GRS extension is set, a normalization exception is signaled. The locations of the exponent field and hidden bit depend on fp_precision. If both traps are processed, the overflow trap has higher priority. Instruction execution time is limited to 65 CPU-clock cycles by the instruction expiration counter.

notc

Complement Carry

notc (--) carry inverted 1101 1101
0xDD
1 CPU-clock

Invert the state of carry.

or

Bitwise OR

```
or                ( n1 n2 -- n3 )                    carry clear  1110 0000
                                                                        0xE0
                                                                 1 CPU-clock
```

Perform a bitwise OR on *n1* and *n2* giving the result *n3*.

Equivalent to Java byte code `ior`.

Equivalent to ANS Forth word `OR`.

pop

pop (*n* --)
1011 0011
0xB3
1 CPU-clock

Discard *n*.

Equivalent to Java byte codes `pop`, `l2i`.
Equivalent when executed twice to Java byte code `pop2`.

Equivalent to ANS Forth word `D>S`, `DROP`, `FDROP`.
Equivalent when executed twice to ANS Forth word `2DROP`.

pop ct (*n* --)
1011 0100
0xB4
1 CPU-clock

Replace the value in `ct` with *n*.

pop g*i* (*n* --)
0101 xxxx
0x5?
1 CPU-clock

Replace the value in g*i* (global register *i*, i.e., `g0`–`g15`) with *n*. To eliminate contentions on registers `g1`–`g15`, if the DMAC or the VPU is using one of these global registers when the MPU attempts access, the MPU stalls until the registers are available. Contentions are not possible on `g0`.

pop la (*addr* --)
(L: …j_n…j_1 --)
1011 1101
0xBD
1+M CPU-clocks

Replace the value in `la` with cell-aligned address *addr*. The contents of the local-register stack cache, …j_n…j_1, are discarded. The two least-significant bits of `la` are cleared. The bit `ls_boundary` is cleared. A stack refill is performed at *addr*+4 to initialize `r0`.

pop lstack (*n* --)
(L: -- *n*)
1011 1010
0xBA
1 CPU-clock

Remove *n* from the operand stack and push it onto the local-register stack (into `r0`). The previous contents of `r0` are placed in `r1`, the previous contents of `r1` are placed in `r2`, and so on.

Equivalent to ANS Forth word `>R`.
Equivalent when executed twice to ANS Forth word `2>R`.

pop mode (*n* --) 1011 1001
0xB9
1 CPU-clock

Replace the value in mode with *n* and clear power_fail. The mode bits power_fail, ls_boundary and os_boundary are not writeable.

pop r*i* (*n* --) 1010 xxxx
0xA?
1 CPU-clock

Replace the value in r*i* (local register *i*, i.e., r0–r14) with *n*.

If r*i* is in the local-register stack cache (*i* ≤ ldepth) the value in r*i* is replaced with *n*. If r*i* is not currently in the local-register stack cache (*i* > ldepth), cells starting at r(ldepth+1) are read from memory sequentially to fill the cache until r*i* is reached. r*i* is then replaced with the value *n*.

Equivalent to Java byte codes astore_0, astore_1, astore_2, astore_3, fstore_0, fstore_1, fstore_2, fstore_3, istore_0, istore_1, istore_2, istore_3.
Equivalent when executed twice to Java byte codes dstore_0, dstore_1, dstore_2, dstore_3, lstore_0, lstore_1, lstore_2, lstore_3.
Equivalent for indexes up to fourteen (almost all actual cases) to Java byte codes astore (vindex), fstore (vindex), istore (vindex).
Equivalent when executed twice for indexes up to thirteen (almost all actual cases) to Java byte codes dstore (vindex), lstore (vindex).

pop sa ($\ldots j_n \ldots j_1$ *m1 m2 addr* -- *m1 m2*) 1011 1100
0xBC
1+M CPU-clocks

Replace the value in sa with cell-aligned address *addr*. The contents of the operand stack cache, $\ldots j_n \ldots j_1$, are discarded. The two least-significant bits of sa are cleared. The bit os_boundary is cleared. A stack refill is performed at *addr*+4 to initialize s2.

pop x (*n* --) 1011 1000
0xB8
1 CPU-clock

Replace the value in x with *n*.

push

push (*n* -- *n n*) 1001 0010
0x92
1 CPU-clock

Duplicate *n*.

Equivalent to Java byte code dup.

push ct (-- *n*) 1001 0100
0x94
1 CPU-clock

n is the value in ct.

push g*i* (-- *n*) 0111 xxxx
0x7?
1 CPU-clock

n is the value in g*i* (global register *i*, i.e., g0–g15). To eliminate contentions on registers g1–g15, if the DMAC or the VPU is using one of these global registers when the MPU attempts access, the MPU stalls until the registers are available. Contentions are not possible on g0.

push la (-- *addr*) 1001 1101
0x9D
1 CPU-clock

addr is the value in la.

push lstack (-- n) 1001 1010
0x9A
(L: n --) 1 CPU-clock

Pop *n* from the local-register stack (from r0) and push it onto the operand stack. The previous contents of r1 are placed in r0, the previous contents of r2 are placed in r1, and so on.

Equivalent to ANS Forth word R>.
Equivalent when executed twice to ANS Forth word 2R>.

push mode (-- *n*) 1001 0001
0x91
1 CPU-clock

n is the value in mode.

`push r`*`i`*	(-- *n*)		1000 xxxx 0x8? 1 CPU-clock

n is the value in `r`*`i`* (local register *i*, i.e. `r0`–`r14`).

If `r`*`i`* is in the local-register stack cache (*i* ≤ `ldepth`) the value in `r`*`i`* is pushed onto the operand stack. If `r`*`i`* is not currently in the local-register stack cache (*i* > `ldepth`), cells starting at `r(ldepth+1)` are read from memory sequentially until `r`*`i`* is reached. The value in `r`*`i`* is then pushed onto the operand stack.

Equivalent to Java byte codes `aload_0`, `aload_1`, `aload_2`, `aload_3`, `fload_0`, `fload_1`, `fload_2`, `fload_3`, `iload_0`, `iload_1`, `iload_2`, `iload_3`.
Equivalent when executed twice to Java byte codes `lload_0`, `lload_1`, `lload_2`, `lload_3`, `dload_0`, `dload_1`, `dload_2`, `dload_3`.
Equivalent for indexes up to fourteen (almost all actual cases) to Java byte codes `aload (vindex)`, `fload (vindex)`, `iload (vindex)`.
Equivalent when executed twice for indexes up to thirteen (almost all actual cases) to Java byte codes d`load (vindex)`, `lload (vindex)`.

Equivalent to ANS Forth word R@.
Equivalent when executed twice to ANS Forth word 2R`@.`

`push s`*`i`*	(-- *n*)	`s0`	1001 0010 0x92
		`s1`	1001 0011 0x93
		`s2`	1001 1110 0x9E 1 CPU-clock

n is the value in `s`*`i`* (operand stack register *i*, i.e., `s0`, `s1` or `s2`)

Equivalent to Java byte code `dup`.
Equivalent when executed twice to Java byte code `dup2`.

Equivalent to ANS Forth words `2DUP`, `DUP`, `FDUP`, `FOVER`, `OVER`.

`push sa`	(-- *addr*)		1001 1100 0x9C 1 CPU-clock

addr is the value in `sa`.

`push x`	(-- *n*)		1001 1000 0x98 1 CPU-clock

n is the value in `x`.

```
push.b #n          ( -- n )                                  1001 0000
                                                                  0x90
                                                           1 CPU-clock
```

n is an eight-bit literal value in the range 0–255. The byte literal is encoded as the last byte in the instruction group. This allows only one unique `push.b #` value per instruction group. Multiple `push.b #` opcodes in the same instruction group push the same value.

Equivalent for positive values to Java byte code `bipush`.
Equivalent for some values to Java byte code `sipush`.

```
push.l #n          ( -- n )                                  0100 1111
                                                                  0x4F
                                                          M CPU-clocks
```

n is a 32-bit literal value. The value is compiled as a full cell following the instruction group. Multiple `push.l #` in an instruction group are compiled with data in sequential cells following the instruction group in memory. As the `push.l #` opcodes are executed, the internally maintained next pc is incremented to move past each cell as it is fetched and pushed on the stack. Note that `skip`ping a `push.l #` causes the MPU to execute the literal value because the skipped `push.l #` will not have incremented next pc to move past the value.

Equivalent to Java byte code `fconst_1`, `fconst_2`, `ldc`, `ldc_w`, `sipush`.
Equivalent when executed twice to Java byte code `ldc2_w`.

```
push.n #n          ( -- n )                                  0010 xxxx
                                                                  0x2?
                                                           1 CPU-clock
```

n is a literal value in the range -7 to 8. The four least-significant bits of the opcode encode the value for *n*. The value is encoded as a two's-complement representation of *n* except that -8 (1000 binary) is decoded to be +8.

Equivalent to Java byte codes `aconst_null`, `fconst_0`, `iconst_m1`, `iconst_0`, `iconst_1`, `iconst_2`, `iconst_3`, `iconst_4`, `iconst_5`.
Equivalent for some values to Java byte code `bipush`.
Equivalent when executed twice to Java byte codes `dconst_0`, `lconst_0`, `lconst_1`.

Equivalent to ANS Forth words FALSE, TRUE.

replb

Replace Byte

```
replb            ( n1 n2 -- n3 )                       1101 1010
                                                            0xDA
                                                     1 CPU-clock
```

Replace the target byte of *n2* with the least-significant byte of *n1*, leaving the result *n3*. The target byte is selected by the two least-significant bits of x, as when accessing a byte in memory.

For example, if x = 0x121, *n1* = 0xCCDDEEFF, and *n2* = 0x12345678, then *n3* = 0x12FF5678.

replexp

Replace Exponent

```
replexp          ( n1 n2 -- n3 )                       1011 0101
                                                            0xB5
                                                     1 CPU-clock
```

Replace the exponent field and sign bits of *n1* with the corresponding bits of *n2*. Clear the GRS extension. The location of the exponent field depends on `fp_precision`.

ret

Return

```
ret              ( -- )                                0110 1110
                 ( L: addr -- )                             0x6E
Return from Subroutine                              M CPU-clocks
```

Pop *addr* from the local-register stack into pc to transfer execution to *addr*.

Equivalent to ANS Forth word EXIT.

```
reti             ( -- )                                0110 1111
                 ( L: addr -- )                             0x6F
Return from Interrupt                               M CPU-clocks
```

Pop *addr* from the local-register stack into pc to transfer execution to *addr*. Clear the current interrupt under-service bit.

ADVANCE INFORMATION

rev

Revolve Operand Stack

```
rev               ( n1 n2 n3 -- n2 n3 n1 )                     1110 0100
                                                                    0xE4
                                                             1 CPU-clock
```

Rotate the top three cells of the stack to bring *n1* to the top.

Equivalent to the run-time for the ANS Forth words FROT, ROT.

rnd

Round

```
rnd               ( n1 -- n2 )                    carry±       1101 0001
                                                                    0xD1
                                                             1 CPU-clock
                  ( L: -- addr )    only when trap processed   3+M CPU-clocks
```

Round *n1* giving *n2*. Rounding is based on fp_round_mode, the sign of ct, and the GRS extension. See *Rounding*, page 34. If an increment carried out of bit 31 then set carry, clear carry otherwise.

If the value of *n2* is different from *n1*, a rounded exception is signaled. The exception is detected as a change in the value of bit zero.

scache

See _cache.

sdepth

See _depth.

sexb

Sign-extend byte

sexb	(*n1* -- *n2*)	1101 1000 0xD8 1 CPU-clock

Copy the value of bit seven of *n1* into bits eight to thirty-one, leaving *n2*.

Equivalent to Java byte code `i2b`.

shift_

The number of CPU-clock cycles required to shift the specified number of bits depends on the number of bits requested. While the count ≥ eight the value (single or double) is shifted eight bits each CPU-clock cycle. When the count becomes less than eight the shifting is finished at one bit per CPU-clock cycle. For instance, the worst-case useful `shift` is 31 bits (either left or right) and takes eleven CPU-clock cycles—three 8-bit shifts and seven 1-bit shifts plus one CPU-clock cycle for setup. A 32-bit shift would take five CPU-clock cycles. The counts are modulo 64 in sign-magnitude representation using only the six least-significant bits for the magnitude and bit 31 for the sign. A zero in the six least-significant bits represents zero. (Sign-magnitude representation here is a positive integer count in the six least-significant bits, the middle bits ignored, and bit 31 indicating the sign, zero is positive, one is negative).

`shift` (*n1 n2* -- *n3*) carry± (*n2*>0)1110 1110
0xEE
1 to 11 CPU-clocks

Shift *n1* by |*n2*| bits leaving the result *n3*. If *n2* is positive the shift is to the left, each bit is shifted out through `carry`, and zero is shifted into each bit on the right. If *n2* is negative the shift is to the right, each bit shifted out is shifted through the GRS extension, and `carry` is copied into each high order bit of *n1* vacated by the shift. See text above regarding execution time and format of negative counts.

Equivalent to ANS Forth word `LSHIFT`.

`shiftd` (*n1 n2 n3* -- *n4 n5*) carry± (*n3*>0)1110 1111
Shift Double 0xEF
1 to 15 CPU-clocks

Shift the cell pair *n2n1* by |*n3*| bits leaving the resulting cell pair *n5n4*. If *n3* is positive the shift is to the left, each bit is shifted out of *n2* through `carry`, and zero is shifted into each bit on the right into *n1*. If *n3* is negative the shift is to the right, each bit shifted out of *n1* is shifted through the GRS extension, and `carry` is copied into each high order bit of *n2* vacated by the shift. See text above regarding execution time and format of negative counts.

shl_

Shift Left

```
shl #1             ( n1 -- n2 )                      carry±       1110 0010
Shift Left                                                             0xE2
                                                                1 CPU-clock
```

Shift *n1* one bit to the left leaving the result *n2*. The high order bit of *n1* shifted out goes into `carry`. The vacated bit on the right of *n1* is filled with zero.

Equivalent to ANS Forth word `2*`.

```
shl #8             ( n1 -- n2 )                      carry±       1110 1100
Shift Left Byte                                                        0xEC
                                                                1 CPU-clock
```

Shift *n1* eight bits (one byte) to the left leaving *n2*. The last bit shifted out goes into `carry`. The vacated eight bits on the right are filled with zeros.

```
shld #1            ( n1 n2 -- n3 n4 )                carry±       1110 0110
Shift Left Double                                                      0xE6
                                                                1 CPU-clock
```

Shift cell pair *n2n1* one bit to the left leaving the result *n4n3*. The high order bit of *n2* shifted out goes into `carry`. The vacated bit on the right of *n1* is filled with zero.

Equivalent to ANS Forth word `D2*`.

ADVANCE INFORMATION

shr_

Shift Right

shr #1 (*n1* -- *n2*) 1110 0011
Shift Right 0xE3
1 CPU-clock

Shift *n1* one bit to the right leaving the result *n2*. The bit shifted out is shifted into the GRS extension. The vacated bit on the left is filled with `carry`.

shr #8 (*n1* -- *n2*) 1110 1101
Shift Right Byte 0xED
1 CPU-clock

Shift *n1* eight bits (one byte) to the right leaving the result *n2*. The bits shifted out are shifted into the GRS extension. The vacated eight bits on the left are filled with `carry`.

shrd #1 (*n1 n2* -- *n3 n4*) 1110 0111
Shift Right Double 0xE7
1 CPU-clock

Shift cell pair *n2n1* one bit to the right leaving the result *n4n3*. The bit shifted out of *n1* is shifted into the GRS extension. The vacated bit in *n2* on the left is filled with `carry`.

skip_

Skip if Condition

skip conditionally or unconditionally skips execution of the remainder of the instruction group. If the condition is true, skip the remainder of the instruction group and continue execution with the following instruction group. If condition is false, continue execution with the next instruction.

WARNING: Do not skip a push.l #. Since the MPU will not have executed the push.l # opcode, the corresponding literal cell is not skipped. The result will be the MPU executing the literal cell.

```
skip                ( -- )                                  0011 0000
Skip Unconditionally                                             0x30
                                             Mprefetch CPU-clocks
```

Unconditionally skip the remainder of the instruction group.

```
skipc               ( -- )                                  0011 0011
Skip if Carry                                                    0x31
                           1 (no carry)   Mprefetch (carry) CPU-clocks
```

If carry is set, skip the remainder of the instruction group and continue execution with the next instruction group; otherwise, continue execution with the next instruction.

```
skipn
skipnp              ( n -- )                                0011 0010
Skip if Negative/Not Positive                                    0x32
                             1 (not neg)   Mprefetch (neg) CPU-clocks
```

If *n* is negative (neither positive nor zero), skip the remainder of the instruction group and continue execution with the next instruction group; otherwise, continue execution with the next instruction.

```
skipnc              ( -- )                                  0011 0111
Skip if Not Carry                                                0x35
                                                     M CPU-clocks
```

If carry is clear, skip the remainder of the instruction group and continue execution with the next instruction group; otherwise, continue execution with the next instruction.

```
skipnn
skipp               ( n -- )                                0011 0110
Skip if Not Negative/Positive                                    0x36
                             1 (neg)   Mprefetch (not neg) CPU-clocks
```

If *n* is not negative (either positive or zero), skip the remainder of the instruction group and continue execution with the next instruction group; otherwise, continue execution with the next instruction.

```
skipnz             ( n -- )                                     0011 0001
Skip if Not Zero                                                     0x37
                               1 (zero)   Mprefetch (non-zero) CPU-clocks
```

If *n* is not zero, skip the remainder of the instruction group and continue execution with the next instruction group; otherwise, continue execution with the next instruction.

```
skipz              ( n -- )                                     0011 0101
Skip if Zero                                                         0x33
                               1 (non-zero)   Mprefetch (zero) CPU-clocks
```

If *n* is zero, skip the remainder of the instruction group and continue execution with the next instruction group; otherwise, continue execution with the next instruction.

split

Split Cell

```
split              ( n1 -- n2 n3 )                              1001 1001
                                                                     0x99
                                                              1 CPU-clock
```

Split *n1* into two parts so that the lower-half of *n1* is in the lower-half of *n2* and the upper-half of *n1* is in the lower-half of *n3*.

For example, if *n1* = 0x12345678 then *n2* = 0x5678 and *n3* = 0x1234.

st

Store Indirect to Memory

`st [--r0]` (*n* --) 0110 0100

0x64

1+M CPU-clocks

Decrement `r0` by four. Store the cell *n* into memory at the new address in `r0`. The two least-significant bits of the address are ignored and treated as zero.

`st [--x]` (*n* --) 0110 1000

0x68

1+M CPU-clocks

Decrement `x` by four. Store the cell *n* into memory at the new address in `x`. The two least-significant bits of the address are ignored and treated as zero.

`st [r0++]` (*n* --) 0110 0110

0x66

M CPU-clocks

Store the cell *n* into memory at the address in `r0`. Increment `r0` by four. The two least-significant bits of the address are ignored and treated as zero.

`st [r0]` (*n* --) 0110 0010

0x62

M CPU-clocks

Store the cell *n* into memory at the address in `r0`. The two least-significant bits of the address are ignored and treated as zero.

`st [x++]` (*n* --) 0110 1001

0x69

M CPU-clocks

Store the cell *n* into memory at the address in `x`. Increment `x` by four. The two least-significant bits of the address are ignored and treated as zero.

`st [x]` (*n* --) 0110 0001

0x61

M CPU-clocks

Store the cell *n* into memory at the address in `x`. The two least-significant bits of the address are ignored and treated as zero.

```
st []             ( n addr -- n )                    0110 0000
                                                          0x60
                                                  M CPU-clocks
```

Store the cell *n* into memory at address *addr*. The two least-significant bits of the address are ignored and treated as zero.

step

Single-Step Processor

```
step              ( -- )                             0011 0100
                  ( L: addr1 -- addr2 )                   0x34
                                            2M+2+inst CPU-clocks
```

Pop *addr1* from the local-register stack into pc and continue execution at *addr1* for one instruction. Then perform a call subroutine to the single-step trap location, 0x138. *addr2* is the address of the next instruction following *addr1*.

sto

Store Indirect to On-Chip Resource

```
sto []            ( n addr -- n )                    1011 0000
                                                          0xB0
                                                   1 CPU-clock
```

Store *n* into the on-chip resource register at address *addr*. The programmer must ensure that `sto []` is not executed to access (even if not changed) any configuration register containing information for a memory group with a bus transaction in process. For valid values of *addr*, see *On-Chip Resource Registers*, page 129.

```
sto.i []          ( n bit_addr -- n )                1011 0001
                                                          0xB1
                                                   1 CPU-clock
```

If *n* is non-zero, set the bit at the on-chip resource register address *bit_addr*; otherwise, clear the bit. For valid values of *addr*, see *On-Chip Resource Registers*, page 129.

sub

Subtract

```
sub                ( n1 n2 -- n3 )                 carry±       1100 1000
                                                                     0xC8
                                                              1 CPU-clock
```

Subtract *n2* from *n1* leaving the difference *n3*. If computing the difference required a borrow, `carry` is set; otherwise, `carry` is cleared.

Equivalent to Java byte code `isub`.

Equivalent to ANS Forth word `-`.

subb

Subtract with Borrow

```
subb               ( n1 n2 -- n3 )                 carry±       1100 1010
                                                                     0xCA
                                                              1 CPU-clock
```

Subtract *n2* and `carry` from *n1* leaving the difference *n3*. If computing the difference required a borrow, `carry` is set; otherwise, `carry` is cleared.

subexp

Subtract Exponents

```
subexp          ( n1 n2 -- n3 n4 n5 )                              1101 0011
                                                                        0xD3
                                                                2 CPU-clocks
                ( L: -- addr )   only when trap processed     4+M CPU-clocks
```

Perform the following:

Exponent_Field(*n5*) = Exponent_Field(*n1*) - Exponent_Field(*n2*) + BIAS - 1

Sign_Bit(*n5*) = Sign_Bit(*n1*) XOR Sign_Bit(*n2*)

BIAS is 127 (0x3F800000 in bit position) for single precision and 1023 (0x3FF00000 in bit position) for double precision, as selected by `fp_precision`.

Compute as described above. Clear the exponent-field bits and sign bit and set the hidden bit of *n1* and *n2* giving *n3* and *n4*, respectively. *n5* is the result of the computation. After completion, if the exponent-field calculation result equaled or exceeded the maximum value of the exponent field (exponent result ≥ 255 for single, exponent result ≥ 2047 for double) an overflow exception is signaled. If the exponent-field calculation result is less than or equal to zero an underflow exception is signaled. When an exception is signaled, the exponent field of *n5* contains as many bits of the result as it will hold.

testb

Test Bytes for Zero

```
testb           ( n -- n )                         carry±          1101 1001
                                                                        0xD9
                                                                 1 CPU-clock
```

If any byte of *n* is zero set `carry`, otherwise clear `carry`.

testexp

Test Exponent

testexp (*n1 n2* -- *n1 n2*) carry± 1101 0100
0xD4
1 CPU-clock
(L: -- *addr*) only when trap processed 3+M CPU-clocks

Clear the GRS extension. If the exponent field in *n1* or *n2* is all zeros or all ones, an exponent exception is signaled and carry is set; otherwise, carry is cleared. The location of the exponent field depends on fp_precision.

xcg

Exchange

xcg (*n1 n2* -- *n2 n1*) 1011 0010
0xB2
1 CPU-clock

Exchange the top two operand stack cells.

Equivalent to Java byte code swap.

Equivalent to the ANS Forth words FSWAP, SWAP.

xor

Bitwise Exclusive OR

xor (*n1 n2* -- *n3*) carry clear 1100 0011
0xC3
1 CPU-clock

Perform a bitwise EXCLUSIVE OR of *n1* and *n2* giving the result *n3*.

Equivalent to Java byte code ixor.

Equivalent to ANS Forth word XOR.

ADVANCE INFORMATION

Table 39. MPU Mnemonics and Opcodes (Mnemonic Order)

Mnemonic		Opcode	Mnemonic		Opcode	Mnemonic		Opcode	Mnemonic		Opcode
add		c0	lframe		be	pop	r0	a0	push	r3	83
add	pc	bb	mloop		38	pop	r1	a1	push	r4	84
adda		e8	mloopc		39	pop	r2	a2	push	r5	85
addc		c2	mloopn		3a	pop	r3	a3	push	r6	86
addexp		d2	mloopnc		3d	pop	r4	a4	push	r7	87
and		e1	mloopnn		3e	pop	r5	a5	push	r8	88
bkpt		3c	mloopnp		3a	pop	r6	a6	push	r9	89
br	*offset*	00…	mloopnz		3f	pop	r7	a7	push	r10	8a
br	[]	4b	mloopp		3e	pop	r8	a8	push	r11	8b
bz	*offset*	10…	mloopz		3b	pop	r9	a9	push	r12	8c
call	*offset*	08…	mulfs		d6	pop	r10	aa	push	r13	8d
call	[]	4e	muls		d5	pop	r11	ab	push	r14	8e
cmp		cb	mulu		d7	pop	r12	ac	push	s0	92
copyb		d0	mxm		df	pop	r13	ad	push	s1	93
dbr	*offset*	18…	neg		c9	pop	r14	ae	push	s2	9e
dec	#1	cf	nop		ea	pop	sa	bc	push	sa	9c
dec	#4	cd	norml		c7	pop	x	b8	push	x	98
dec	ct	c1	normr		c6	push		92	push.b	*#byte*	90
denorm		c5	notc		dd	push	ct	94	push.l	*#cell*	4f
di		b7	or		e0	push	g0	70	push.n	#-7	29
divu		de	pop		b3	push	g1	71	push.n	#-6	2a
ei		b6	pop	ct	b4	push	g2	72	push.n	#-5	2b
eqz		e5	pop	g0	50	push	g3	73	push.n	#-4	2c
expdif		c4	pop	g1	51	push	g4	74	push.n	#-3	2d
extexp		db	pop	g2	52	push	g5	75	push.n	#-2	2e
extsig		dc	pop	g3	53	push	g6	76	push.n	#-1	2f
iand		e9	pop	g4	54	push	g7	77	push.n	#0	20
inc	#1	ce	pop	g5	55	push	g8	78	push.n	#1	21
inc	#4	cc	pop	g6	56	push	g9	79	push.n	#2	22
lcache		4d	pop	g7	57	push	g10	7a	push.n	#3	23
ld	[--r0]	44	pop	g8	58	push	g11	7b	push.n	#4	24
ld	[--x]	4a	pop	g9	59	push	g12	7c	push.n	#5	25
ld	[r0++]	46	pop	g10	5a	push	g13	7d	push.n	#6	26
ld	[r0]	42	pop	g11	5b	push	g14	7e	push.n	#7	27
ld	[x++]	49	pop	g12	5c	push	g15	7f	push.n	#8	28
ld	[x]	41	pop	g13	5d	push	la	9d	replb		da
ld	[]	40	pop	g14	5e	push	lstack	9a	replexp		b5
ld.b	[]	48	pop	g15	5f	push	mode	91	ret		6e
ldo	[]	96	pop	la	bd	push	r0	80	reti		6f
ldo.i	[]	97	pop	lstack	ba	push	r1	81	rev		e4
ldepth		9b	pop	mode	b9	push	r2	82	rnd		d1

Table 39. MPU Mnemonics and Opcodes (Mnemonic Order, continued)

Mnemonic		Opcode	Mnemonic		Opcode	Mnemonic		Opcode	Mnemonic		Opcode
scache		45	shr	#8	ed	skipz		33	sto	[]	b0
sdepth		9f	shrd	#1	e7	split		99	sto.i	[]	b1
sexb		d8	skip		30	st	[--r0]	64	sub		c8
sframe		bf	skipc		31	st	[--x]	68	subb		ca
shift		ee	skipn		32	st	[r0++]	66	subexp		d3
shiftd		ef	skipnc		35	st	[r0]	62	testb		d9
shl	#1	e2	skipnn		36	st	[x++]	69	testexp		d4
shl	#8	ec	skipnp		32	st	[x]	61	xcg		b2
shld	#1	e6	skipnz		37	st	[]	60	xor		e3
shr	#1	e3	skipp		36	step		34			

ADVANCE INFORMATION

Table 40. MPU Mnemonics and Opcodes (Opcode Order)

Opcode	Mnemonic	Opcode	Mnemonic	Opcode	Mnemonic	Opcode	Mnemonic
00…07	br *offset*	45	scache	72	push g2	9e	push s2
08…0f	call *offset*	46	ld [r0++]	73	push g3	9f	sdepth
10…17	bz *offset*	47		74	push g4	a0	pop r0
18…1f	dbr *offset*	48	ld.b []	75	push g5	a1	pop r1
20	push.n #0	49	ld [x++]	76	push g6	a2	pop r2
21	push.n #1	4a	ld [--x]	77	push g7	a3	pop r3
22	push.n #2	4b	br []	78	push g8	a4	pop r4
23	push.n #3	4c		79	push g9	a5	pop r5
24	push.n #4	4d	lcache	7a	push g10	a6	pop r6
25	push.n #5	4e	call []	7b	push g11	a7	pop r7
26	push.n #6	4f	push.l *#cell*	7c	push g12	a8	pop r8
27	push.n #7	50	pop g0	7d	push g13	a9	pop r9
28	push.n #8	51	pop g1	7e	push g14	aa	pop r10
29	push.n #-7	52	pop g2	7f	push g15	ab	pop r11
2a	push.n #-6	53	pop g3	80	push r0	ac	pop r12
2b	push.n #-5	54	pop g4	81	push r1	ad	pop r13
2c	push.n #-4	55	pop g5	82	push r2	ae	pop r14
2d	push.n #-3	56	pop g6	83	push r3	af	
2e	push.n #-2	57	pop g7	84	push r4	b0	sto []
2f	push.n #-1	58	pop g8	85	push r5	b1	sto.i []
30	skip	59	pop g9	86	push r6	b2	xcg
31	skipc	5a	pop g10	87	push r7	b3	pop
32	skipn	5b	pop g11	88	push r8	b4	pop ct
32	skipnp	5c	pop g12	89	push r9	b5	replexp
33	skipz	5d	pop g13	8a	push r10	b6	ei
34	step	5e	pop g14	8b	push r11	b7	di
35	skipnc	5f	pop g15	8c	push r12	b8	pop x
36	skipnn	60	st []	8d	push r13	b9	pop mode
36	skipp	61	st [x]	8e	push r14	ba	pop lstack
37	skipnz	62	st [r0]	8f		bb	add pc
38	mloop	63		90	push.b *#byte*	bc	pop sa
3a	mloopnp	66	st [r0++]	92	push s0	bf	sframe
3b	mloopz	67		93	push s1	c0	add
3c	bkpt	68	st [--x]	94	push ct	c1	dec ct
3d	mloopnc	69	st [x++]	95		c2	addc
3e	mloopnn	6a		96	ldo []	c3	xor
3e	mloopp	6b		97	ldo.i []	c4	expdif
3f	mloopnz	6c		98	push x	c5	denorm
40	ld []	6d		99	split	c6	normr
41	ld [x]	6e	ret	9a	push lstack	c7	norml
42	ld [r0]	6f	reti	9b	ldepth	c8	sub
43		70	push g0	9c	push sa	c9	neg
44	ld [--r0]	71	push g1	9d	push la	ca	subb

Table 40. MPU Mnemonics and Opcodes (Opcode Order, continued)

Opcode	Mnemonic	Opcode	Mnemonic	Opcode	Mnemonic	Opcode	Mnemonic
cb	cmp	d9	testb	e7	shrd #1	f5	
cc	inc #4	da	replb	e8	adda	f6	
cd	dec #4	db	extexp	e9	iand	f7	
ce	inc #1	dc	extsig	ea	nop	f8	
cf	dec #1	dd	notc	eb		f9	
d0	copyb	de	divu	ec	shl #8	fa	
d1	rnd	df	mxm	ed	shr #8	fb	
d2	addexp	e0	or	ee	shift	fc	
d3	subexp	e1	and	ef	shiftd	fd	
d4	testexp	e2	shl #1	f0		fe	
d5	muls	e3	shr #1	f1		ff	
d6	mulfs	e4	rev	f2			
d7	mulu	e5	eqz	f3			
d8	sexb	e6	shld #1	f4			

ADVANCE INFORMATION

Virtual Peripheral Unit

The Virtual Peripheral Unit (VPU) is a special-purpose processing unit that executes instructions to transfer data between devices and memory, refresh dynamic memory, measure time, manipulate bit inputs and bit outputs, and perform system timing functions. With these functions the VPU can be programmed to emulate serial ports, analog to digital converters, digital to analog converters, PWM outputs, timers, and other peripherals. VPU programs are usually written to be entirely temporally deterministic. Because it can be difficult or impossible to write programs that contain conditional execution paths that execute in an efficient temporally deterministic manner, the VPU contains no computational and minimal decision-making ability. VPU programs are intended to be relatively simple, using interrupts to the MPU to perform computation or decision making.

To ensure temporally deterministic execution, the VPU exercises absolute priority over bus access. Bus timing must *always* be deterministic; “wait states” of fixed length are programmed in the MIF. Temporal determinism is achieved by counting VPU-execution and bus CPU-clock cycles between the timed VPU events. Bus access is granted to the VPU unless it is executing `delay`, which allows MPU and DMA requests access to the bus during a specified time. Thus, when a memory access is required, the VPU simply seizes the bus and performs the required operation at precisely the programmed instant.

The MIF ensures that the bus is available when the VPU requires it. The MPU and the DMAC request the bus from the MIF, which prioritizes the requests and grants the bus while the VPU is executing `delay`. The MIF ensures that any transactions are completed before the delay time of the VPU expires and the VPU next requires the bus.

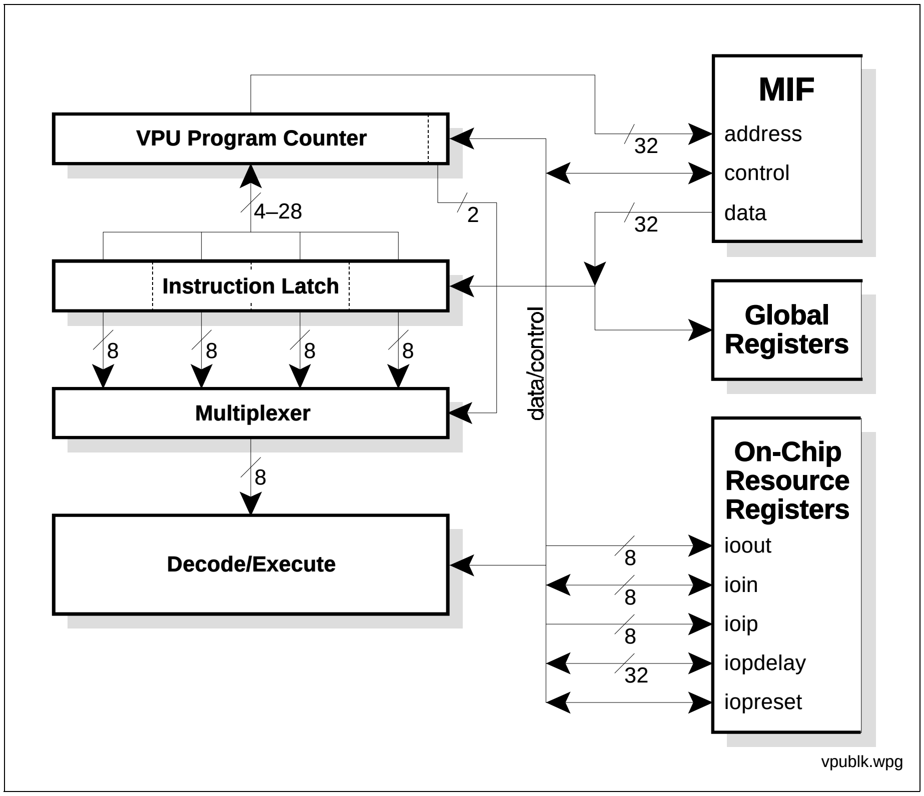


Figure 12. VPU Block Diagram

When transferring data, the VPU does not modify any data that is transferred; it only causes the bus transaction to occur at the programmed time. It performs time-synchronous I/O-channel transfers, as opposed to the DMAC, which prioritizes and performs asynchronous I/O-channel transfers. Other than how they are initiated, the two types of transfers are identical.

Usage

A VPU program can be used to eliminate external logic and simplify system designs. By using the VPU for timing-dependent system and application operations, timing constraints on the MPU program can often be eliminated or greatly relaxed. Additionally, the VPU with the assistance of the MPU can emulate a wide variety of system peripherals including serial ports, analog to digital converters, digital to analog converters, PWM outputs, timers, and other peripherals.

For example, a VPU program of about 150 bytes supplies the data transfers and timing for a video display . The program produces vertical and horizontal sync, and transfers data from DRAM to a video shift register or palette. Additionally, the VPU supplies flexibility. Video data from various areas of memory could be displayed, without requiring that the data be moved to create a contiguous frame buffer. As new data areas are specified, the VPU instructions are rewritten by the MPU to change the program the VPU executes for the next video frame. While this is executing, the MPU still has access to the bus to execute instructions and process data, and the DMAC still has access to the bus to transfer data.

Many other applications are possible. The VPU is best used for applications that require data to be moved, or some other event to occur, at specific times. For example:

- sending digitized 16-bit data values to a pair of DACs to play CD-quality stereo sound,
- sampling data from input devices at specified time intervals for the MPU to later process,
- sending data and control signals to display images on an LCD display,
- transferring data packets for an intelligent network interface,
- transferring synchronous data blocks for an intelligent SCSI controller,
- sending multiple channels of data to DACs for a wave-table synthesizer,
- controlling video and I/O for serial and X-Windows video terminals or PC video accelerators,
- controlling timed events in process-control environments,
- controlling ignition and fuel for automotive engines,
- inputting and outputting serial data streams,
- producing PWM output directly or for integration by an external R-C network for a low-cost digital to analog converter, or
- combining several of the above to significantly reduce system cost.

The VPU is designed to dictate access to the bus (to ensure temporally deterministic execution), but to be a slave to the MPU. The VPU can communicate status to the MPU by:

- the status changing on a device the VPU has accessed,
- loading a value in a global register,
- setting a bit output, or
- consuming a bit input.

The MPU can control the VPU by:

- rewriting VPU instructions in memory,
- modifying the global registers the VPU is using,
- clearing a bit input, or
- resetting the VPU.

The events controlled are not required to occur at a persistent, constant rate. The VPU is appropriate for applications whose event rates must be consistently controlled, whether once or many times. As an example of the former, the VPU can take audio data from memory and send it to a DAC to play the sound at a continuous rate, for as long as the audio clip lasts. As an example of the latter, the VPU can be synchronized to the rotation of an automotive engine by the MPU in order for the VPU to time fuel injection and ignition, with the synchronization constantly changed by the MPU (by changing global registers or rewriting the VPU program) as the MPU monitors engine performance.

Resources

The VPU consists of instruction decode and execution processes, and control paths to other CPU resources, as shown in Figure 12. The VPU and related registers include:

- Bit input register, `ioin`: bit inputs configured as DMA or interrupt requests, or general bit inputs. See Figure 26, page 131.
- Interrupt pending register, `ioip`: indicates which interrupts have been recognized but are waiting to be prioritized and serviced. See Figure 27, page 132.
- Bit output register, `ioout`: bits that were last written by either the MPU or the VPU. See Figure 29, page 134.
- VPU reset register, `vpureset`: writing any value causes the VPU to begin execution at the VPU software reset address. See Figure 51, page 155.
- Global registers `g1` through `g7`: contain values used by `delay`.
- Global registers `g8` through `g15`: contain loop counts or I/O-channel transfer specifications. Transfer specifications consist of device and memory transfer addresses and control bits. See Figure 16, page 104.

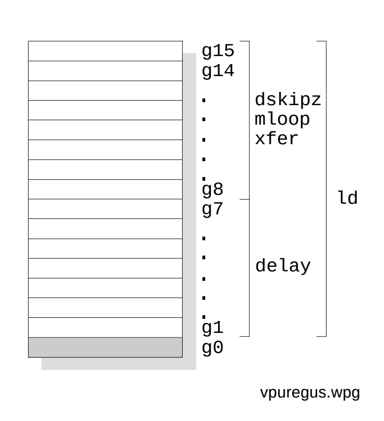


Figure 13. VPU Register Usage

Register Usage

The VPU shares global registers `g1–g15` with the MPU, and uses them for loop counts, `delay` initialization counts, and `transfer` information. See Figure 13. Loop counts and `delay` counts are 32 bits. Transfer addresses in bits 31–2 typically address cells, but can also address bytes, depending on the I/O-channel configuration. Bit one determines whether the transfer is a memory write or a memory read, and bit zero enables interrupts on 1024-byte memory page boundary crossings (see *Interrupts*, below). See Figure 16, page 104.

The MPU can read or write any registers used by the VPU at any time. If there is a register-access contention between the MPU and the VPU, the MPU is held off until the VPU access is complete.

Table 41. VPU Instructions

DELAY	NO OPERATION
DECREMENT AND SKIP	OUTPUT TRUE
INTERRUPT MPU	OUTPUT FALSE
JUMP	REFRESH
LOAD REGISTER	TEST INPUT AND SKIP
MICRO-LOOP	TRANSFER

Instruction Set

Table 41 lists the VPU instructions; Table 44 and Table 45, page 101, list the mnemonics and opcodes. Details of instruction execution are given in *Instruction Reference*, page 95.

Instruction Formats

All instructions consist of eight bits except for `ld`, which requires 32-bit immediate data, and `jump`, which requires a page-relative destination address. The use of eight-bit instructions allows up to four instructions (referred to as an *instruction group*) to be obtained on each instruction fetch, thus reducing memory-bandwidth requirements compared to typical 32-bit processors. This characteristic also allows looping on the instruction group (a micro-loop) without additional instruction fetches, further increasing efficiency. Instruction formats are depicted in Figure 14.

Jumps

The instruction `jump` is variable-length. The `jump` opcode can occur in any position within the instruction group. The four least-significant bits in the opcode and all of the bits in the current instruction group to the right of the opcode are used for the page-relative destination address. See Figure 14 and Table 42. The size of the encoded page-relative destination address depends on the location of the opcode within the current instruction group. The bits are used to replace the same cell-address bits within the next VPU `pc`. These destination addresses are cell-aligned to maximize the range of the destination address bits and the number of instructions that are executed at the destination. The next VPU `pc` is the cell-aligned address following the current instruction group, incremented for each `ld` instruction that preceded the `jump` in the current instruction group. If the destination address bits are not of sufficient range for the `jump` to reach the destination, the `jump` must be moved to an instruction group where more destination address bits are available.

Table 42. VPU Branch Ranges

Bits	Page-Relative Range
4	64 bytes
12	4096 bytes
20	1048576 bytes
28	268435456 bytes
Encoded bits replace the same number of bits from A2 upward in the VPU next PC; A1 and A0 are zero.	

Literals

The instruction `ld` requires a total of 40 bits, eight bits for the opcode in the current instruction group, and 32 bits following the current instruction group for the literal data. The `ld` opcode can occur in any position within the instruction group. The data for the first `ld` in an instruction group immediately follows the instruction group in memory; the data for each subsequent `ld` occupies successive locations. The four least-significant bits in the opcode contain the number of the global register that is the destination for the data. Global register zero (`g0`) is not allowed.

Others

All other instructions require eight bits. Most have a register or bit number encoded in the three or four least-significant bits of the opcode. See *Instruction Reference*, page 95, for details on the other individual instructions.

Execution Timing

Counting execution CPU-clocks cycles is the key to programming the VPU. Each instruction requires execution time as described in *Instruction Reference*, page 95. In general, instructions execute in one CPU-clock cycle, and, if they require a bus transaction, the instruction execution overlaps the time for the bus transaction. A timing with an “M” indicates the specified number of bus requests and bus transactions (memory cycles) for the instruction to complete. Bus requests require two CPU-clock cycles and bus transaction times are as programmed and described in *Programmable Memory Interface*, page 117, and *Bus Operation*, page 157.The value used for “M” includes both the bus request and bus transaction times.

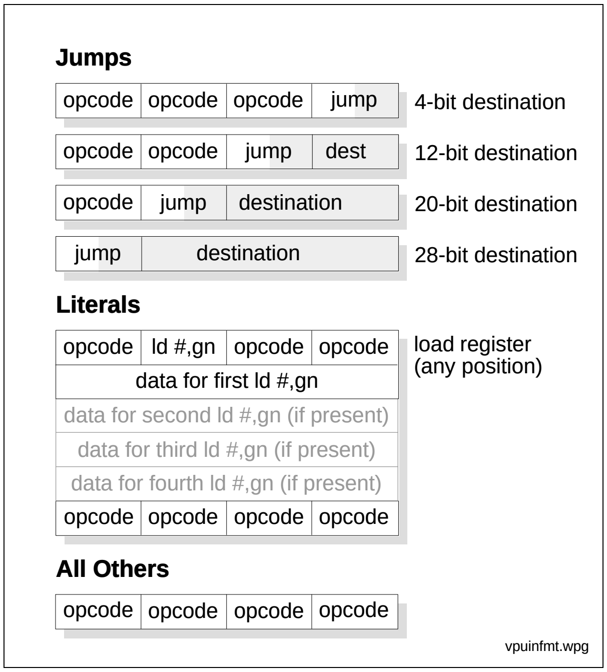

Jumps
opcode | opcode | opcode | jump
4-bit destination
opcode | opcode | jump | dest
12-bit destination
opcode | jump | destination
20-bit destination
jump | destination
28-bit destination
Literals
opcode | ld #,gn | opcode | opcode
load register (any position)
data for first ld #,gn
data for second ld #,gn (if present)
data for third ld #,gn (if present)
data for fourth ld #,gn (if present)
opcode | opcode | opcode | opcode
All Others
opcode | opcode | opcode | opcode
vpuinfmt.wpg

Figure 14. VPU Instruction Formats

Additionally, instruction fetch between the execution of instruction groups must be considered and requires “M” CPU-clock cycles. There is no instruction prefetch in the VPU, so timing computation is simplified. When execution of the instructions in an instruction group completes, instruction fetch begins during the next CPU-clock cycle.

To ensure deterministic timing, the programmer must keep track of the addresses being accessed and whether or not a RAS cycle or a CAS cycle will occur. This is fairly simple. There are only two cases in which RAS cycles occur in the VPU:

1. A RAS cycle is forced by the VPU on the first bus transaction to each memory group after the execution of `delay` or `refresh`. This guarantees, regardless of whether or not the current RAS page on a memory group is the target page, that the bus timing will be known: a RAS cycle.

2. A RAS cycle occurs when the memory page accessed is not the current RAS page on the target memory group. While this seems unknowable, with case 1, above, and a little care, it is easy to know if the target page is the current page. Case 1 eliminates all possibilities of the MPU or DMA making bus access timing non-deterministic. This limits RAS cycles to only those caused by the VPU program. Here, again, other than at initialization, there are only two cases:

A. Locating the VPU program to fully reside within a single RAS page, or in SRAM, eliminates RAS cycles due to instruction fetch page crossings. Alternatively, so long as the location of the page crossing is known, the RAS cycle can be considered in the VPU programming execution timing.

B. Planning of data transfers with the instruction `xfer` allows timing to be known and considered. Placing data transfer buffers fully within a single RAS page, or planning the starting address to know when page crossings occur, allows deterministic timing.

Techniques

Creating correct timing in A VPU program is matter of counting instruction executions and determining the type of memory accesses and the bus transaction times involved. Most simple, and many complex, programs executes an infinite loop. More complex programs execute continually changing program code.

- Straight in-line code is the easiest to program as there is only one path and no inner loops. Simply count the cycles through the path to determine the timing.
- `mloops` are also simple to program. The first access to the instruction group will require a bus transaction, but subsequent iterations will execute the instruction group without refetching the instructions.
- Counted program loops (other than `mloops`) are a little more complex. They are programmed using:

```
backward_label::
   …                  ; put loop body here
   dskipz             gx, forward_label
   jump               backward_label
forward_label::
```

They are more complex because the exit timing is one CPU-clock cycle shorter than the looping timing.

Table 43. Code Example: VPU DRAM Refresh

```
; VPU DRAM Refresh
;
; A typical 256K DRAM requires 512 refreshes every
; 8 ms. That means we require a refresh every
; 15.625 us, or a total loop time below of 31.250 us
; since we do two refreshes per loop. Assuming a RAS
; cycle with the bus request takes 11 CPU-clock cycles,
; the loop below takes 11 + 11 + 2 + delay or 35 + delay
; CPU-clock cycles to execute.

    External_clock     = 50000000   ; Hz
    CPU_clock          = (External_clock * 2)/100000
                                    ; x100KHz
    HndrdKHz_per_ns = 10000         ; scaling factor

VPU_start::
    ; Enter here from A VPU software reset.
    ; Total time to be taken by one loop iteration in
    ; nanoseconds.
    Loop_ns            = 31250

    ; Number of CPU-clocks required by
    ; instructions except delay time.
    Overhead_clocks  = 35

    ; Instruction overhead in nanoseconds.
    Overhead_ns        = ((Overhead_clocks *
            HndrdKHz_per_ns) / CPU_clock)

    ; CPU-clock delay value required to achieve
    ; Loop_ns above.
    Refresh_delay      = (Loop_ns - Overhead_ns) /
             (HndrdKHz_per_ns / CPU_clock)

    ld    #Refresh_delay,g7
                                    ; Inst. Fetch, 11
VPU_Refresh_Loop::

    refresh                         ; 11
    refresh                         ; 11

    delay     g7                    ; 2 + delay
    jump      VPU_Refresh_Loop      ; 11
```

ADVANCE INFORMATION

- Maintaining consistent timing on an event that is repeated throughout the program containing loops is even more complex. A good example of such a requirement is video generation, where horizontal sync must be maintained throughout the main program loop. Nested loops are used to create the top and bottom margins and data area of the screen and must generate precisely timed horizontal sync throughout. Separate `delay` values are required for the transitions into, out of, and inside each loop. When programmed appropriately, timing is simplified to loading at each point the delay count equal to the fixed interval required minus the interval instruction execution time.
- Alternatively, loops can be unrolled at the expense of additional memory. Timing is to the straight in-line case.
- Timing is also simplified by keeping duplicate timing code arranged with the same timing at each occurrence. In the video example, the horizontal sync pulse (three instructions) is always kept within a single instruction group, thus creating a fixed timing element.
- Rearranging the sequence of instructions, where the sequence is not critical, can assist in creating the correct program timing. For instance, a register load for a loop, delay, or xfer value can occur anywhere preceding the instruction. Refresh instructions also can generally occur at any convenient location, so long as the overall rate is maintained.
- Often, timing is not required to be absolutely precise, nor absolutely consistent. Tolerances make coding easier. For instance, a 40KHz audio stream could probably be played consistently, or randomly, plus or minus one microsecond and the variations not be audible to the listener.

A code example of a typical refresh routine is given in Table 43, and example video code is included with the Patriot software development tools..

Address Space, Memory and Device Addressing

The VPU uses the same 32-bit address space as the MPU, but has its own program counter and executes independently and concurrently. I/O devices addressed during the execution of `xfer` are within the same address space. `xfer` bus transactions are identical to I/O-channel bus transactions except for how they are initiated. See *Direct Memory Access Controller,* page 103.

Interrupts

The VPU can request any of the eight MPU interrupts by executing `int`. The VPU can also request an MPU interrupt by accessing the last location in a 1024-byte memory page during the execution of `xfer`. `xfer` transfer interrupts and I/O-channel transfer interrupts are identical. See *Direct Memory Access Controller,* page 103, for more information. The MPU can respond to interrupt requests when the VPU next executes `delay`.

Bus Transactions

VPU instruction-fetch bus transactions are identical to MPU memory-read bus transactions. `xfer` bus transactions are identical to DMA bus transactions except for how they are initiated. See *Bus Operations*, page 157.

Bit Inputs and Bit Outputs

The bit inputs in `ioin` are accessed by the VPU with `tskipz`. This instruction tests an input bit, consumes it, and conditionally skips the remainder of the instruction group. This allows for polled device transfers or complex device-transfer sequences rather than the simple asynchronous transfers available with the DMAC. See *Bit Inputs*, page 111. Note that since `tskipz` causes conditional execution, care must be taken when designing program code that contains `tskipz` if deterministic execution is expected.

The bit outputs in `ioout` can be individually set or cleared by the VPU with `outt` and `outf`. They can be used to activate external events, generate synchronization pulses, etc. See *Bit Outputs*, page 115.

VPU Hardware and Software Reset

After hardware reset, the VPU begins executing at address 0x80000004, before the MPU begins execution. The VPU can then perform the RAS cycles required to initialize DRAM, and begin a program loop to maintain DRAM refresh, before executing `delay` to allow the MPU to configure the system.

Once the MPU has configured the system, the VPU typically is required to begin execution of its application program code. The VPU power-on-reset address selects the boot memory device, usually because A31 is set and other high address bits are zero. To clear A31 and thus begin execution in non-boot memory, a software reset must be issued by the MPU. See Table 43, page 94. The software reset is the only way to clear A31. The software reset can also be used in other instances to cause the VPU to begin execution of a new program. See *Processor Startup*, page 181.

Instruction Reference

The following text contains a description of each of the VPU instructions. In addition to a functional description, at the right margin is the instruction opcode and the number of CPU-clock cycles required to execute. See *Execution Timing*, page 92.

MNEMONIC OPCODE

delay

```
delay gi                                        0101 0xxx
                                                   5i hex
                                          2+gi CPU-clocks
```

Load vpudelay from g*i* (global register *i*, g1–g7) and wait the specified number of CPU-clock cycles, allowing bus access for DMA and the MPU. g*i* is unchanged. vpudelay counts down once each CPU-clock cycle. After vpudelay reaches zero, the VPU instruction after delay executes. Note that instruction decode and termination requires two CPU-clock cycle for a total execution time of 2+g*i* CPU-clock cycles. Within the opcode 0101 0xxx binary, xxx is the register number (1–7).

DMA and MPU bus transactions are granted bus access only when vpudelay indicates that sufficient time remains for the complete bus transaction to occur. The first VPU memory access to each memory group after delay executes is forced to be a RAS cycle so that VPU execution timing is deterministic. See Table 53, page 126.

dskipz

Decrement and Skip if Zero

```
dskipz gi                                       0110 1xxx
                                                   6i hex
                                  (not zero) 1 CPU-clock
                                      (zero)M CPU-clocks
```

Decrement g*i* (global register *i*, g8–g15). If g*i* is zero, then skip the remainder of the instruction group and continue execution with the next instruction group; otherwise, continue execution with the next instruction. Primarily used to create program loops by following dskipz with jump. Loops can be nested by using a different global register for each level of loop counter. Within the opcode 0110 xxxx binary, xxxx is the register number (8–15).

int

Set Interrupt

```
int n                                           1001 0xxx
                                                   9n hex
                                              1 CPU-clock
```

Set bit *n* of ioip to request interrupt *n*. Used to notify the MPU that an event has occurred. Within the opcode 1001 0xxx binary, xxx is the input bit number (0–7).

`jump`

`jump` *`destination`*

```
0011 xxxx
   3? hex
M CPU-clocks
```

Transfer execution to the page-relative, cell-aligned *destination*. The bits of *destination* replace the same cell-address bits within the current VPU `pc`. The number of bits within *destination* depends on the position of `jump` within the current instruction group. See page 92. Note that because of how `jump` functions, it cannot change A30 or A31. A VPU software reset from the MPU is used to clear A31 after power-on reset. See *VPU Power-on and Software Reset*, page 94.

`ld`

Load Register

`ld #`*`value`*`, g`*`i`*

```
0010 xxxx
   2n hex
M CPU-clocks
```

Load `g`*i* (global register *i*, `g1`–`g15`) with the 32-bit constant *value*. Used to load values for `xfer`, `mloop`, `dskipz` and `delay`, or to communicate with the MPU. Within the opcode 0010 xxxx binary, xxxx is the register number (1–15).

`mloop`

Micro-Loop on Register

`mloop g`*`i`*

```
0111 1xxx
   7i hex
1 CPU-clock
```

Decrement `g`*i* (global register *i*, `g8`–`g15`). If `g`*i* is non-zero, transfer execution to the beginning of the instruction group. If `g`*i* is zero, continue execution with the instruction following `mloop`. Used to loop on sequences of up to three other instructions without requiring the re-fetching of the instructions from memory. Within the opcode 0111 xxxx binary, xxxx is the register number (8–15).

nop

No Operation

```
nop                                                    1111 0000
                                                          F0 hex
                                                     1 CPU-clock
```

Do nothing. Used to waste time or as a placeholder for an instruction to be later placed.

outf

Set Bit Output False

```
outf n                                                 1011 0xxx
                                                          Bn hex
                                                     1 CPU-clock
```

Clear bit output *n*. Within the opcode 1011 0xxx binary, xxx is the bit number (0–7).

outt

Set Bit Output True

```
outt n                                                 1010 0xxx
                                                          An hex
                                                     1 CPU-clock
```

Set bit output *n*. Within the opcode 1010 0xxx binary, xxx is the bit number (0–7).

refresh

```
refresh                                                    0001 0000
                                                              10 hex
                                                        M CPU-clocks
```

Perform a RAS-only memory refresh cycle simultaneously on all memory groups so enabled. `msrra`, `msrha`, and `msra31` are used as the RAS refresh address. `msrra` is incremented. `msrtg` specifies the memory group whose RAS cycle timing is used for the refresh cycle. See Figure 44, page 151. `mgXrd` enables or disables refresh on each memory group. See Figure 33, page 139. VPU program code must be written to include `refresh` at intervals adequate for any DRAM used. The first VPU memory access to each memory group after `refresh` executes is forced to be a RAS cycle so that VPU execution timing is deterministic. See Table 53, page 126.

tskipz

Test Bit Input and Skip if Zero

```
tskipz n                                                   1000 0xxx
                                                              8n hex
                                               (not zero) 1 CPU-clock
                                                  (zero) M CPU-clocks
```

If bit input *n* is zero, then consume the input and skip the remainder of the instruction group and continue execution with the next instruction group; otherwise, continue execution with the next instruction. Used to cause the VPU code to operate conditionally on bit inputs. See *Bit Inputs*, page 111. Within the opcode 1000 0xxx binary, xxx is the input bit number (0–7).

MNEMONIC OPCODE

xfer

Transfer Data

xfer g*i* 0000 1xxx

0*i* hex

M CPU-clocks

Cause an I/O-channel transfer to occur immediately using g*i*, (global register *i*, g8–g15). g*i* contains the device address, memory address, and control information. See Figure 16.If bit one of g*i* is zero, perform a write bus transaction; if it is one, perform a read bus transaction. Increment bits 2–15 of g*i*. If bits 2–15 of g*i* are zero and bit zero of g*i* is one, then assert interrupt request *i*−8. Within the opcode 0000 xxxx binary, xxxx is the register number (8–15).

The type of bus transaction performed depends on whether the memory group involved is cell-wide or byte-wide (see Figure 34, page 140) and on the device transfer type (see Figure 46 and Figure 47, page 152). xfer bus transactions are identical to DMA bus transactions except for how they are initiated. See *Direct Memory Access Controller*, page 103.

ADVANCE INFORMATION

Table 44. VPU Mnemonics and Opcodes (Mnemonic Order)

Mnemonic	Opcode	Mnemonic	Opcode	Mnemonic	Opcode	Mnemonic	Opcode
delay g1	51	int 6	96	mloop g11	7b	outt 7	a7
delay g2	52	int 7	97	mloop g12	7c	refresh	10
delay g3	53	jump *dest*	30…	mloop g13	7d	tskipz 0	80
delay g4	54	ld #, g1	21	mloop g14	7e	tskipz 1	81
delay g5	55	ld #, g2	22	mloop g15	7f	tskipz 2	82
delay g6	56	ld #, g3	23	nop	f0	tskipz 3	83
delay g7	57	ld #, g4	24	outf 0	b0	tskipz 4	84
dskipz g8	68	ld #, g5	25	outf 1	b1	tskipz 5	85
dskipz g9	69	ld #, g6	26	outf 2	b2	tskipz 6	86
dskipz g10	6a	ld #, g7	27	outf 3	b3	tskipz 7	87
dskipz g11	6b	ld #, g8	28	outf 4	b4	xfer g8	08
dskipz g12	6c	ld #, g9	29	outf 5	b5	xfer g9	09
dskipz g13	6d	ld #, g10	2a	outf 6	b6	xfer g10	0a
dskipz g14	6e	ld #, g11	2b	outf 7	b7	xfer g11	0b
dskipz g15	6f	ld #, g12	2c	outt 0	a0	xfer g12	0c
int 0	90	ld #, g13	2d	outt 1	a1	xfer g13	0d
int 1	91	ld #, g14	2e	outt 2	a2	xfer g14	0e
int 2	92	ld #, g15	2f	outt 3	a3	xfer g15	0f
int 3	93	mloop g8	78	outt 4	a4		
int 4	94	mloop g9	79	outt 5	a5		
int 5	95	mloop g10	7a	outt 6	a6		

ADVANCE INFORMATION

Table 45. VPU Mnemonics and Opcodes (Opcode Order)

Opcode	Mnemonic	Opcode	Mnemonic	Opcode	Mnemonic	Opcode	Mnemonic
00…07		2e	ld #, g14	7b	mloop g11	a1	outt 1
08	xfer g8	2f	ld #, g15	7c	mloop g12	a2	outt 2
09	xfer g9	30	jump *dest*	7d	mloop g13	a3	outt 3
0a	xfer g10	40…50		7e	mloop g14	a4	outt 4
0b	xfer g11	51	delay g1	7f	mloop g15	a5	outt 5
0c	xfer g12	52	delay g2	80	tskipz 0	a6	outt 6
0d	xfer g13	53	delay g3	81	tskipz 1	a7	outt 7
0e	xfer g14	54	delay g4	82	tskipz 2	a8…af	
0f	xfer g15	55	delay g5	83	tskipz 3	b0	outf 0
10	refresh	56	delay g6	84	tskipz 4	b1	outf 1
11…20		57	delay g7	85	tskipz 5	b2	outf 2
21	ld #, g1	58…67		86	tskipz 6	b3	outf 3
22	ld #, g2	68	dskipz g8	87	tskipz 7	b4	outf 4
23	ld #, g3	69	dskipz g9	88…8f		b5	outf 5
24	ld #, g4	6a	dskipz g10	90	int 0	b6	outf 6
25	ld #, g5	6b	dskipz g11	91	int 1	b7	outf 7
26	ld #, g6	6c	dskipz g12	92	int 2	b8…ef	
27	ld #, g7	6d	dskipz g13	93	int 3	f0	nop
28	ld #, g8	6e	dskipz g14	94	int 4	f1…ff	
29	ld #, g9	6f	dskipz g15	95	int 5		
2a	ld #, g10	70…77		96	int 6		
2b	ld #, g11	78	mloop g8	97	int 7		
2c	ld #, g12	79	mloop g9	98…9f			
2d	ld #, g13	7a	mloop g10	a0	outt 0		

Direct Memory Access Controller

A Direct Memory Access Controller (DMAC) allows I/O devices to transfer data to and from system memory without the intervention of the MPU. The DMAC supports eight I/O channels prioritized from eight separate sources. Direct memory access (DMA) requests are received from the bit inputs through `ioin`. DMA and MPU bus request priorities are either fixed, which allows higher-priority requests to block lower-priority requests, or revolving, which prevents higher-priority requests that cannot be satisfied from blocking lower-priority requests.

DMA is supported for both cell-wide and byte-wide devices in both cell-wide and byte-wide memory. Each I/O channel can be individually configured as to the type of device and bus timing requirements. Byte-wide devices transfer data on `AD[7:0]` and can be configured as either one-byte byte-transfer or four-byte byte-transfer devices. Transfers are flybys or are buffered, as required for the I/O-channel bus transaction. See Table 57, page 158. DMAC and VPU `xfer` transfers are identical except for how they are initiated. DMAC transfers occur from asynchronous requests whereas `xfer` transfers occur at their programmed time.

Resources

The DMAC consists of several registers and associated control logic. DMA request zero, which corresponds to bit zero of the registers, has the highest priority; DMA request seven, which corresponds to bit seven of the registers, has the lowest priority. The DMAC and related registers include:

- Bit input register, `ioin`: bit inputs configured as DMA or interrupt requests, or general bit inputs. See Figure 26, page 131.
- Interrupt enable register, `ioie`: indicates which `ioin` bits are to be recognized as interrupt requests. See Figure 30, page 135.
- DMA enable register, `iodmae`: indicates which `ioin` bits are to be recognized as DMA requests. If DMA is enabled on an `ioin` bit, interrupt enable by `ioie` on that bit is ignored. See Figure 31, page 136.
- DMA enable expiration register, `iodmaex`:

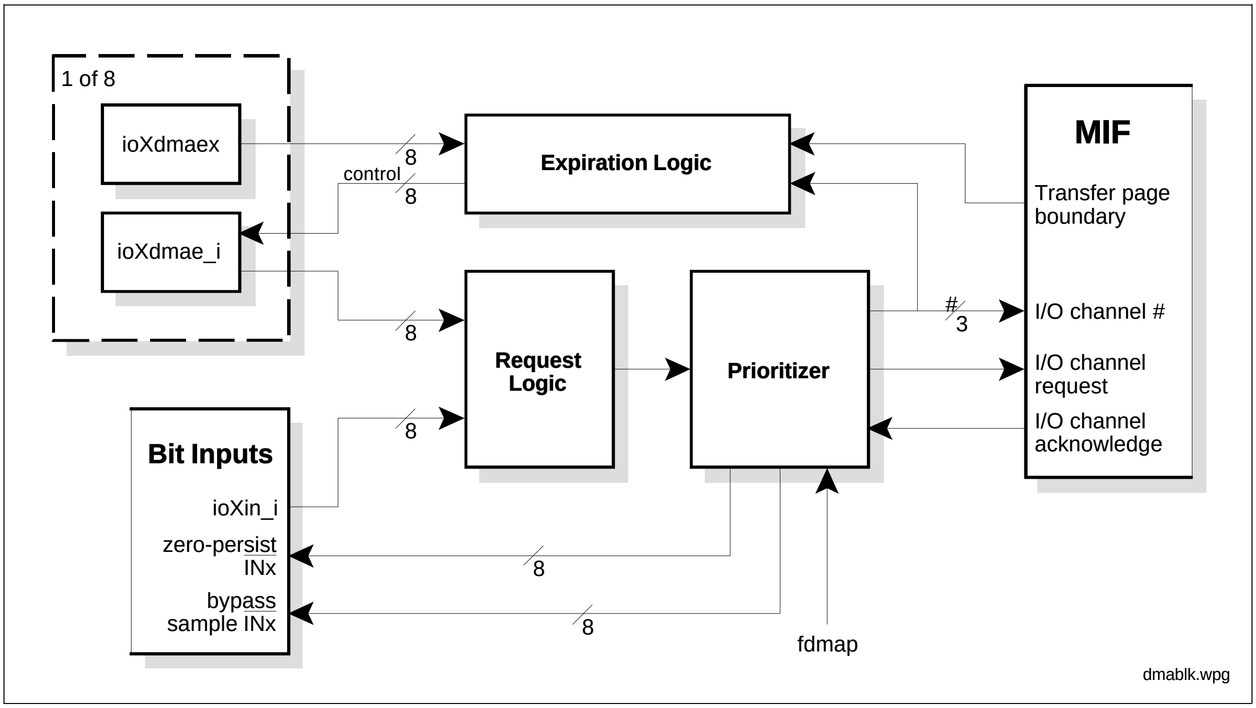


Figure 15. DMAC Block Diagram

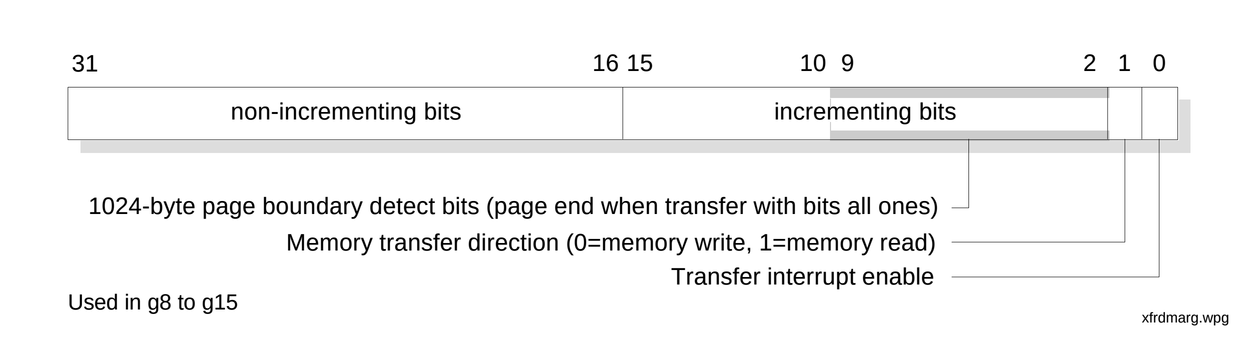

31
16 15
10 9
2 1 0
non-incrementing bits
incrementing bits
1024-byte page boundary detect bits (page end when transfer with bits all ones)
Memory transfer direction (0=memory write, 1=memory read)
Transfer interrupt enable
Used in g8 to g15
xfrdmarg.wpg

Figure 16. I/O-Channel Transfer Data Format

ADVANCE INFORMATION

indicates which `iodmae` bits are cleared following a DMA transfer involving the last location in a 1024-byte memory page occurring on that channel. See Figure 49, page 153.

- Global registers `g8` through `g15`: contain I/O-channel transfer specifications. Transfer specifications consist of device and memory transfer addresses and control bits. See Figure 16, page 104.
- Fixed DMA priorities bit, `fdmap`, in register miscellaneous B, `miscb`: prevents or allows lower-priority bus requests to contend for access to the bus if a higher-priority request cannot be satisfied (i.e., the available bus transaction slot is too small). See Figure 34, page 140.

DMA Requests

An `ioin` bit is configured as a DMA request source when the corresponding `iodmae` bit is set and the corresponding `ioie` bit is clear (though `ioie` is ignored when `iodmae` is set). Once a zero reaches `ioin`, it is available to request a DMA I/O-channel transfer. See *DMA Usage*, page 112. A DMA request is forced in software by clearing the corresponding `ioin` bit. Individually disabling DMA operations on an I/O channel by clearing its `iodmae` bit prevents a corresponding zero bit in `ioin` from being recognized as a DMA request, but does not affect the zero-persistence of the corresponding bit in `ioin`.

Prioritization

A DMA request is prioritized with other pending DMA requests, and, if the request has the highest priority or is the next request in revolving-priority sequence (see below), its corresponding I/O channel is the next to request the bus. DMA request prioritization requires one CPU-clock cycle to complete. When the I/O channel bus request is made, the MIF waits until the current bus transaction, if any, is almost complete. It then checks `vpudelay` to determine if the available bus slot is large enough for the required I/O channel bus transaction. If the bus slot is large enough, the bus is granted to the I/O channel, and the bus transaction begins.

The VPU always seizes the bus when `vpudelay` decrements to zero. Otherwise, a DMA I/O channel bus request and an MPU bus request contend for the bus, with the DMA I/O channel bus request having higher priority.

If `fdmap` is set and the bus slot is too small, the DMA I/O channel does not get the bus. Until a higher-priority DMA I/O channel request is made that fits the shrinking available bus slot, no bus transactions occur until the VPU seizes the bus. When the VPU next executes `delay`, the highest-priority DMA request, or the MPU if there are no DMA requests, repeats the bus request process.

If `fdmap` is clear and the bus slot is too small, the DMA I/O channel does not get the bus. The next lower-priority bus request is then allowed to request the bus, with the MPU as the lowest-priority request. The process repeats until the bus is granted or the VPU seizes the bus. When the VPU next executes `delay`, the highest-priority DMA request, or the MPU if there are no DMA requests, repeats the bus request process.

Memory and Device Addressing

Addresses used for I/O channel transfers contain both the I/O device address and the memory address. By convention, the uppermost address bits (when A31 is set) select I/O device addresses, while the lower address bits select the memory source/destination for the transfer. Multi-cycle transfer operations (e.g., transferring between a byte device and cell memory) assume A31 is part of the external I/O-device address decode and pass/clear A31 to select/deselect the I/O device as required during the bus transaction. See *I/O Addressing*, page 158, and *I/O-Channel Transfers*, page 159.

1024-byte memory page boundaries have special significance to I/O channel transfers. When each I/O-channel bus transaction completes, bits 15–2 of the memory address in the global register are incremented. The new address is evaluated to determine if the last location in a 1024-byte memory page was just transferred (by detecting that bits 9–2 are now zero). When the last location in a 1024-byte memory page was just transferred, an MPU interrupt can be requested or DMA can be disabled. See *Interrupts* and *Terminating DMA I/O-Channel Transfers*, below.

Interrupts

An MPU interrupt can be requested after an I/O channel transfer accesses the last location in a 1024-byte memory page. The interrupt requested is the same as the I/O-channel number, and occurs if interrupts are enabled on that channel (i.e., if bit zero of the corresponding global register is set). See Figure 16, and *Interrupt Controller*, page 107. This allows, for example, the MPU to be notified that a transfer has completed (by aligning the end of a transfer memory area with the end of a 1024-byte memory page), or to inform the MPU of progress during long transfers.

Note that for the interrupt to be serviced, the MPU must obtain the bus for sufficient time to execute the ISR. If the VPU does not execute delay, or continuous DMA transfers occur, the MPU will be unable to get the bus.

Bus Transaction Types

The type of bus transaction performed with an I/O device depends on whether the memory group involved is cell-wide or byte-wide and the whether the device is a one-byte byte-transfer, four-byte byte-transfer, or one-cell cell-transfer device. See *I/O-Channel Transfers*, page 159.

Device Access Timing

Any I/O device accessed during an I/O-channel transfer must complete the transfer by the end of the programmed bus cycle. Wait states are not available. Since I/O devices generally have longer access times than memory, during an I/O-channel bus cycle the programmed bus timing for the accessed memory group is modified by substituting ioXebt for the corresponding value in mgXebt. Note that ioXebt must be adequate both for the I/O device and for any memory group involved in the transfer. See *Programmable Memory Interface*, page 117.

Maximum Bandwidth Transfers

When the external input source for ioin is $\overline{\mathtt{IN}}$[7:0], maximum-bandwidth, back-to-back DMA transfers are possible. To achieve this, at the end of the DMA bus transaction an internal circuit bypasses the input sampling circuitry to check the DMA request bit directly on $\overline{\mathtt{IN}}$[7:0]; if the signal is low and no higher-priority requests are pending, another DMA bus request occurs immediately without the usual sampling and prioritization delays. This requires that the external DMA hardware ensure the bit is valid at this time. See Figure 80, page 217. If the remaining bus slot is large enough, the DMA bus request is granted, and the transfer starts immediately. To terminate back-to-back DMA bus transactions, the DMA request input must go high before the end of the current DMA bus transaction, or the corresponding DMA enable bit must be cleared. See *Terminating DMA I/O-Channel Transfers*, below. The maximum possible transfer rate is four bytes every two CPU-clock cycles. For example, with a 50-MHz 1X clock, the maximum transfer rate is 200 MB/second.

Terminating DMA I/O-Channel Transfers

DMA I/O channel bus transactions occur on an I/O channel while DMA remains enabled and DMA requests are received. To limit DMA transfers to a specified number of transactions:

- program the DMA transfer address so that the last data transfer desired occurs using the last location in a 1024-byte memory page, and
- set the corresponding `iodmaex` bit.

When the above transaction completes, the DMA enable bit in `iodmae` is cleared. If the transfer interrupt is enabled in the global register for the corresponding I/O channel, a corresponding MPU interrupt is also requested.

If more than 1024 bytes are to be transferred, enable the transfer interrupt for the I/O channel in the corresponding global register. Program the interrupt service routine to check the global register for the next-to-last 1024-byte page, and, at that time, set the corresponding `iodmaex` bit. When the last location in the next 1024-byte page is transferred, the corresponding bit in `iodmae` is cleared, disabling DMA on that channel. Note that this assumes the bus is available to the MPU to execute the ISR during the DMA transfers.

Other Capabilities

The DMAC can also be used to count events, and to interrupt the MPU when a given count is reached. To do this, events are designed to produce a normal DMA memory read request, and the resulting transfer cycle increments the “address” in the corresponding global register. This “address” becomes the event counter. The MPU can also examine the register at any time to determine how many events have occurred. To interrupt the MPU after a given event count, program the global register for a negative count value within bits 9–2, and enable the page-boundary interrupt. The MPU is interrupted when the counter reaches zero.

Interrupt Controller

An interrupt controller (INTC) allows multiple external or internal requests to gain, in an orderly and prioritized manner, the attention of the MPU. The INTC supports up to eight prioritized interrupt requests from twenty-four sources. Interrupts are received from the bit inputs through `ioin`, from I/O-channel transfers, or from the VPU interrupt instruction `int`.

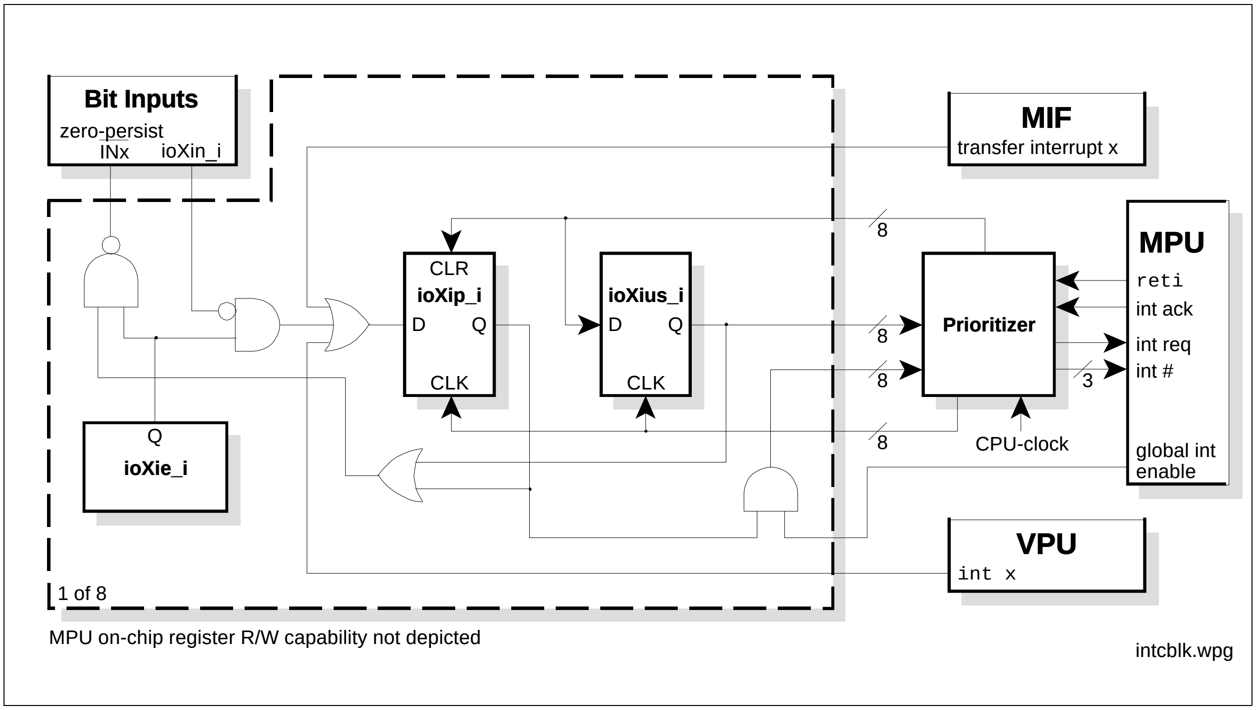
Resources

The INTC consists of several registers and associated control logic. Interrupt zero, which corresponds to bit zero of the registers, has the highest priority; interrupt seven, which corresponds to bit seven of the registers, has the lowest priority. The INTC and related registers include:

- Bit input register, `ioin`: bit inputs configured as DMA or interrupt requests, or general bit inputs. See Figure 26, page 131.
- Interrupt enable register, `ioie`: indicates which `ioin` bits are to be recognized as interrupt requests. See Figure 30, page 135.
- Interrupt pending register, `ioip`: indicates which interrupts have been recognized, but are waiting to be prioritized and serviced. See Figure 27, page 132.
- Interrupt under service register, `ioius`: indicates which interrupts are currently being serviced. See Figure 28, page 133.
- Global registers `g8` through `g15`: contain I/O-channel transfer specifications. Transfer specifications consist of device and memory transfer addresses and control bits. Bit zero enables interrupts during I/O-channel transfers on the corresponding channel. See Figure 16, page 104.
- DMA enable register, `iodmae`: indicates which `ioin` bits are to be recognized as DMA requests. If DMA is enabled on an `ioin` bit, interrupt enable by `ioie` on that bit is ignored. See Figure 31, page 136.

Table 46. Sources of Interrupts

Interrupt #	Interrupt Source
X	`ioin` bit X I/O channel X (register `g(8+X)`) VPU instruction `int` X



MPU on-chip register R/W capability not depicted

Figure 17. INTC Block Diagram

Operation

Each interrupt request is shared by three sources. A request can arrive from a zero bit in ioin (typically from an external input low), from an I/O-channel transfer interrupt, or from the VPU instruction int. Interrupt request zero comes from ioin bit zero, I/O channel zero (using g8), or int 0; interrupt request one comes from ioin bit one, I/O channel one (using g9), or int 1; the other interrupt requests are similarly assigned. See Table 46. Application usage typically designates only one source for an interrupt request, though this is not required.

Associated with each of the eight interrupt requests is an interrupt service routine (ISR) executable-code vector located in external memory. See Figure 5, page 16. A single ISR executable-code vector for a given interrupt request is used for all requests on that interrupt. It is programmed to contain executable code, typically a branch to the ISR. When more than one source is possible, the current source might be determined by examining associated bits in ioin, ioie, iodmae and the global registers.

Interrupt Request Servicing

When an interrupt request from any source occurs, the corresponding bit in ioip is set, and the interrupt request is now a *pending interrupt*. Pending interrupts are prioritized each CPU-clock cycle. The interrupt_en bit in mode holds the current global interrupt enable state. It can be set with the MPU enable-interrupt instruction, ei; cleared with the disable-interrupt instruction, di; or changed by modifying mode. Globally disabling interrupts allows all interrupt requests to reach ioip, but prevents the pending interrupts in ioip from being serviced.

When interrupts are enabled, interrupts are recognized by the MPU between instruction groups, just before the execution of the first instruction in the group. This allows short, atomic, uninterruptable instruction sequences to be written easily without having to save, restore, and manipulate the interrupt state. The stack architecture allows interrupt service routines to be executed without requiring registers to be explicitly saved, and the stack caches minimize the memory accesses required when making additional register resources available.

If interrupts are globally enabled and the highest-priority ioip bit has a higher priority than the highest-priority ioius bit, the highest-priority ioip bit is cleared, the corresponding ioius bit is set, and the MPU is interrupted just before the next execution of the first instruction in an instruction group. This nests the interrupt servicing, and the pending interrupt is now the current *interrupt under service*. The ioip bits are not considered for interrupt servicing while interrupts are globally disabled, or while none of the ioip bits has a higher priority than the highest-priority ioius bit.

Unless software modifies ioius, the current interrupt under service is represented by the highest-priority ioius bit currently set. reti is used at the end of ISRs to clear the highest-priority ioius bit that is set and to return to the interrupted program. If the interrupted program was a lower-priority interrupt service routine, this effectively "unnests" the interrupt servicing.

External Interrupts

An ioin bit is configured as an "external" interrupt request source if the corresponding ioie bit is set and the corresponding iodmae bit is clear. Once a zero reaches ioin, it is available to request an interrupt. An interrupt request is forced in software by clearing the corresponding ioin bit or by setting the corresponding ioip bit. Individually disabling an interrupt request by clearing its ioie bit prevents a corresponding zero bit in ioin from being recognized as an external interrupt request, but does not affect a corresponding interrupt request from another source.

While an interrupt request is being processed, until its ISR terminates by executing reti, the corresponding ioin bit is not zero-persistent and follows the sampled level of the external input pin. Specifically, for a given interrupt request, while its ioie bit is set, and its ioip bit or ioius bit is set, its ioin bit is not zero-persistent. This effect can be used to disable zero-persistent behavior on non-interrupting bits.

For waveforms, see Figure 82, page 219, and Figure 83, page 221.

I/O-Channel Transfer Interrupts

If an `ioin` bit is configured as a DMA request, or if that I/O channel is used by `xfer`, interrupt requests occur after a transfer involving the last location in a 1024-byte memory page, provided bit zero in the corresponding global register is set (i.e., transfer interrupts are enabled). The request occurs by the corresponding `ioip` bit being set, and is thus not disabled by clearing the corresponding `ioie` bit. See *Direct Memory Access Controller*, page 103, and *Virtual Peripheral Unit*, page 89.

VPU `int` Interrupts

The VPU can also directly request any of the eight available interrupts by executing `int`. The request occurs by the corresponding `ioip` bit being set, and is thus not disabled by clearing the corresponding `ioie` bit. The MPU is able to respond to the interrupt request when the VPU next executes `delay`. VPU interrupts are disabled by modifying the VPU instructions in memory to remove the instruction `int`.

ISR Processing

When an interrupt request is recognized by the MPU, a `call` to the corresponding ISR executable-code vector is performed, and interrupts are blocked until an instruction that begins in byte one of an instruction group is executed. To service an interrupt without being interrupted by a higher-priority interrupt:

- the ISR executable-code vector typically contains a four-byte branch, and
- the first instruction group of the interrupt service routine must globally disable interrupts.

See the code example in Table 47.

If interrupts are left globally enabled during ISR processing, a higher-priority interrupt can interrupt the MPU during processing of the current ISR. This allows devices with more immediate servicing requirements to be serviced promptly even when frequent interrupts at many priority levels are occurring.

Table 47. Code Example: ISR Vectors

```
; Interrupt Vectors

    .quad   4
    .text   vectors         ; org 0x100 set in linker

    br      int_0_ISR       ; highest-priority ISR
    br      int_1_ISR
    ...
    br      int_7_ISR       ; lowest-priority ISR
    ...

    .text   ISRs            ; org set in linker file

int_0_ISR::
    push    mode            ; save carry
    ; This ISR can't be interrupted because int 0
    ; has the highest priority.
    ...
    pop     mode            ; restore carry
    reti
    ...

int_A_ISR::
    push    mode            ; save carry
    ...
    ; This ISR can be interrupted by a higher
    ; priority interrupt.
    pop     mode
    reti
    ...

int_B_ISR::
    push    mode            ; save carry & ei state
    di
    ...
    ; Don't allow this ISR to be interrupted at all.
    ...
    ; ensure return before interrupts re-enabled
    .quad   2
    pop     mode
    reti
    ...

int_C_ISR::
    push    mode            ; save carry & ei state
    pop     lstack          ; place accessible
    di
    ; Don't allow this critical part of the ISR to be
    ; interrupted.
    ...
    push    r0
    pop     mode            ; restore ei state
    ...
    ; ISR can be interrupted by higher-priority
    ; interrupts now
    ...
    push    lstack
    pop     mode            ; restore carry
    reti
    ...
```

Note that there is a delay of one CPU-clock cycle between the execution of `ei`, `di`, or `pop mode` and the change in the global interrupt enable state taking effect. To ensure the global interrupt enable state change takes effect before byte zero of the next instruction group, the state-changing instruction must not be the last instruction in the current instruction group.

If the global interrupt enable state is to be changed by the ISR, the prior global interrupt enable state can be saved with `push mode` and restored with `pop mode` within the ISR. Usually a `pop mode`, `reti` sequence is placed in the same instruction group at the end of the ISR to ensure that `reti` is executed, and the local-register stack unnests, before another interrupt is serviced. Since the return address from an ISR is always to byte zero of an instruction group (because of the way interrupts are recognized), another interrupt can be serviced immediately after execution of `reti`. See the code example in Table 47.

As described above for processing ISR executable-code vectors, interrupt requests are similarly blocked during the execution of all traps. This allows software to prevent, for example, further data from being pushed on the local-register stack due to interrupts during the servicing of a local-register-stack overflow exception. When resolving concurrent trap and interrupt requests, interrupts have the lowest priority.

Bit Inputs

Eight external bit inputs are available in bit input register `ioin`. They are shared for use as interrupt requests, as DMA requests, as input to the VPU instruction `tskipz`, and as bit inputs for general use by the MPU. They are sampled externally from one of two sources determined by the state of `pkgio`.

Resources

The bit inputs consist of several registers, package pins, and associated input sampling circuitry. These resources include:

- Bit input register, `ioin`: bit inputs configured as DMA or interrupt requests, or general bit inputs. See Figure 26, page 131.
- Interrupt enable register, `ioie`: indicates which `ioin` bits are to be recognized as interrupt requests. See Figure 30, page 135.
- Interrupt pending register, `ioip`: indicates which interrupts have been recognized, but are waiting to be prioritized and serviced. See Figure 27, page 132.
- Interrupt under service register, `ioius`: indicates which interrupts are currently being serviced. See Figure 28, page 133.
- DMA enable register, `iodmae`: indicates which `ioin` bits are to be recognized as DMA requests for the corresponding I/O channels. If DMA is enabled on an `ioin` bit, interrupt enable by `ioie` on that bit is ignored. See Figure 31, page 136.
- Package I/O pins bit, `pkgio`, in register miscellaneous B, `miscb`: selects whether the bit inputs are sampled from the dedicated inputs $\overline{\mathtt{IN}}$`[7:0]` or multiplexed off AD`[7:0]`. See Figure 34, page 140.

Input Sources and Sampling

If `pkgio` is clear, the bit inputs are sampled from AD`[7:0]` while $\overline{\mathtt{RAS}}$ is low and $\overline{\mathtt{CAS}}$ is high. External

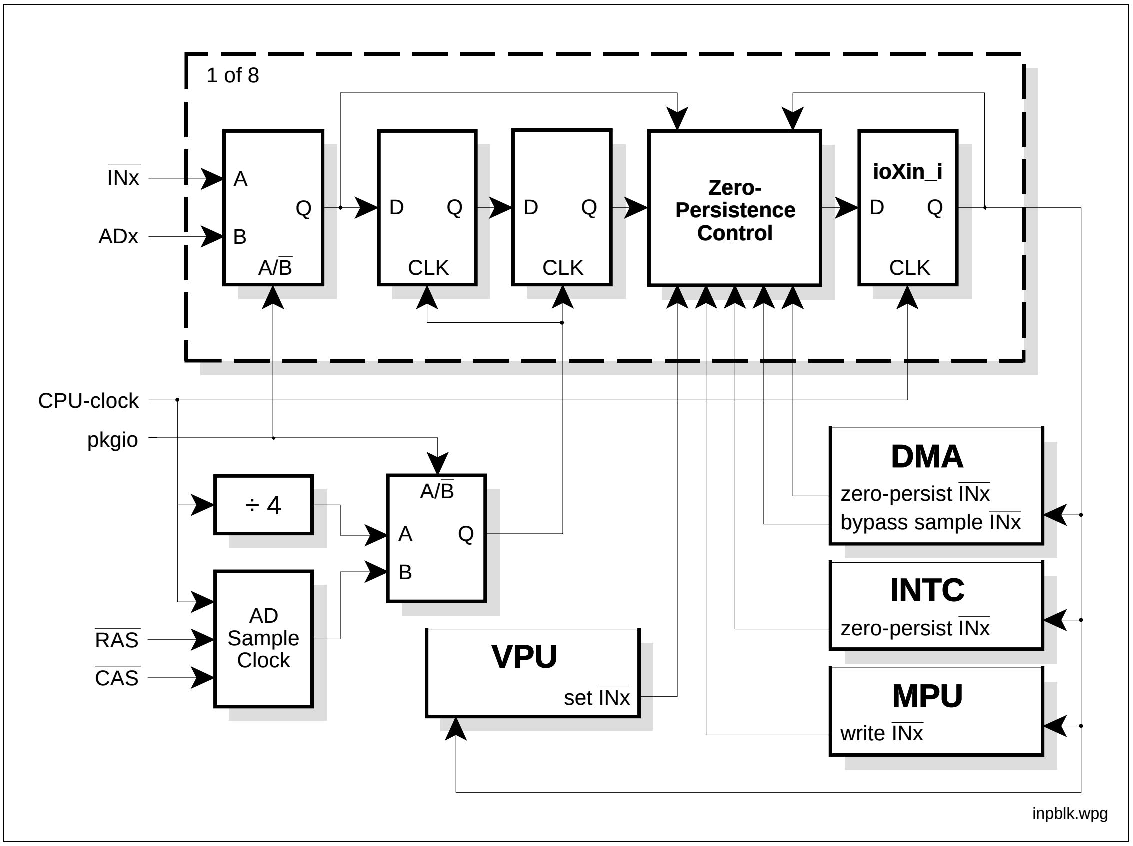


Figure 18. Bit Input Block Diagram

ADVANCE INFORMATION

hardware must place the bit inputs on `AD[7:0]` and remove them at the appropriate time. Using `AD[7:0]` for bit inputs can reduce PWB area and cost compared with using $\overline{\texttt{IN}}$`[7:0]`. `AD[7:0]` are sampled for input:

- while $\overline{\texttt{CAS}}$ is high, four CPU-clock cycles after $\overline{\texttt{RAS}}$ transitions low,
- every four CPU-clock cycles while $\overline{\texttt{CAS}}$ remains high,
- immediately before $\overline{\texttt{CAS}}$ transitions low if at least four CPU-clock cycles have elapsed since the last sample, and
- four CPU-clock cycles after $\overline{\texttt{CAS}}$ transitions high, provided $\overline{\texttt{CAS}}$ is still high.

This ensures:

- time for external hardware to place data on the bus before sampling,
- continuous sampling while $\overline{\texttt{CAS}}$ is high, and
- at least one sample every $\overline{\texttt{CAS}}$ bus cycle when four CPU-clocks have elapsed since the last sample.

To ensure sampling in a given state, an input bit must be valid at the designated sample times or remain low for a worst-case sample interval, which, as described above, depends on the programmed bus timing and activity. See Figure 83, page 221, for waveforms.

If `pkgio` is set, the bit inputs are sampled from $\overline{\texttt{IN}}$`[7:0]` every four CPU-clock cycles. To ensure sampling in a given state, a bit input must be valid for just more than four CPU-clock cycles. See Figure 82, page 219, for waveforms.

All asynchronously sampled signals are susceptible to metastable conditions. To reduce the possibility of metastable conditions resulting from the sampling of the bit inputs, they are held for four CPU-clock cycles to resolve to a valid logic level before being made available to `ioin` and thus for use within the CPU. The worst-case sampling delay for bit inputs taken from `AD[7:0]` to reach `ioin` depends on the bus cycle times. The worst-case sampling delay for bit inputs from $\overline{\texttt{IN}}$`[7:0]` to reach `ioin` is eight CPU-clock cycles. The sample delay causes bit-input consumers not to detect an external signal change for the specified period.

The bit inputs reaching `ioin` are normally zero-persistent. That is, once an `ioin` bit is zero, it stays zero regardless of the bit state at subsequent samplings until the bit is “consumed” and released, or is written with a one by the MPU. Zero-persistent bits have the advantage of both edge-sensitive and level-sensitive inputs, without the noise susceptibility and non-shareability of edge-sensitive inputs. Under certain conditions during DMA request servicing and `ioin` interrupt servicing, the `ioin` bits are not zero-persistent. See *DMA Usage* and *Interrupt Usage* below. An effect of the INTC can be used to disable zero-persistent behavior on the bits. See *General-Purpose Bits* below.

DMA Usage

An `ioin` bit is configured as a DMA request source when its corresponding `iodmae` bit is set. After the DMA bus transaction begins, the `ioin` bit is consumed.

When the external input source for `ioin` is $\overline{\texttt{IN}}$`[7:0]`, maximum-bandwidth back-to-back DMA transfers are possible. To achieve this, an internal circuit bypasses the sampling and zero-persistence circuitry to check the DMA request bit on $\overline{\texttt{IN}}$`[7:0]` at the end of the DMA bus transaction without the usual sampling and prioritization delays. See *Maximum Bandwidth Transfers*, page 105.

Interrupt Usage

An `ioin` bit is configured as an interrupt request source when the corresponding `ioie` bit is set and the corresponding `iodmae` bit is clear. While an interrupt request is being processed, until its ISR terminates by executing `reti`, the corresponding `ioin` bit is not zero-persistent and follows the sampled level of the external input. Specifically, for a given interrupt request, while its `ioie` bit is set, and its `ioip` bit or `ioius` bit is set, its `ioin` bit is not zero-persistent. This effect can be used to disable zero-persistent behavior on non-interrupting bits (see below).

Table 48. Code Example: Bit Input Without Zero-Persistence

```
; Disable zero-persistence for bit input 7
        push.n   #-1          ; true flag

        push.b   #io7ius_i
        sto.i    []           ; set under service bit

        push.b   #io7ie_i
        sto.i    []           ; enable interrupt
        pop                   ; discard flag
        ...
```

General-Purpose Bits

If an `ioin` bit is configured neither for interrupt requests nor for DMA requests, then it is a zero-persistent general-purpose `ioin` bit. Alternatively, by using an effect of the INTC, general-purpose `ioin` bits can be configured without zero-persistence. Any bits so configured should be the lowest-priority `ioin` bits to prevent blocking a lower-priority interrupt. They are configured by setting their `ioie` and `ioius` bits. The `ioius` bit prevents the `ioin` bit from zero-persisting and from being prioritized and causing an interrupt request. See the code example in Table 48.

VPU Usage

An `ioin` bit are used as input to `tskipz`. This instruction reads, tests, and consumes the bit. The `ioin` bits cannot be written by the VPU. General-purpose `ioin` bits are typically used for `tskipz`, but there are no hardware restrictions on usage.

MPU Usage

Bits in `ioin` are read and written by the MPU as a group with `ldo [ioin]` and `sto [ioin]`, or are read and written individually with `ldo.i [ioXin_i]` and `sto.i [ioXin_i]`. Writing zero bits to `ioin` has the same effect as though the external bit inputs had transitioned low for one sampling cycle, except that there is no sampling delay. This allows software to simulate events such as external interrupt or DMA requests. Writing one bits to `ioin`, unlike data from external inputs when the bits are zero-persistent, releases persisting zeros to accept the current sample. The written data is available immediately after the write completes. The MPU can read `ioin` at any time, without regard to the designations of the `ioin` bits, and with no effect on the state of the bits. The MPU does not consume the state of `ioin` bits during reads. See the code examples in Table 49.

Table 49. Code Example: MPU Usage of Bit Inputs

```
; Force service on bit 5 (Interrupt or DMA, as
; configured)

      push.n   #0          ; false flag

      push.n   #io5in_i
      sto.i    []          ; clear input bit
      pop                  ; discard flag
      ...

; Read last sampled state of zero-persistent bit
; inputs. (Assumes all bits are configured as
; zero-persistent).

      push.n   #-1         ; all ones for all bits

      push.n   #ioin
      sto  []              ; temporarily remove
                           ; persistence, latest
                           ; sample latches,
      pop                  ; discard -1

      push.n   #ioin
      ldo  []              ; get last sample
      ...
```

To perform a “real-time” external-bit-input read on zero-persistent bits, ones bits are written to the bits of interest in `ioin` before reading `ioin`. This releases any persisting zeros, latches the most recently resolved sample, and reads that value. Bits that are not configured as zero-persistent do not require this write. Note that any value read can be as much as two worst-case sample delays old. To read the values currently on the external inputs requires waiting two worst-case sample delays for the values to reach `ioin`. See the code example in Table 50.

Table 50. Code Example: MPU “Real-Time” Bit Input Read

```
; Read current state of zero-persistent input pins.
; (Assumes pkgio is set, and bits are zero-persistent)

; Assume we just tickled a device and we want to
; see if it just responded, but we have the bits
; configured as zero-persistent. The sample interval
; of four CPU-clock cycles and the sample holding
; delay of four CPU-clock cycles means there is a
; worst-case delay of eight CPU-clock cycles before
; the data is available in ioin. So...

    ; Put programming to tickle device here...

    nop                 ; wait the delay time
    nop
    nop
    nop
    nop
    nop                 ; 6 here, two below

; Read last sampled state of all zero-persistent
; bit inputs (Assumes all bits are configured as
; zero-persistent)

    push.n   #-1        ; all ones for all bits (7)

    push.n   #ioin      ; (CPU-clock cycle # 8)
                        ; ...data is now available
                        ; to ioin.
    sto  []             ; Temporarily remove
                        ; persistence, latest
                        ; sample latches,
    pop                 ; discard -1

    push.n   #ioin
    ldo  []             ; get last sample
    ...
```

Bit Outputs

Eight general-purpose bit outputs can be set high or low by either the MPU or the VPU. The bits are available in the bit output register, `ioout`.

Resources

The bit outputs consist of a register, package pins, and associated circuitry. These resources include:

- Bit output register, `ioout`: bits that were last written by either the MPU or the VPU. See Figure 29, page 134.
- Outputs, `OUT[7:0]`: the dedicated output pins.
- Address Data bus, `AD[7:0]`: multiplexed bit outputs on these pins while $\overline{\texttt{RAS}}$ is high.
- Output pin driver current bits, `outdrv`, in driver current register, `driver`: sets the drive capability of `OUT[7:0]`. See Figure 50, page 154.

Usage

The bits are read and written by the MPU as a group with `ldo [ioout]` and `sto [ioout]`, or are read and written individually with `ldo.i [ioXout_i]` and `sto.i [ioXout_i]`.

The bit outputs are written individually by the VPU with `outt` and `outf`. The bit outputs cannot be read by the VPU.

When written, the new values are available immediately after the write completes. Note that if both the MPU and VPU write the same bit during the same CPU-clock cycle, any one bit written prevails.

The bits are always available on `OUT[7:0]`, and on `AD[7:0]` when $\overline{\texttt{RAS}}$ is high. When sampled from `AD[7:0]`, external hardware is required to latch the bits when $\overline{\texttt{RAS}}$ falls. Note that (by definition) these bits are only updated when a RAS cycle occurs. Using `AD[7:0]` for output can reduce PWB area and cost compared to using `OUT[7:0]`. See Figure 81, page 218, for waveforms.

The drive capability of `OUT[7:0]` can be programmed in `driver`.

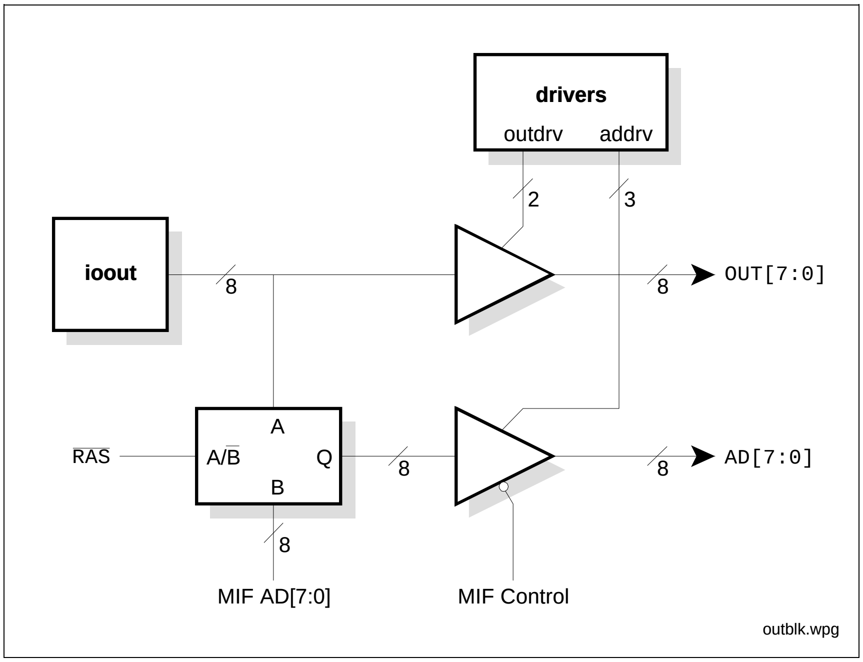


Figure 19. Bit Outputs Block Diagram

Programmable Memory Interface

The Programmable Memory Interface (MIF) allows the timing and behavior of the CPU bus interface to be adapted to the requirements of peripheral devices with minimal external logic, thus reducing system cost while maintaining performance. A variety of memory devices are supported, including EPROM, SRAM, DRAM and VRAM, as well as a variety of I/O devices. All operations on the bus are directed by the MIF. Most aspects of the bus interface are programmable, including address setup and hold times, data setup and hold times, output buffer enable and disable times, write enable activation times, memory cycle times, DRAM-type device address multiplexing, and when DRAM-type RAS cycles occur. Additional specifications are available for I/O devices, including data setup and hold times, output buffer enable and disable times, and device transfer type (one-byte, four-byte or one-cell).

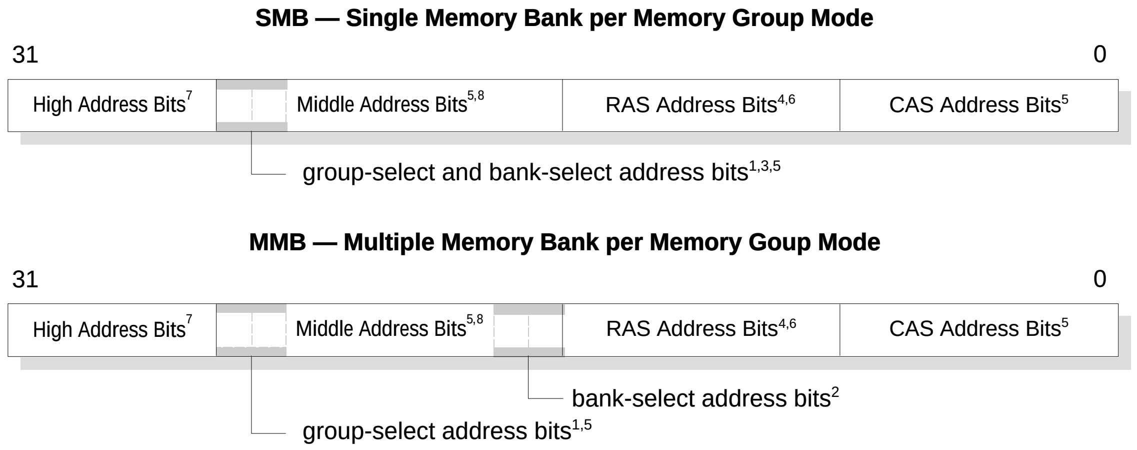
Resources

The MIF consists of several registers, package pins, and associated control logic. These resources include:

- VRAM control bit register, `vram`: controls $\overline{OE}$, $\overline{LWE}$, CASes, RASes, and DSF to initiate special VRAM operations. See Figure 32, page 137.
- Miscellaneous A register, `misca`: controls refresh and RAS-cycle generation. See Figure 33, page 139.
- Miscellaneous B register, `miscb`: selects each memory group data width (cell-wide or byte-wide), and the memory bank-select architecture. See Figure 34, page 140.
- Memory system group-select mask register, `msgsm`: indicates which address bits are decoded to select groups of memory devices. See Figure 37, page 143.
- Memory group device size register, `mgds`: indicates the size and configuration of memory devices for each memory group. See Figure 38, page 144.
- Miscellaneous C register, `miscc`: controls RAS-cycle generation and the location of bank-select address bits for SRAM memory groups. See Figure 39, page 145.
- Memory group X extended bus timing register, `mgXebt`: indicates memory-cycle expansion or extension values, which create longer data setup and hold times and output buffer enable and disable times for the memory devices in the corresponding memory group. See Figure 40, page 146.
- Memory group X CAS bus timing register, `mgXcasbt`: indicates the unexpanded and unextended address and data strobe activation times for the CAS portion of a bus cycle. See Figure 41, page 147.
- Memory group X RAS bus timing register, `mgXrasbt`: indicates the RAS precharge and address hold times to be prepended to the CAS part of a bus cycle to create a RAS cycle. See Figure 42, page 149.
- I/O channel X extended bus timing register, `ioXebt`: indicates memory cycle expansion or extension values, which create longer data setup and hold times and output buffer enable and disable times for the I/O device on the corresponding I/O channel. See Figure 43, page 150.
- Memory system refresh address, `msra`: indicates the row address to be used during the next DRAM refresh cycle. See Figure 44, page 151.
- I/O device transfer types A register, `iodtta`: indicates the type of transfer for each of I/O channels 0, 1, 2 and 3. See Figure 46, page 152.
- I/O device transfer types B register, `iodttb`: indicates the type of transfer for each of I/O channels 4, 5, 6 and 7. See Figure 47, page 152.
- Driver current register, `driver`: indicates the relative drive current of the various output drivers. See Figure 50, page 154.

Memory System Architecture

The MIF supports direct connection to a variety of memory and peripheral devices. The primary requirement is that the device access time be deterministic; wait states are not available because they create non-deterministic timing for the VPU. The MIF directly supports a wide range of sizes for multiplexed-address devices (DRAM, VRAM, etc.) up to 128 MB, as well as sizes for demultiplexed-address devices (SRAM, EPROM, etc.) up to 1 MB. Fast-page mode access and RAS-only refresh to DRAM-type devices are supported. SRAM-type devices appear to the MIF as DRAM with no RAS address bits and a large number of CAS address bits. See Figure 38, page 144.


SMB — Single Memory Bank per Memory Group Mode
31
0
High Address Bits[7]
Middle Address Bits[5,8]
RAS Address Bits[4,6]
CAS Address Bits[5]
group-select and bank-select address bits[1,3,5]
MMB — Multiple Memory Bank per Memory Goup Mode
31
0
High Address Bits[7]
Middle Address Bits[5,8]
RAS Address Bits[4,6]
CAS Address Bits[5]
bank-select address bits[2]
group-select address bits[1,5]

Notes

1. Located by bits in `msgsm`.
2. DRAM—2 bits immediately above the RAS address bits. SRAM—2 bits located by `mssbs` in `miscc`.
3. SRAM and DRAM.
4. DRAM only, field is zero length in SRAM.
5. Excluded from RAS-cycle determination, except for A31 (see note 7).
6. Included in RAS-cycle determination.
7. Optionally included in RAS-cycle determination.
8. If `msgsm` is zero, see text.

gsbsbits.wpg

Figure 20. Group-Select and Bank-Select Bit Locations

Address bits are multiplexed out of the CPU on AD`[31:9]` to reduce package pin count. DRAM-type devices collect the entire memory address in two pieces, referred to as the *row address* (upper address bits) and *column address* (lower address bits). Their associated bus cycles are referred to as *Row Address Strobe* (RAS) cycles and *Column Address Strobe* (CAS) cycles. With the exception of memory faults, refresh, and CAS-before-RAS VRAM cycles, a RAS cycle contains, enclosed within the $\overline{\mathtt{RAS}}$ active period, a CAS cycle. Thus, RAS cycles are longer than CAS cycles. While RAS cycles are not required for the operation of SRAM-type devices, RAS cycles can occur for several reasons which are discussed below.

Though I/O devices can be addressed like memory for access by the MPU, I/O-channel transfers require addressing an I/O device and a memory location simultaneously. This is achieved by splitting the available 32 address bits into two areas: the lower address bits, which address memory, and the higher address bits, which address I/O devices. The location

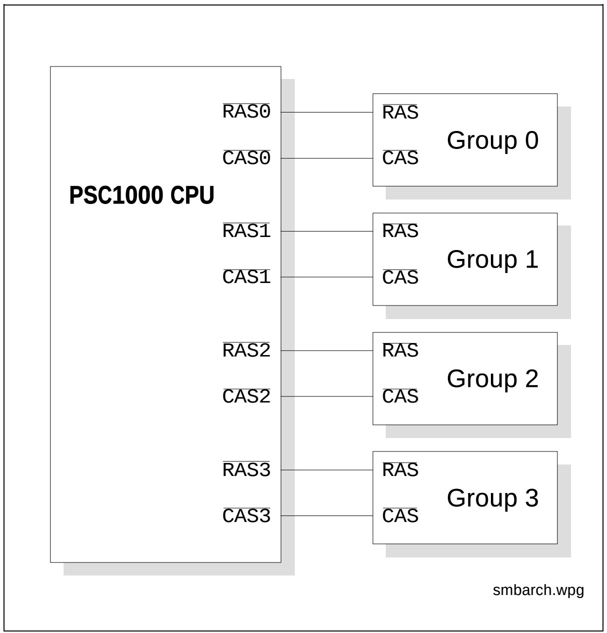


Figure 21. SMB Memory Architecture

of the split depends upon application requirements for the quantity of addressable memory and I/O devices installed. The areas can overlap, if required, with the side effect that an I/O device can only transfer data with a corresponding area of memory. These higher address bits are discussed below.

Memory Groups

The MIF operates up to four *memory groups*, maintaining for each the most recent RAS address bits and a unique configuration. Up to two address bits are decoded to determine the current group. The address bits for this function are set in the memory system

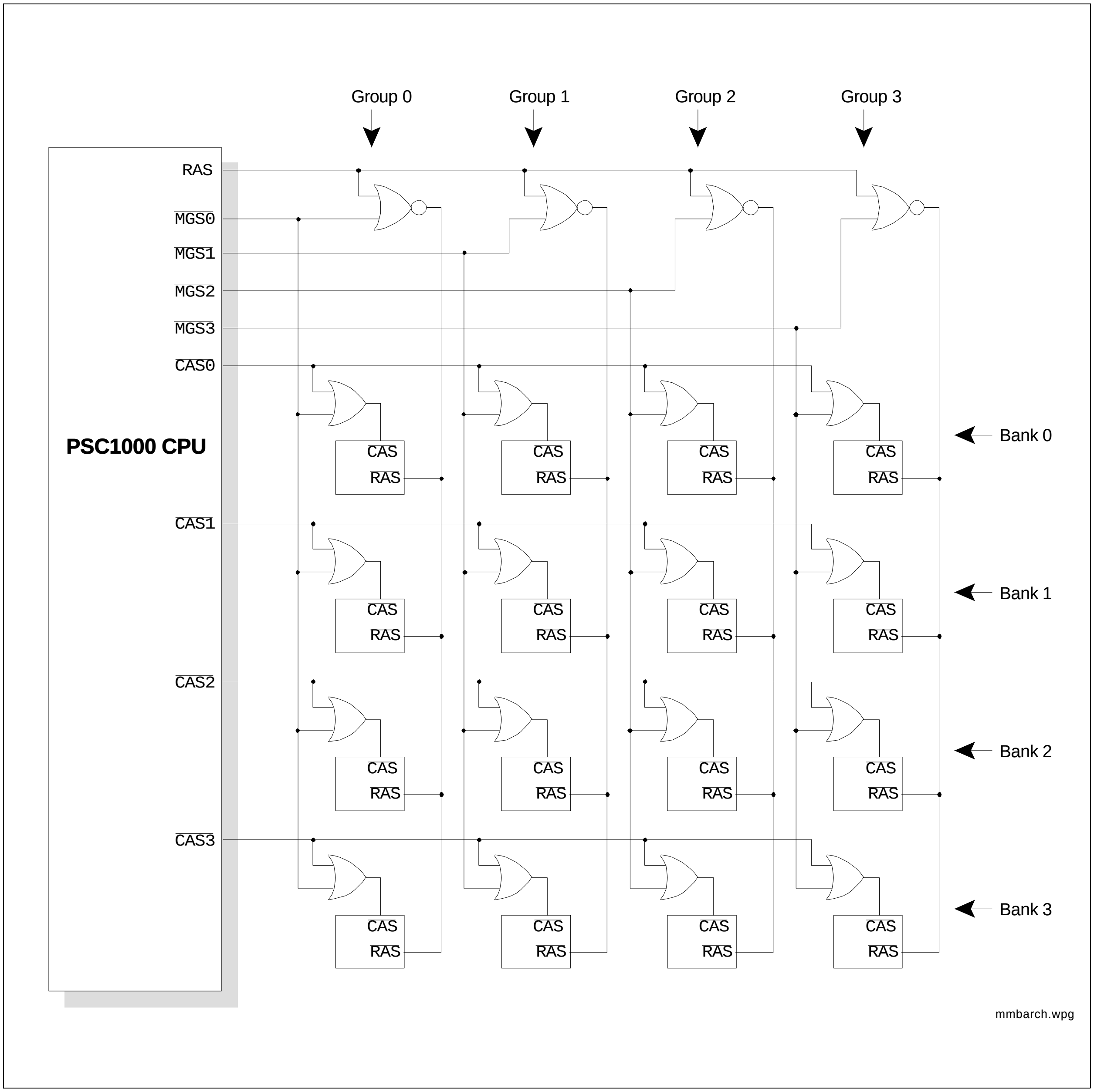


Figure 22. MMB Memory Architecture

ADVANCE INFORMATION

ADVANCE INFORMATION

group-select mask register, `msgsm`. Each memory group is programmed for device width, bus timing, and device size (which specifies how address bits are multiplexed onto AD`[31:9]`). Address bits below the group-select mask are typically used to address memory devices or portions of an I/O device, and bits above the group-select mask are typically used to address I/O devices.

Memory Banks

Each memory group can have one or more *memory banks*, which are selected in a manner dependent upon the bus interface mode. All memory banks within a memory group share the configuration and most recent RAS address of that group. Two address bits are decoded to determine the current memory bank.

In Single Memory Bank (SMB) mode (mmb = 0), `msgsm` sets the group-select and bank-select bits to be the same bits. This allows up to four groups at one bank per group, totaling four banks: group 0, bank 0; group 1, bank 1; group 2, bank 2; and group 3, bank 3. $\overline{\mathtt{MGSx}}$/$\overline{\mathtt{RASx}}$ output $\overline{\mathtt{RASx}}$ signals for direct connection to memory devices. See Figure 21.

In Multiple Memory Bank (MMB) mode (mmb = 1), depending on whether `msgsm` overlaps the bank-select bits, one, two or four banks can be selected in each group. This allows up to sixteen banks for all groups combined; more banks can be decoded by defining additional bank-select bits with external logic. The address bits that select the current memory bank either are located immediately above the row-address bits for DRAM devices (mgX`ds` values 0–0x0e), or are specified by the `mssbs` bits for all SRAM devices in the system (mgX`ds` value 0x0f). The group-select bits determine the $\overline{\mathtt{MGSx}}$/$\overline{\mathtt{RASx}}$ (which output the $\overline{\mathtt{MGSx}}$ signal), and the bank-select bits determine the $\overline{\mathtt{CASx}}$ that activates in any given bus cycle. See Figure 20. Gating the four $\overline{\mathtt{MGSx}}$ signals with the four $\overline{\mathtt{CASx}}$ signals creates up to sixteen memory bank selects. See Figure 22.

A hybrid of the two modes can also be programmed by selecting MMB mode and placing the `msgsm` bits overlapping the banks bits. This allows using $\overline{\mathtt{MGSx}}$ directly as a faster chip select for SRAM-type devices than $\overline{\mathtt{CASx}}$ is in SMB mode. For DRAM-type devices, the $\overline{\mathtt{CASx}}$ strobes can be connected directly to the memory device and only one NOR gate per group is required to create the RAS for that group.

Device Requirements Programming

Each memory group can be programmed with a unique configuration of device width, device size, and bus timing. After a CPU reset, the system operates in byte-wide mode, with the slowest possible bus timing, and executes from memory group zero, typically from an external PROM. See *Processor Startup*, page 181. Usually, the program code in the PROM initially executes code to determine and set the proper configurations for the memory groups, I/O devices, and other requirements of the system.

Device Sizes

Memory device sizes are programmed to one of sixteen settings in `mgds`. Most currently available and soon to be available DRAM-type device sizes can be selected, as well as an SRAM-type option. The selection of the device size and width determines the arrangement of the address bits on AD`[31:9]`. See Table 51, page 122, and Table 52, page 123.

For DRAM, during both RAS and CAS cycles, some or all of the high address bits are on AD above those AD used for the RAS and CAS address bits. These high address bits can be used by the application, e.g., for decoding by external hardware to select I/O devices. On high-performance systems with fast CAS cycles, RAS cycles are often required for I/O address decoding. If the external decoding hardware is sufficiently fast, however, CAS-cycle I/O is possible.

For SRAM, to allow addressing as much memory as possible with CAS cycles, the only high address bit that appears during CAS address time is A31. I/O devices can still be selected on CAS cycles by translating the device addressing bits in software to lower address bits, provided that these translated bits do not interfere with the desired SRAM memory addressing. The device addressing bits must be translated to those address bits that appear during SRAM access on the AD that are externally decoded for I/O addressing.

Device Width
Memory device widths are either 8-bits (byte) or 32-bits (cell), and are programmed using mgXds in miscb.

As shown in Table 51, cell-wide memory groups do not use A1 or A0 to address the memory device. All accesses to cell-wide devices are cell-aligned and transfer the entire cell. Memory device address lines are attached to the CPU on AD[x:11] (x is determined by the device size).

Accesses to a byte-wide memory group are also cell-aligned and transfer all four bytes within the cell, from most significant to least significant (i.e., 0, 1 ,2, 3). The only exception is for an I/O-channel transfer with a one-byte byte-transfer device, in which case only one arbitrarily addressed byte is transferred. See *Bus Operation*, page 157.

As shown in Table 52, byte-wide memory devices require the use of A1 and A0. Since for DRAM the RAS and CAS memory device address bits must be on the same AD, the address lines (except A31) are internally rotated left two bits. This properly places A0 on AD11 for connection to DRAM. This also means, however, that the high address bits used for I/O address decoding appear on AD differently for a byte-wide memory group than for a cell-wide memory group. Since I/O device address decoding hardware is wired to fixed AD, the address bits used to access a device are different when transferring data with a byte-wide memory device than when transferring data with a cell-wide memory device.

ADVANCE INFORMATION

Table 51. RAS/CAS Address Line Configuration, Cell memory

Device Size	0,1	0	1	2,3	2	3	4,5, 6	4	5	6	7,8, 9	7	8	9	10, 11, 12	10	11	12	13, 14	13	14	15	
	64, 128 K	64K	128 K	256, 512 K	256 K	512 K	1, 2,4 M	1M	2M	4M	4, 8,16 M	4M	8M	16M	16, 32,6 4M	16M	32M	64M	64, 128 M	64M	128 M	SRAM	
	CAS	**RAS**		**CAS**	**RAS**		**CAS**	**RAS**			**CAS**	**RAS**			**CAS**	**RAS**			**CAS**	**RAS**		**CAS**	**RAS**
#BITS[1]	8	8	9	9	9	10	10	10	11	12	11	11	12	13	12	12	13	14	13	13	14	n/a	n/a
AD9	A0	A0	A0	A0	A0	A0	A0	A0	A0	A0	A0	A0	A0	A0	A0	A0	A0	A0	A0	A0	A0	A0	A0
AD10	A1	A1	A1	A1	A1	A1	A1	A1	A1	A1	A1	A1	A1	A1	A1	A1	A1	A1	A1	A1	A1	A1	A1
AD11	A2	A10	A10	A2	A11	A11	A2	A12	A12	A12	A2	A13	A13	A13	A2	A14	A14	A14	A2	A15	A15	A2	A11
AD12	A3	A11	A11	A3	A12	A12	A3	A13	A13	A13	A3	A14	A14	A14	A3	A15	A15	A15	A3	A16	A16	A3	A12
AD13	A4	A12	A12	A4	A13	A13	A4	A14	A14	A14	A4	A15	A15	A15	A4	A16	A16	A16	A4	A17	A17	A4	A13
AD14	A5	A13	A13	A5	A14	A14	A5	A15	A15	A15	A5	A16	A16	A16	A5	A17	A17	A17	A5	A18	A18	A5	A14
AD15	A6	A14	A14	A6	A15	A15	A6	A16	A16	A16	A6	A17	A17	A17	A6	A18	A18	A18	A6	A19	A19	A6	A15
AD16	A7	A15	A15	A7	A16	A16	A7	A17	A17	A17	A7	A18	A18	A18	A7	A19	A19	A19	A7	A20	A20	A7	A16
AD17	A8	A16	A16	A8	A17	A17	A8	A18	A18	A18	A8	A19	A19	A19	A8	A20	A20	A20	A8	A21	A21	A8	A17
AD18	A9	A17	A17	A9	A18	A18	A9	A19	A19	A19	A9	A20	A20	A20	A9	A21	A21	A21	A9	A22	A22	A9	A18
AD19	A18	A18	A18	A10	A19	A19	A10	A20	A20	A20	A10	A21	A21	A21	A10	A22	A22	A22	A10	A23	A23	A10	A19
AD20	A19	A19	A19	A20	A20	A20	A11	A21	A21	A21	A11	A22	A22	A22	A11	A23	A23	A23	A11	A24	A24	A11	A20
AD21	A20	A20	A20	A21	A21	A21	A21	A21	A22	A22	A12	A23	A23	A23	A12	A24	A24	A24	A12	A25	A25	A12	A21
AD22	A21	A21	A21	A22	A22	A22	A22	A22	A22	A23	A22	A22	A24	A24	A13	A25	A25	A25	A13	A26	A26	A13	A22
AD23	A22	A22	A22	A23	A23	A23	A23	A23	A23	A23	A23	A23	A23	A25	A23	A23	A26	A26	A14	A27	A27	A14	A23
AD24	A24	A24	A24	A24	A24	A24	A24	A24	A24	A24	A24	A24	A24	A24	A24	A24	A24	A27	A24	A24	A28	A15	A24
AD25	A25	A25	A25	A25	A25	A25	A25	A25	A25	A25	A25	A25	A25	A25	A25	A25	A25	A25	A25	A25	A25	A16	A25
AD26	A26	A26	A26	A26	A26	A26	A26	A26	A26	A26	A26	A26	A26	A26	A26	A26	A26	A26	A26	A26	A26	A17	A26
AD27	A27	A27	A27	A27	A27	A27	A27	A27	A27	A27	A27	A27	A27	A27	A27	A27	A27	A27	A27	A27	A27	A18	A27
AD28	A28	A28	A28	A28	A28	A28	A28	A28	A28	A28	A28	A28	A28	A28	A28	A28	A28	A28	A28	A28	A28	A19	A28
AD29	A29	A29	A29	A29	A29	A29	A29	A29	A29	A29	A29	A29	A29	A29	A29	A29	A29	A29	A29	A29	A29	A20	A29
AD30	A30	A30	A30	A30	A30	A30	A30	A30	A30	A30	A30	A30	A30	A30	A30	A30	A30	A30	A30	A30	A30	A21	A30
AD31	A31	A31	A31	A31	A31	A31	A31	A31	A31	A31	A31	A31	A31	A31	A31	A31	A31	A31	A31	A31	A31	A31	A31

Notes: 1. #BITS is the number of CAS or RAS address bits for the specified device size.

▨ Location of DRAM CAS or RAS address bits for the specified device size.

□ Location of bank-select bits in MMB mode for the specified device size.

Table 52. RAS/CAS Address Line Configuration, Byte Memory

Device Size	0,1	0	1	2,3	2	3	4,5,6	4	5	6	7,8,9	7	8	9	10,11,12	10	11	12	13,14	13	14	15	
	64,128K	64K	128K	256,512K	256K	512K	1,2,4M	1M	2M	4M	4,8,16M	4M	8M	16M	16,32,64M	16M	32M	64M	64,128M	64M	128M	SRAM	
	CAS	RAS		CAS	RAS		CAS	RAS			CAS	RAS			CAS	RAS			CAS	RAS		CAS	RAS
#BITS[1]	**8**	**8**	**9**	**9**	**9**	**10**	**10**	**10**	**11**	**12**	**11**	**11**	**12**	**13**	**12**	**12**	**13**	**14**	**13**	**13**	**14**	**n/a**	**n/a**
AD9	A29	A29	A29	A29	A29	A29	A29	A29	A29	A29	A29	A29	A29	A29	A29	A29	A29	A29	A29	A29	A29	A29	A29
AD10	A30	A30	A30	A30	A30	A30	A30	A30	A30	A30	A30	A30	A30	A30	A30	A30	A30	A30	A30	A30	A30	A30	A30
AD11	A0	A8	A8	A0	A9	A9	A0	A10	A10	A10	A0	A11	A11	A11	A0	A12	A12	A12	A0	A13	A13	A0	A9
AD12	A1	A9	A9	A1	A10	A10	A1	A11	A11	A11	A1	A12	A12	A12	A1	A13	A13	A13	A1	A14	A14	A1	A10
AD13	A2	A10	A10	A2	A11	A11	A2	A12	A12	A12	A2	A13	A13	A13	A2	A14	A14	A14	A2	A15	A15	A2	A11
AD14	A3	A11	A11	A3	A12	A12	A3	A13	A13	A13	A3	A14	A14	A14	A3	A15	A15	A15	A3	A16	A16	A3	A12
AD15	A4	A12	A12	A4	A13	A13	A4	A14	A14	A14	A4	A15	A15	A15	A4	A16	A16	A16	A4	A17	A17	A4	A13
AD16	A5	A13	A13	A5	A14	A14	A5	A15	A15	A15	A5	A16	A16	A16	A5	A17	A17	A17	A5	A18	A18	A5	A14
AD17	A6	A14	A14	A6	A15	A15	A6	A16	A16	A16	A6	A17	A17	A17	A6	A18	A18	A18	A6	A19	A19	A6	A15
AD18	A7	A15	A15	A7	A16	A16	A7	A17	A17	A17	A7	A18	A18	A18	A7	A19	A19	A19	A7	A20	A20	A7	A16
AD19	A16	A16	A16	A8	A17	A17	A8	A18	A18	A18	A8	A19	A19	A19	A8	A20	A20	A20	A8	A21	A21	A8	A17
AD20	A17	A17	A17	A18	A18	A18	A9	A19	A19	A19	A9	A20	A20	A20	A9	A21	A21	A21	A9	A22	A22	A9	A18
AD21	A18	A18	A18	A19	A19	A19	A19	A19	A20	A20	A10	A21	A21	A21	A10	A22	A22	A22	A10	A23	A23	A10	A19
AD22	A19	A19	A19	A20	A20	A20	A20	A20	A20	A21	A20	A20	A22	A22	A11	A23	A23	A23	A11	A24	A24	A11	A20
AD23	A20	A20	A20	A21	A21	A21	A21	A21	A21	A21	A21	A21	A21	A23	A21	A21	A24	A24	A12	A25	A25	A12	A21
AD24	A21	A21	A21	A22	A22	A22	A22	A22	A22	A22	A22	A22	A22	A22	A22	A22	A22	A25	A13	A22	A26	A13	A22
AD25	A22	A22	A22	A23	A23	A23	A23	A23	A23	A23	A23	A23	A23	A23	A23	A23	A23	A23	A14	A23	A23	A14	A23
AD26	A24	A24	A24	A24	A24	A24	A24	A24	A24	A24	A24	A24	A24	A24	A24	A24	A24	A24	A24	A24	A24	A15	A24
AD27	A25	A25	A25	A25	A25	A25	A25	A25	A25	A25	A25	A25	A25	A25	A25	A25	A25	A25	A25	A25	A25	A16	A25
AD28	A26	A26	A26	A26	A26	A26	A26	A26	A26	A26	A26	A26	A26	A26	A26	A26	A26	A26	A26	A26	A26	A17	A26
AD29	A27	A27	A27	A27	A27	A27	A27	A27	A27	A27	A27	A27	A27	A27	A27	A27	A27	A27	A27	A27	A27	A18	A27
AD30	A28	A28	A28	A28	A28	A28	A28	A28	A28	A28	A28	A28	A28	A28	A28	A28	A28	A28	A28	A28	A28	A19	A28
AD31	A31	A31	A31	A31	A31	A31	A31	A31	A31	A31	A31	A31	A31	A31	A31	A31	A31	A31	A31	A31	A31	A31	A31

Notes: 1. #BITS is the number of CAS or RAS address bits for the specified device size.

▨ Location of DRAM CAS or RAS address bits for the specified device size.

▣ Location of bank-select bits in MMB mode for the specified device size.

ADVANCE INFORMATION

Programmable Timing

The timing for RAS and CAS cycles on each memory group, as well as data setup and hold times for each I/O channel, is programmable. Depending on the parameter, timing granularity is in either CPU-clock cycles or 2X-CPU-clock cycles. In some cases, timing is specified in CPU-clock cycles with a modifier available to advance the event by one 2X-CPU-clock cycle.

In all cases, the hardware actually counts time in CPU-clock cycle granules and then delays or advances the signal transition by any 2X-CPU-clock granularity timing specified. If the rate of the external clock is changed during operation, 2X-CPU-clock granularity timing generated by the 2X-CPU-clock PLL must not be in effect during the time of the change because the PLL cannot track the change and improper clock cycles will be generated. Simply program the timing to measure an integral number of CPU-clock cycles. See 72, page 204.

Timing specification is broken into three pieces: RAS prefix, basic CAS cycle, and CAS extension/expansion timing. All CAS cycles consist of the basic CAS cycle timing and the appropriate CAS extension/expansion timing. This combination is referred to as the *CAS part* of the memory cycle. All RAS cycles consist of a RAS prefix plus a CAS part. Bus transactions of multiple bus cycles are simply the required sequence of RAS prefixes and CAS parts in immediate succession. Only discrete read cycles or write cycles are performed; read-modify-write cycles are not performed.

To gain access to the bus, the bus address must be transferred to the MIF and a check made to see if the bus is available for the time required to complete the bus transaction. This bus request process takes two CPU-clock cycles at the beginning of each bus transaction. Memory-reference MPU and VPU instructions always overlap one cycle of instruction execution with the bus request process. DMA operation can overlap both cycles of the bus request process with a preceding MPU bus transaction. Thus, except for DMA overlapped with an MPU bus transaction, there are two inactive CPU-clock cycles on the bus preceding each bus transaction. Instruction execution times listed herein include the bus access and programmed bus transaction time as part of the entire memory reference time.

RAS Prefix Timing

This timing for a memory group is specified by programming the fields in the corresponding mgXrasbt. The RAS prefix of a RAS cycle consists of a leading CPU-clock cycle; the $\overline{\mathtt{RAS}}$ inactive portion, also referred to as RAS precharge (mgbtras); and the RAS address hold time (mgbtrhld). The last two are modified by the early RAS bit (mgbteras). For computation of the RAS-cycle duration, mgbtrast must contain the sum of mgbtras and mgbtrhld plus one. During this time the DRAM RAS address bits, high address bits, and bit outputs are on AD. See Figure 42, page 149.

CAS Part Timing

This timing for a memory group is specified by programming the fields in mgXcasbt and mgXebt. The CAS part of the cycle begins with the timing for the $\overline{\mathtt{CAS}}$ inactive portion, also referred to as CAS precharge (mgbtcas). Next is the CAS address hold time/beginning of data time (mgbtdob), when $\overline{\mathtt{DOB}}$, and possibly $\overline{\mathtt{OE}}$ or $\overline{\mathtt{LWE}}$, go active. Then $\overline{\mathtt{CAS}}$, $\overline{\mathtt{DOB}}$, and either $\overline{\mathtt{OE}}$ (if a memory read) or both $\overline{\mathtt{EWE}}$ and $\overline{\mathtt{LWE}}$ (if a memory write) go inactive again (mgbtcast). To accommodate longer data setup and buffer delay times, the CAS cycle can be expanded at $\overline{\mathtt{DOB}}$ fall (mgebtdobe). To accommodate longer data hold and output buffer disable times, the CAS strobes can be extended following $\overline{\mathtt{DOB}}$ inactive (mgebtcase). Memory write cycles can be programmed to have $\overline{\mathtt{EWE}}$ go active either at the beginning of the CAS cycle (before $\overline{\mathtt{RAS}}$ rise if a RAS cycle) or at $\overline{\mathtt{CAS}}$ fall (mgbtewea). Similarly, $\overline{\mathtt{LWE}}$ can be programmed to go active either at $\overline{\mathtt{DOB}}$ fall plus expansion or at $\overline{\mathtt{DOB}}$ fall plus expansion plus one 2X-CPU-clock cycle (mgbtlwea). $\overline{\mathtt{EWE}}$ generally accommodates SRAM-type devices and $\overline{\mathtt{LWE}}$ accommodates DRAM-type devices. Further, $\overline{\mathtt{DOB}}$ going inactive tracks $\overline{\mathtt{EWE}}$/$\overline{\mathtt{LWE}}$ or $\overline{\mathtt{OE}}$, either of which can be made to go inactive earlier than the unextended CAS time by one 2X-CPU-clock cycle (mgbtewe and mgbteoe). For computation of CAS-cycle duration, mgbtcast is added to mgebtsum, the latter of which must contain the sum of `mgebtdobe` and `mgebtcase`. See Figure 41, page 147, and Figure 40, page 146.

When MPU bus transactions or VPU instruction-fetch bus transactions occur, the bus cycle timing for the memory group uses the values in `mgXebt`, as described above. When an I/O channel bus transaction occurs, the values in `ioXebt` for the appropriate I/O channel are substituted for the `mgXebt` values. The `ioXebt` values must be programmed to accommodate any memory group that might be involved in the transfer, as well as the I/O device.

DRAM Refresh

DRAM requires periodic accesses to each row within the memory device to maintain the memory contents. Most DRAM devices support several modes of refresh, including the RAS-only refresh mode supplied by the VPU instruction `refresh`. The VPU must be programmed to execute `refresh` at intervals short enough for the most restrictive DRAM in the system. The timing during the refresh cycle uses the RAS cycle timing of the memory group indicated by `msrtg`, which must be long enough for the slowest DRAM refresh cycle in the system. Refresh on each memory group can be individually enabled or disabled. See Figure 33, page 139.

`msra` contains data used during each refresh cycle. `refresh` increments the14-bit row address in `msrra` after the refresh cycle completes. The address bits in `msra31` and `msrha` are normally zero, but can be written if the zero values interfere with other system hardware during refresh cycles.

Video RAM Support

Special VRAM operating modes are supported through the use of `vram`. See Figure 32, page 137, and Table 37, page 37. Many VRAM modes use a RAS cycle to set an operating state in the VRAM device that persists until the next RAS cycle occurs on that VRAM device. Unexpected RAS cycles can thus cause undesirable results.

Refresh cycles are one source of unexpected RAS cycles; these can be disabled on groups containing VRAM by setting the appropriate `mgXrd` bits. See Figure 33, page 139.

Changes in the high address bits are a second source of unexpected RAS cycles; these can be prevented from occurring on memory group `msvgrp` by setting `msevhacr`. The high address bits are typically used for I/O device addresses, and require a RAS cycle when these bits change if `mshacd` is clear. An I/O-channel transfer immediately prior to a VRAM group access is an example of such an occurrence. The RAS cycle might be required for proper system operation, but the VRAM group can be prevented from receiving the RAS cycle by setting `msevhacr`. The RAS precharge portion of the cycle will occur on RAS and $\overline{\mathtt{RAS}}$, but not on the $\overline{\mathtt{MGSx}}$/$\overline{\mathtt{RASx}}$ of the VRAM group. Note that if more than one memory group is used for VRAM then this protection is not effective. See Figure 39, page 145.

ADVANCE INFORMATION

System Requirements Programming

RAS Cycle Generation

RAS cycles are primarily required to bring new row addresses onto AD for DRAM-type devices. They are also required, in certain instances, to ensure temporally deterministic execution of the VPU, or to ensure correct operation after certain events. The MIF handles these cases automatically. RAS cycles can also be configured to occur in order to supply additional time for decoding I/O addresses, for example. Since RAS cycles generally take considerably longer than CAS cycles, it is desirable to minimize their use. The various sources of RAS cycles are listed in Table 53, page 126.

When the current and previous addresses are compared to determine if a RAS cycle is required, the MIF uses the following rules:

- The current DRAM RAS address bits are compared to those from the most recent RAS cycle on the current memory group. If the bits are different, a RAS cycle occurs.

Table 53. Sources of RAS cycles

Group	Access	Reason	Configuration	Requirement	MPU DMA VPU
all	any	High address bits changed	mshacd clear	S	all
all	any	A31 changed	mshacd clear, msexa31hac clear	S	all
all	any	A31 set	msras31d clear	S	all
all	any	Memory group row address changed		C	all
pgm	first	After VRAM CAS before RAS	msvgrp	C	all
pgm	first	after refresh on enabled group	mgXrd set	C	MPU,DMA
all	first	after refresh executes		T	VPU
all	first	after memory fault on group		C	all
all	first	mgds written		C	all
all	first	CPU hardware reset		C	all
all	first	delay completes		T	VPU
all	first	VPU software reset		T	VPU

KEY:
all – any group or device with which the event might occur
pgm – any group programmed for the event to occur
any – any arbitrary access creating the specified condition
first – first access on each specified group after the specified event
S – might be required by system hardware
C – might be required for correct operation of devices
T – required for temporally deterministic VPU execution

- The middle address bits are not compared (see Figure 20, page 118). The middle address bits are: for DRAM, above the RAS address bits up to and including `msgsm`; for SRAM, from A22 up to and including `msgsm`. If `msgsm` is zero there are no middle address bits in either case. If `msgsm` includes A31, A31 becomes part of the high address bits and is optionally compared.
- The current high address bits are compared to those from the most recent RAS cycle, depending on the configuration options discussed below. The location of the high address bits depends on `msgsm`. See Figure 37, page 143.

Three high-address-bit configuration options are available to minimize the occurrence of RAS cycles caused by high-address-bit comparisons.

- The high address bits are typically used for I/O device addresses, and thus when they change, a RAS cycle might be required for their proper decoding by external hardware. The high address bits can be excluded from RAS-cycle determination by setting the memory system high-address-bit compare disable (`mshacd`). See Figure 33, page 139.
- During bus transactions between four-byte byte-transfer devices and cell memory or between one-cell cell-transfer devices and byte memory, A31is passed (taken from the global register, usually set) or cleared (by the MIF) to select or deselect the I/O device when required. Decoding A31 externally for this purpose can be done more quickly than a full address decode, so this separate option is available. A31 can be included in or excluded from the high-address-bit compare (`msexa31hac`). See Figure 39, page 145.
- In systems that require a RAS cycle to decode I/O device addresses but not to decode changes in A31 (`mshacd` clear and `msexa31hac` set), it might be necessary for the memory address bits and I/O addressing bits to overlap if the system contains a large amount of memory and I/O devices. This can prevent a RAS cycle from occurring because some of the overlapped address bits do not cause a RAS (middle address bits), or do not require a RAS (DRAM RAS address bits), even though they changed from the last system RAS cycle. In this case, a RAS can be forced to ensure that I/O device addresses are decoded by setting A31 (`msras31d` clear). This option can also be useful any other time forcing a RAS cycle is desirable.

Driver Current

The drive capability of all the package output drivers is programmable. See Figure 50, page 154.

Memory Faults

Virtual memory page-fault detection is enabled through `mflt_enable` in `mode`. The memory fault input can either come from AD8 or $\overline{\texttt{MFLT}}$, depending on the state of `pkgmflt`. See Figure 39, page 145.

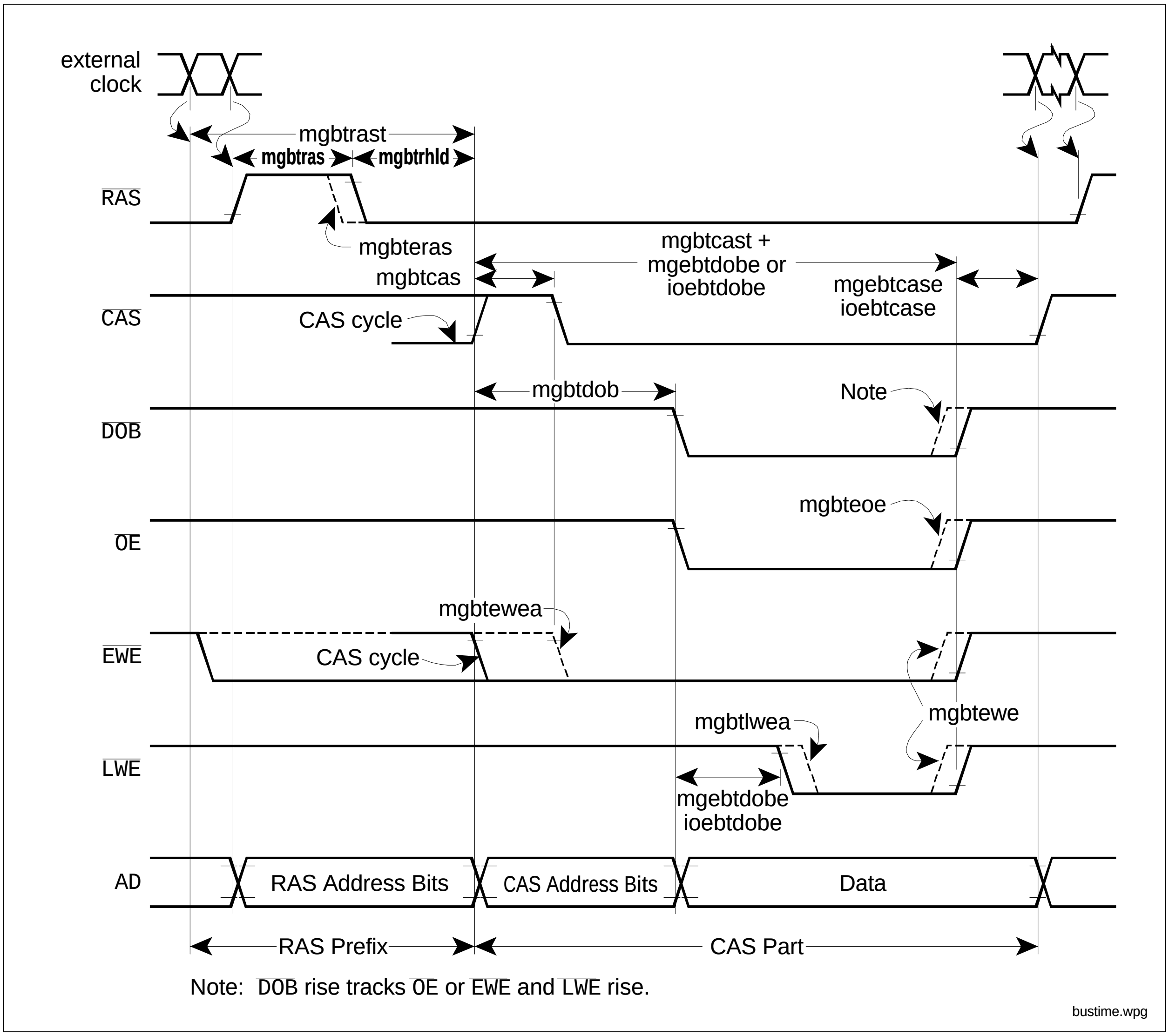
I/O-Channel Programming

As previously discussed, the normal memory-group bus timing is changed during an I/O-channel bus transaction by substituting the values in the corresponding `ioXebt` for the values in `mgXebt` for the memory group involved. This allows each I/O channel to be programmed to meet the requirements of the device. The `ioXebt` values must be adequate for the I/O device, as well as any memory group with which a data transfer might occur. See Figure 43, page 150.

In addition to timing, the type of transfer on each I/O channel can be specified in `iodtta` or `iodttb`. Transfers can either be one byte or four bytes per transaction for byte-wide devices, or one cell per transaction for cell-wide devices. Four-byte byte-transfer devices might contend for the bus less often than one-byte byte-transfer devices, and thus can transfer data more efficiently. Also, with cell-wide memory, four-byte byte transfers are cell-aligned and pack the data into the memory cells, whereas one-byte byte transfers place only one byte per memory cell. See *Bus Operation*, page 157.

See *Direct Memory Access Controller*, page 103, for other I/O-channel transfer options.

ADVANCE INFORMATION

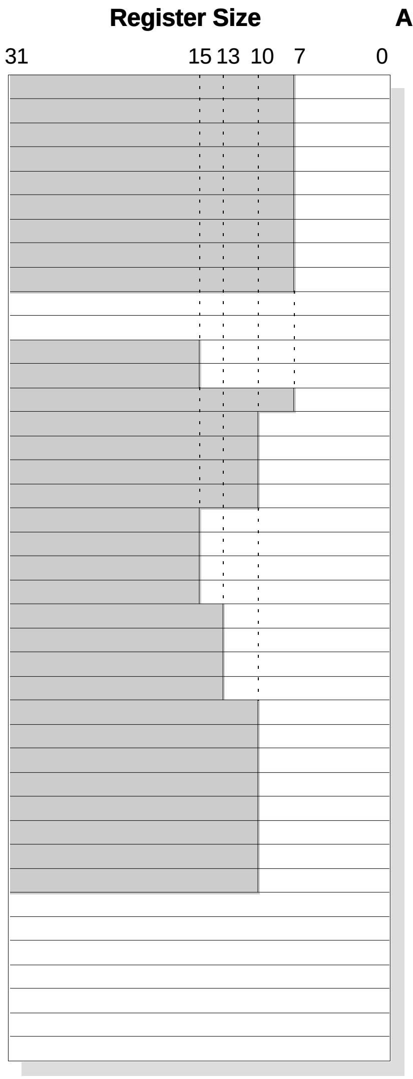


Note: DOB rise tracks OE or EWE and LWE rise.

Figure 23. Programmable Bus Timing Reference

On-Chip Resource Registers

The on-chip resource registers comprise portions of various functional areas on the CPU including the MPU, VPU, DMAC, INTC, MIF, bit inputs, and bit outputs. The registers are addressed from the MPU in their own address space using the instructions `ldo[]` and `sto[]` at the register level, or `ldo.i[]` and `sto.i[]` at the bit level (for those registers that have bit addresses). On other processors, resources of this type are often either memory-mapped or opcode-mapped. By using a separate address space for these resources, the normal address space remains uncluttered, and opcodes are preserved. Except as noted, all registers are readable and writeable. Areas marked “Reserved Zeros” contain no programmable bits and always return zero. Areas marked “Reserved” contain unused programmable bits. Both areas might contain functional programmable bits in the future.



Addr	Mnemonic	Description
000	ioin	Bit Input Register
020	ioip	Interrupt Pending Register
040	ioius	Interrupt Under Service Register
060	ioout	Bit Output Register
080	ioie	Interrupt Enable Register
0a0	iodmae	DMA Enable Register
0c0	vram	VRAM Control Bit Register
0e0	misca	Miscellaneous A Register
100	miscb	Miscellaneous B Register
120	mfltaddr	Memory Fault Address Register
140	mfltdata	Memory Fault Data Register
160	msgsm	Memory System Group Select Mask Register
180	mgds	Memory Group Device Size Register
1a0	miscc	Miscellaneous C Register
1c0	mg0ebt	Memory Group 0 Extended Bus Timing Register
1e0	mg1ebt	Memory Group 1 Extended Bus Timing Register
200	mg2ebt	Memory Group 2 Extended Bus Timing Register
220	mg3ebt	Memory Group 3 Extended Bus Timing Register
240	mg0casbt	Memory Group 0 CAS Bus Timing Register
260	mg1casbt	Memory Group 1 CAS Bus Timing Register
280	mg2casbt	Memory Group 2 CAS Bus Timing Register
2a0	mg3casbt	Memory Group 3 CAS Bus Timing Register
2c0	mg0rasbt	Memory Group 0 RAS Bus Timing Register
2e0	mg1rasbt	Memory Group 1 RAS Bus Timing Register
300	mg2rasbt	Memory Group 2 RAS Bus Timing Register
320	mg3rasbt	Memory Group 3 RAS Bus Timing Register
340	io0ebt	I/O Channel 0 Extended Bus Timing Register
360	io1ebt	I/O Channel 1 Extended Bus Timing Register
380	io2ebt	I/O Channel 2 Extended Bus Timing Register
3a0	io3ebt	I/O Channel 3 Extended Bus Timing Register
3c0	io4ebt	I/O Channel 4 Extended Bus Timing Register
3e0	io5ebt	I/O Channel 5 Extended Bus Timing Register
400	io6ebt	I/O Channel 6 Extended Bus Timing Register
420	io7ebt	I/O Channel 7 Extended Bus Timing Register
440	msra	Memory System Refresh Address Register (WO)
440	vpudelay	VPU Delay Register (RO)
460	iodtta	I/O Device Transfer Types A Register
480	iodttb	I/O Device Transfer Types B Register
7a0	iodmaex	I/O DMA Enable Expiration Register
7c0	drivers	Driver Current Register
7e0	vpureset	VPU Reset Register

onchipmp.wpg

Figure 24. On-Chip Resource Registers

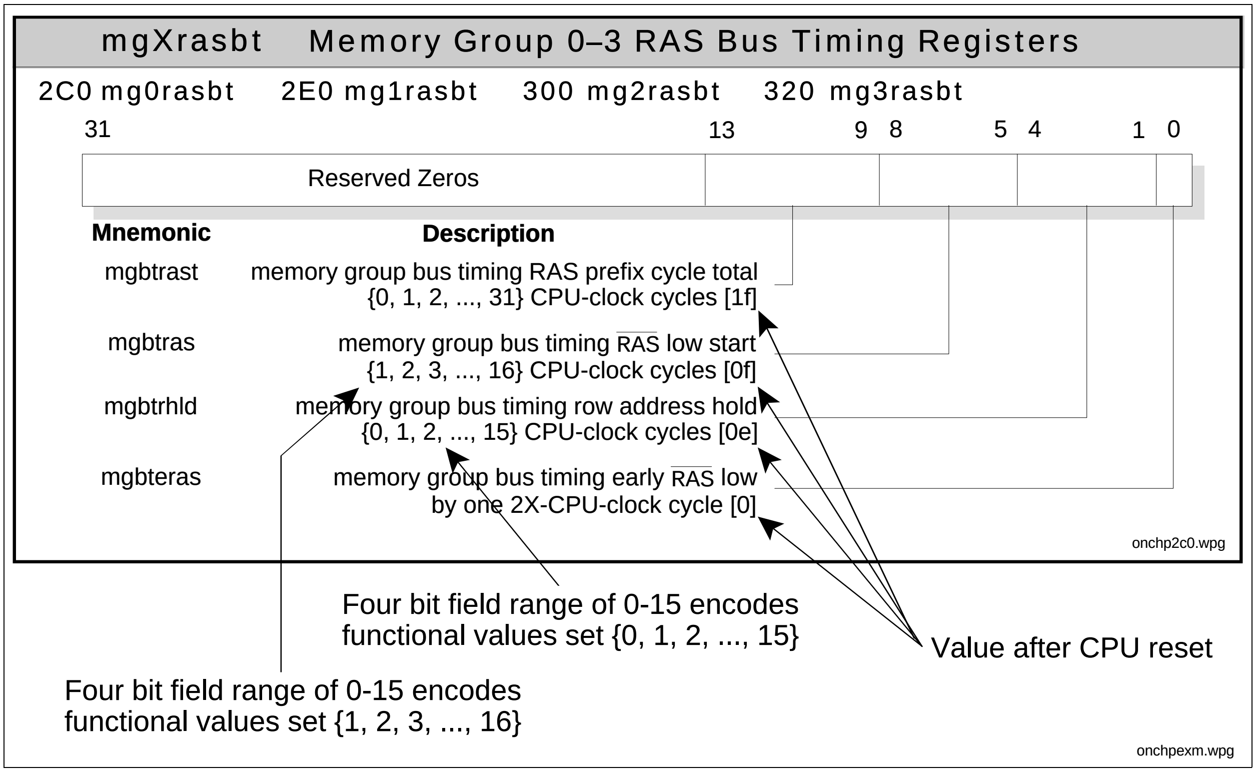

mgXrasbt Memory Group 0–3 RAS Bus Timing Registers
2C0 mg0rasbt
2E0 mg1rasbt
300 mg2rasbt
320 mg3rasbt
31
13
9
8
5
4
1
0
Reserved Zeros
Mnemonic
Description
mgbtrast
memory group bus timing RAS prefix cycle total
{0, 1, 2, ..., 31} CPU-clock cycles [1f]
mgbtras
memory group bus timing RAS low start
{1, 2, 3, ..., 16} CPU-clock cycles [0f]
mgbtrhld
memory group bus timing row address hold
{0, 1, 2, ..., 15} CPU-clock cycles [0e]
mgbteras
memory group bus timing early RAS low
by one 2X-CPU-clock cycle [0]
onchp2c0.wpg
Four bit field range of 0-15 encodes
functional values set {0, 1, 2, ..., 15}
Value after CPU reset
Four bit field range of 0-15 encodes
functional values set {1, 2, 3, ..., 16}
onchpexm.wpg

Figure 25. Example On-Chip Register Diagram

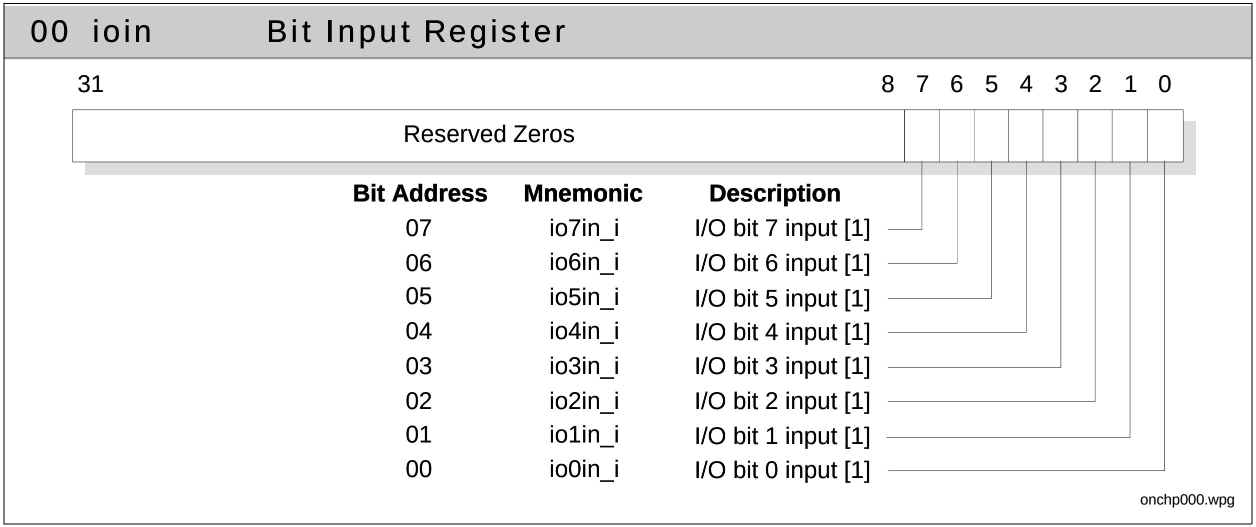
The first several registers are bit addressable in addition to being register addressable. This allows the MPU to modify individual bits without corrupting other bits that might be changed concurrently by the VPU, DMAC, or INTC logic.

Bus activity must be prevented to avoid a possible invalid bus cycle when *changing* the value in any register that affects the bus configuration or timing of a bus cycle that might be in progress. Bus activity can be prevented by ensuring:

- no DMA requests are serviced,
- the VPU does not seize the bus (because vpudelay goes to zero),
- no writes are posted, and
- pre-fetch does not occur.

This is typically not a problem because most changes are made just after power-up when no DMA or VPU activity of concern is occurring. Posted writes can be ensured complete by ensuring an MPU memory access (such as an instruction fetch) occurs after the write is posted.

The diagrams that follow use a {} notation that depicts the decoded set of values represented by ordinal values within the corresponding bit field. The full range of values possible on a bit field are always depicted. Thus {1, 2, 3, 4} is only be possible on a two-bit-wide field. In this case, a zero in the field represents a one value, a one in the field represents a two value, and so on through the list. Note that not all sets are consecutive numbers, such as {0, 1, 2, 4}. Also note that references in the text to usage of a field imply the decoded value represented by the field, not the ordinal values, e.g., references to mgbtras in the example imply the decoded values 1–16 and not the ordinal values 0–15 programmed into the field.



Bit Address	Mnemonic	Description
07	io7in_i	I/O bit 7 input [1]
06	io6in_i	I/O bit 6 input [1]
05	io5in_i	I/O bit 5 input [1]
04	io4in_i	I/O bit 4 input [1]
03	io3in_i	I/O bit 3 input [1]
02	io2in_i	I/O bit 2 input [1]
01	io1in_i	I/O bit 1 input [1]
00	io0in_i	I/O bit 0 input [1]

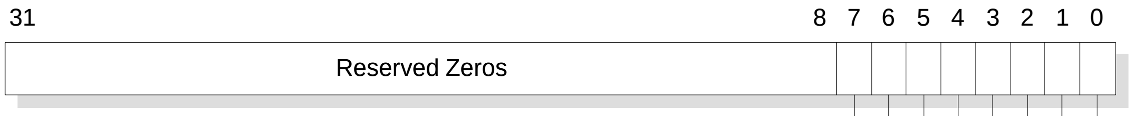
Figure 26. Bit Input Register

Contains sampled data from $\overline{\texttt{IN}}$`[7:0]` or AD`[7:0]`, depending on the value of `pkgio`. `ioin` is the source of inputs for all consumers of bit inputs. Bits are zero-persistent: once a bit is zero in `ioin` it stays zero until consumed by the VPU, DMAC, or INTC, or written by the MPU with a one. Under certain conditions bits become not zero-persistent. See *Bit Inputs*, page 111.

The bits can be individually read, set and cleared to prevent race conditions between the MPU and other CPU logic.

ADVANCE INFORMATION

20 ioip Interrupt Pending Register



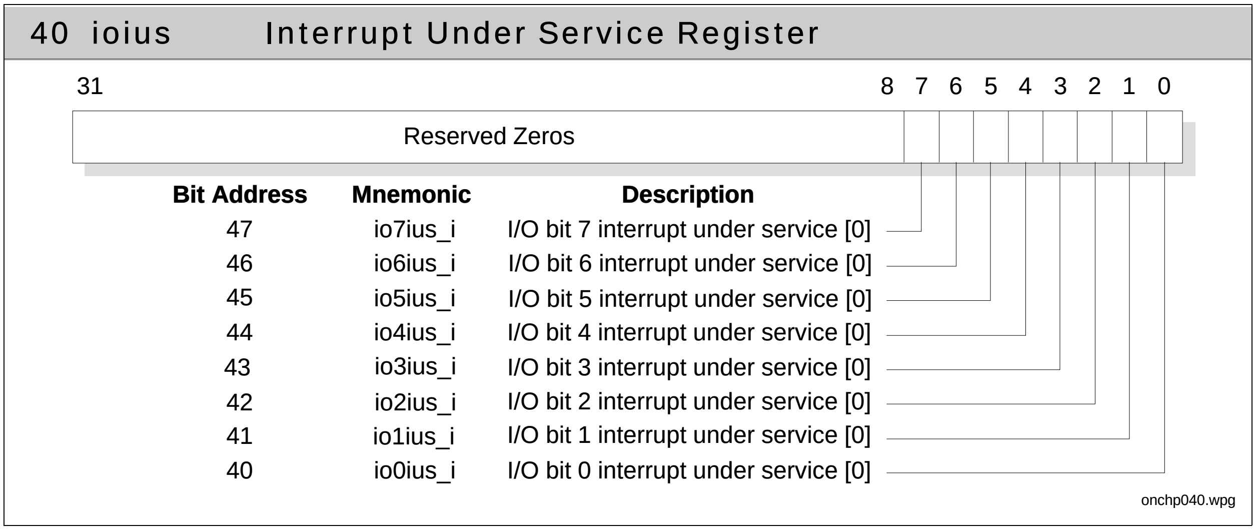
Bit Address	Mnemonic	Description
27	io7ip_i	I/O bit 7 interrupt pending [0]
26	io6ip_i	I/O bit 6 interrupt pending [0]
25	io5ip_i	I/O bit 5 interrupt pending [0]
24	io4ip_i	I/O bit 4 interrupt pending [0]
23	io3ip_i	I/O bit 3 interrupt pending [0]
22	io2ip_i	I/O bit 2 interrupt pending [0]
21	io1ip_i	I/O bit 1 interrupt pending [0]
20	io0ip_i	I/O bit 0 interrupt pending [0]

onchp020.wpg

Figure 27. Interrupt Pending Register

Contains interrupt requests that are waiting to be serviced. Interrupts are serviced in order of priority (0 = highest, 7 = lowest). An interrupt request from an I/O-channel transfer or from `int` occurs by the corresponding pending bit being set. Bits can be set or cleared to submit or withdraw interrupt requests. When an `ioip` bit and corresponding `ioie` bit are set, the corresponding `ioin` bit is not zero-persistent. See *Interrupt Controller*, page 107.

The bits can be individually read, set and cleared to prevent race conditions between the MPU and INTC logic.

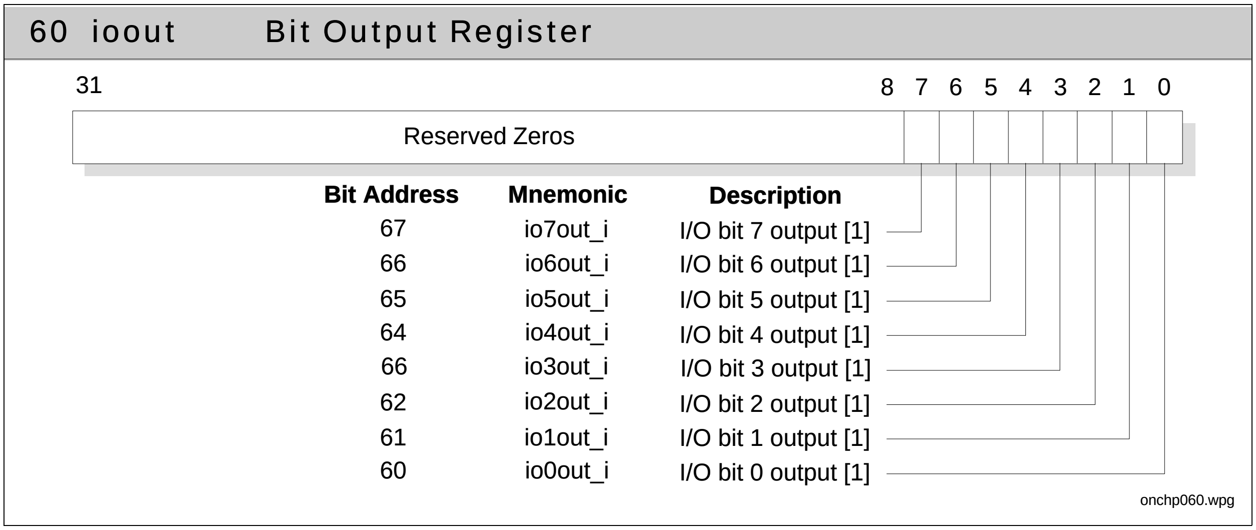


Bit Address	Mnemonic	Description
47	io7ius_i	I/O bit 7 interrupt under service [0]
46	io6ius_i	I/O bit 6 interrupt under service [0]
45	io5ius_i	I/O bit 5 interrupt under service [0]
44	io4ius_i	I/O bit 4 interrupt under service [0]
43	io3ius_i	I/O bit 3 interrupt under service [0]
42	io2ius_i	I/O bit 2 interrupt under service [0]
41	io1ius_i	I/O bit 1 interrupt under service [0]
40	io0ius_i	I/O bit 0 interrupt under service [0]

Figure 28. Interrupt Under Service Register

Contains the current interrupt service request and those that have been temporarily suspended to service a higher-priority request. When an ISR executable-code vector for an interrupt request is executed, the `ioius` bit for that interrupt request is set and the corresponding `ioip` bit is cleared. When an ISR executes `reti`, the highest-priority interrupt under-service bit is cleared. The bits are used to prevent interrupts from interrupting higher-priority ISRs. When an `ioius` bit and corresponding `ioie` bit are set, the corresponding `ioin` bit is not zero-persistent. See *Interrupt Controller*, page 107.

The bits can be individually read, set and cleared to prevent race conditions between the MPU and INTC logic.



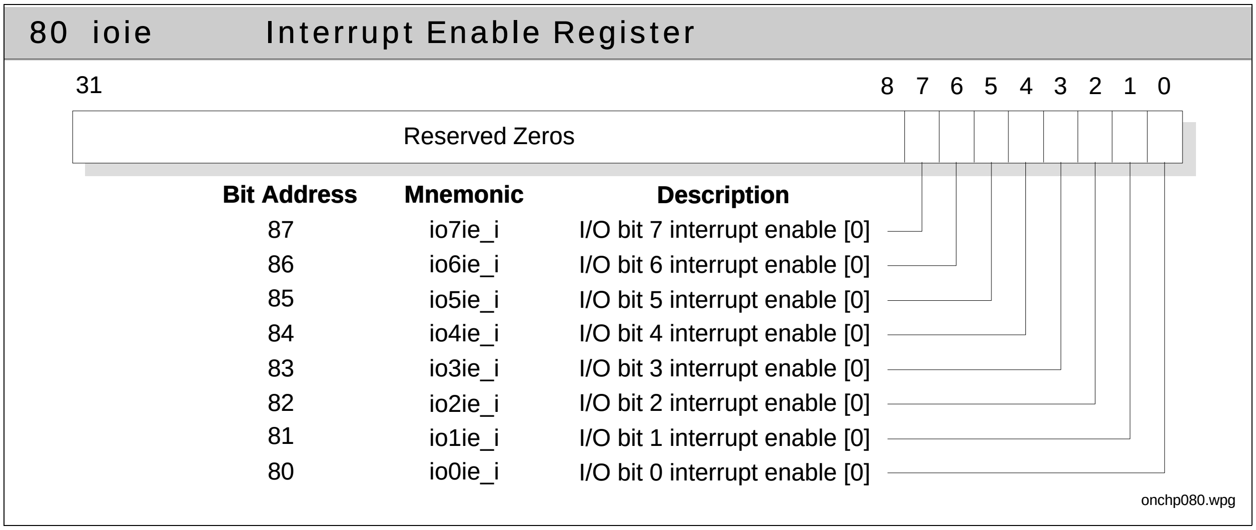
60 ioout Bit Output Register

Bit Address	Mnemonic	Description
67	io7out_i	I/O bit 7 output [1]
66	io6out_i	I/O bit 6 output [1]
65	io5out_i	I/O bit 5 output [1]
64	io4out_i	I/O bit 4 output [1]
66	io3out_i	I/O bit 3 output [1]
62	io2out_i	I/O bit 2 output [1]
61	io1out_i	I/O bit 1 output [1]
60	io0out_i	I/O bit 0 output [1]

Figure 29. Bit Output Register

Contains the bits from MPU and VPU bit-output operations. Bits appear on `OUT[7:0]` immediately after writing and on `AD[7:0]` while $\overline{\texttt{RAS}}$ is inactive. See *Bit Outputs*, page 115.

The bits can be individually read, set and cleared to prevent race conditions between the MPU and VPU.



80 ioie Interrupt Enable Register

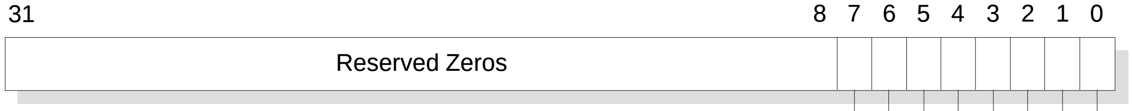
Bit Address	Mnemonic	Description
87	io7ie_i	I/O bit 7 interrupt enable [0]
86	io6ie_i	I/O bit 6 interrupt enable [0]
85	io5ie_i	I/O bit 5 interrupt enable [0]
84	io4ie_i	I/O bit 4 interrupt enable [0]
83	io3ie_i	I/O bit 3 interrupt enable [0]
82	io2ie_i	I/O bit 2 interrupt enable [0]
81	io1ie_i	I/O bit 1 interrupt enable [0]
80	io0ie_i	I/O bit 0 interrupt enable [0]

Figure 30. Interrupt Enable Register

If the corresponding `iodmae` bit is not set, allows a corresponding zero bit in `ioin` to request the corresponding interrupt service. When an enabled interrupt request is recognized, the corresponding `ioip` bit is set and the corresponding `ioin` bit is no longer zero-persistent. See *Interrupt Controller*, page 107.

The bits can be individually read, set and cleared. Bit addressability for this register is an artifact of its position in the address space, and does not imply any race conditions on this register can exist.

A0 iodmae DMA Enable Register



Bit Address	Mnemonic	Description
A7	io7dmae_i	I/O channel 7 DMA enable [0]
A6	io6dmae_i	I/O channel 6 DMA enable [0]
A5	io5dmae_i	I/O channel 5 DMA enable [0]
A4	io4dmae_i	I/O channel 4 DMA enable [0]
A3	io3dmae_i	I/O channel 3 DMA enable [0]
A2	io2dmae_i	I/O channel 2 DMA enable [0]
A1	io1dmae_i	I/O channel 1 DMA enable [0]
A0	io0dmae_i	I/O channel 0 DMA enable [0]

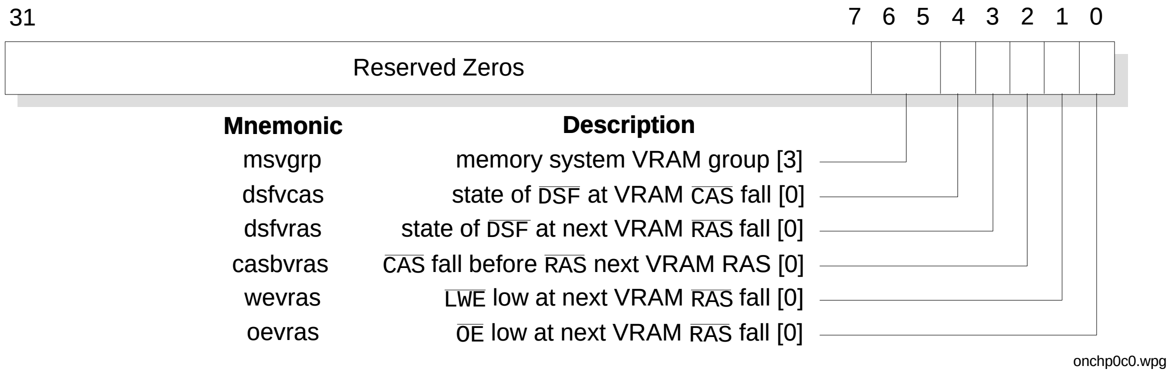
onchp0a0.wpg

Figure 31. DMA Enable Register

Allows a corresponding zero bit in `ioin` to request a DMA I/O-channel transfer for the corresponding I/O channel. When an enabled DMA request is recognized, the corresponding zero bit in `ioin` is set. If the corresponding `iodmaex` bit is set, the `iodmae` bit is cleared (to disable further DMA requests from that channel) when an I/O-channel transfer on that channel accesses the last location in a 1024-byte memory page. See *Direct Memory Access Controller*, page 103. When a `iodmae` bit is set, the corresponding `ioie` bit is ignored.

ADVANCE INFORMATION

C0 vram VRAM Control Bit Register



Mnemonic	Description
msvgrp	memory system VRAM group [3]
dsfvcas	state of $\overline{\text{DSF}}$ at VRAM $\overline{\text{CAS}}$ fall [0]
dsfvras	state of $\overline{\text{DSF}}$ at next VRAM $\overline{\text{RAS}}$ fall [0]
casbvras	$\overline{\text{CAS}}$ fall before $\overline{\text{RAS}}$ next VRAM RAS [0]
wevras	$\overline{\text{LWE}}$ low at next VRAM $\overline{\text{RAS}}$ fall [0]
oevras	$\overline{\text{OE}}$ low at next VRAM $\overline{\text{RAS}}$ fall [0]

Figure 32. VRAM Control Bit Register

These bits control the behavior of $\overline{\text{OE}}$, $\overline{\text{LWE}}$, the CASes, and DSF at $\overline{\text{RAS}}$ fall time; they also control the behavior of DSF at $\overline{\text{CAS}}$ fall time. They can be used in any combination to activate the various modes on VRAMs.

The bits from vram move through a hidden register prior to controlling the memory strobes during a subsequent MPU memory cycle. The bits stored for msvgrp in the hidden register determine which memory group is the current VRAM memory group, whose strobes are affected by the accompanying data in the hidden register. The hidden register is locked once data has been transferred into it from vram until an MPU access to the VRAM memory group occurs, thus consuming the data in the hidden register.

When a sto [] to vram occurs and the hidden register is not currently locked, the data from vram is transferred into the hidden register immediately if a posted write (to any memory group) is not waiting or in process, or at the end of the posted write if a posted write is waiting or in process. When a sto [] to vram occurs and the hidden register is already locked, the data in vram is not transferred (and is replaceable) until after the next access to the VRAM memory group occurs. The next access to the VRAM memory group uses the data in the hidden register, and when the memory access is complete, the data in vram is transferred to the hidden register.

Only MPU memory accesses have an effect on vram or the hidden register. Immediately after transferring vram to the hidden register, dsfvras, casbvras, wevras, and oevras in vram are cleared. After the VRAM group access, additional CAS or RAS cycles can occur on the VRAM memory group without rewriting the register, and use the current (cleared) vram data. When writes to vram are paired with one or more accesses to the VRAM memory group of the required RAS or CAS type, the internal operations described above are transparent to the program. Note that RAS precharge must be at least three CPU-clock cycles in duration for proper VRAM operation. See *Video RAM Support*, pages 37, 125, and 161.

msvgrp
Specifies the memory group containing the VRAM that is controlled by this register. VPU and MPU instructions must not be fetched from the memory group used for VRAM because the VRAM operations will likely occur on an instruction-fetch bus transaction rather than the intended VRAM transaction.

dsfvcas
Contains the state applied to DSF at the start of the next CAS-part of a memory cycle on the VRAM memory group. The bit is persistent and is *not* automatically cleared after being transferred to the hidden register. DSF is low when not accessing the VRAM memory group.

dsfvras

Contains the state applied to DSF two CPU-clock cycles after $\overline{\text{RAS}}$ rises during the next RAS cycle on the VRAM memory group. DSF changes to the dsfvcas state at the expiration of the row-address hold time. The bit is automatically cleared after being transferred to the hidden register.

casbvras

If set, during the next RAS cycle on the VRAM memory group all CAS signals are active two CPU-clock cycles after $\overline{\text{RAS}}$ rises, and are inactive at the normal expiration time. $\overline{\text{OE}}$, $\overline{\text{EWE}}$ and $\overline{\text{LWE}}$ go inactive at the expiration of the row-address hold time. The next access to the memory group msvgrp is forced by internal logic to be a RAS cycle.

Note that since all read and write strobes are inactive throughout their normally active times during the bus cycle, no data I/O with memory can occur. The data associated with the ST or LD used to cause the cycle is lost or undefined. The casbvras bit is automatically cleared after being transferred to the hidden register.

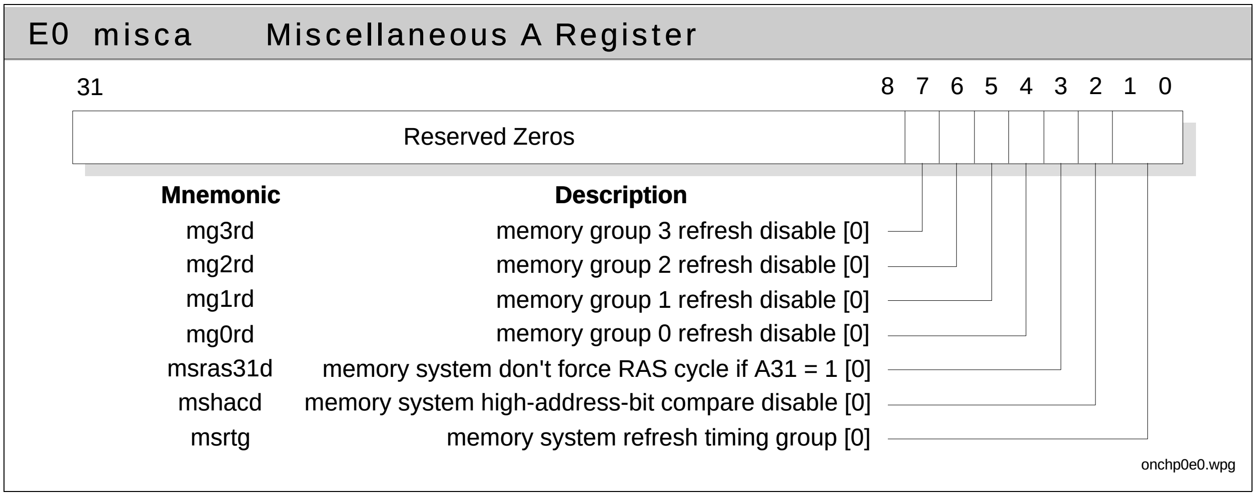
wevras

If set, $\overline{\text{LWE}}$ is low two CPU-clock cycles after $\overline{\text{RAS}}$ rises during the next RAS cycle on the VRAM memory group, and is high at the expiration of the row-address hold time. Otherwise, $\overline{\text{LWE}}$ is high until the expiration of the row-address hold time during the next RAS cycle on the VRAM memory group. In either case, during the CAS portion of the cycle $\overline{\text{LWE}}$ behaves normally and the data transferred is part of the function performed. The bit is automatically cleared after being transferred to the hidden register.

oevras

If set, $\overline{\text{OE}}$ is low two CPU-clock cycles after $\overline{\text{RAS}}$ rises during the next RAS cycle on the VRAM memory group, and is high at the expiration of the row-address hold time. Otherwise, $\overline{\text{OE}}$ is high until the expiration of the row-address hold time during the next RAS cycle on the VRAM memory group. In either case, during the CAS portion of the cycle $\overline{\text{OE}}$ behaves normally and the data transferred is part of the function performed. The bit is automatically cleared after being transferred to the hidden register.

ADVANCE INFORMATION



Mnemonic	Description
mg3rd	memory group 3 refresh disable [0]
mg2rd	memory group 2 refresh disable [0]
mg1rd	memory group 1 refresh disable [0]
mg0rd	memory group 0 refresh disable [0]
msras31d	memory system don't force RAS cycle if A31 = 1 [0]
mshacd	memory system high-address-bit compare disable [0]
msrtg	memory system refresh timing group [0]

Figure 33. Miscellaneous A Register

`mgXrd`
Allows (if clear) or prevents (if set) a refresh cycle from occurring on the corresponding memory group when `refresh` executes. Allowing refresh on some memory groups can be undesirable or inappropriate. For example, the primary side effect of refresh is that the current row address latched in the memory device is changed. This can be undesirable on VRAM devices when a RAS cycle sets persistent operational modes and addresses. Another refresh side effect is that the next memory cycle to the memory group is a RAS cycle to re-select the operational memory row. This is usually undesirable in SRAM because refresh is not required; the refresh and RAS cycles only slow execution, or make otherwise predictable timing unpredictable.

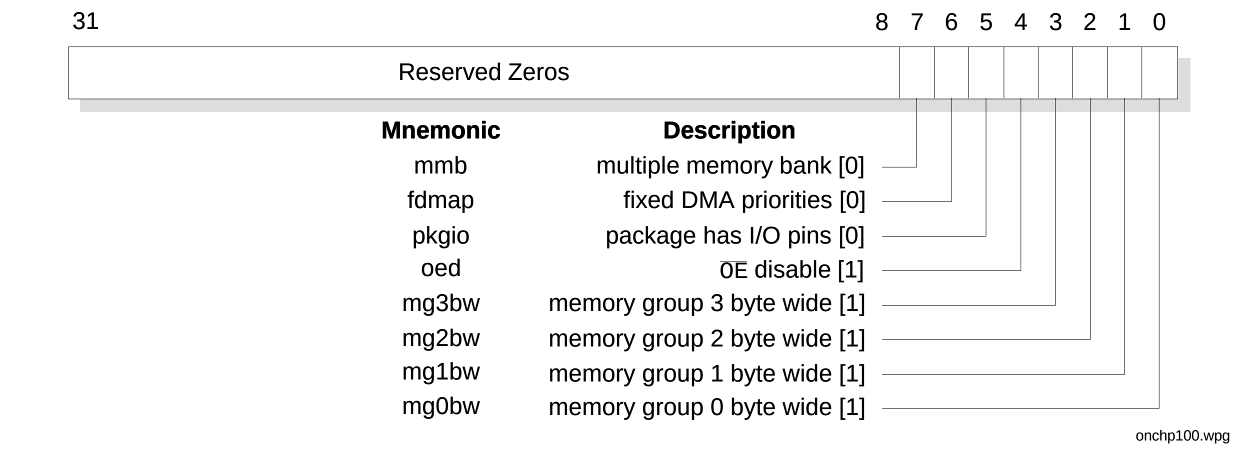
`msras31d`
If set, allows non-RAS cycles when A31 is a one. If clear, forces a RAS cycle on both one-bus-cycle transactions and the first cycle of four-bus-cycle byte transactions when A31 is a one. In large memory systems in which the I/O-device addressing bits overlap the group, bank, or DRAM RAS bits, this option forces a RAS cycle when one might not otherwise occur because these various bits either are excluded from the RAS comparison logic or could inadvertently match the I/O-device address bits. RAS cycles might be required by system design to allow enough time for I/O decode and select. A31 is used in selecting I/O addresses.

`mshacd`
If clear, enables the comparison of the high address bits to those of the most recent RAS cycle to determine if a RAS cycle must occur. If set, disables this comparison. These bits are typically used for I/O addresses that require external decoding logic which might require the additional time available in a RAS cycle for this decoding. However, with high-speed logic it is often possible to decode the I/O address in the time available within a CAS cycle, thus speeding I/O access. A31 can be excluded from the high-address-bit compare by setting `msexa31hac`.

`msrtg`
Contains the number of the memory group whose RAS cycle timing is to be used for refresh cycles produced by `refresh`. The memory group specified must be the group with the most-restrictive (slowest) refresh timing.

ADVANCE INFORMATION

100 miscb Miscellaneous B Register



Mnemonic	Description
mmb	multiple memory bank [0]
fdmap	fixed DMA priorities [0]
pkgio	package has I/O pins [0]
oed	$\overline{\text{OE}}$ disable [1]
mg3bw	memory group 3 byte wide [1]
mg2bw	memory group 2 byte wide [1]
mg1bw	memory group 1 byte wide [1]
mg0bw	memory group 0 byte wide [1]

onchp100.wpg

Figure 34. Miscellaneous B Register

`mmb`

If clear, selects Single Memory Bank (SMB) mode for all memory groups. $\overline{\texttt{RASx}}$ signals appear on the corresponding package pins. Bank-select bits correspond with the `msgsm` bits. Up to four memory banks (i.e., one memory bank per memory group) can be directly connected and accessed. See Figure 21, page 118.

If set, selects Multiple Memory Bank (MMB) mode for all memory groups. $\overline{\texttt{MGSx}}$ signals appear on the corresponding package pins. Bank-select bits are located immediately above the DRAM RAS bits, or for SRAM in the `mssbs` location. Up to sixteen memory banks (i.e., four banks per memory group) can be connected with 1.25 two-input gates per bank. With additional inputs per gate and additional decoding, an arbitrarily large number of memory banks can easily be connected. See Figure 22, page 119.

`fdmap`

DMA requests contend for the bus; the highest-priority request gets the first chance at access. If `vpudelay` is large enough to allow bus access by the highest-priority request, the bus is granted to the device.

If `fdmap` is set and `vpudelay` is too small for the highest-priority DMA request, the DMA request does not get the bus. Unless a higher-priority DMA request occurs that fits the shrinking available bus slot, no bus transactions occur until the VPU seizes the bus. When the VPU next executes `delay`, the highest-priority DMA request—or the MPU if there are no DMA requests—repeats the bus request process.

If `fdmap` is clear and `vpudelay` is too small for the highest-priority DMA request, the request does not get the bus. The next lower-priority bus request is then allowed to request the bus, with the MPU as the lowest-priority request. The process repeats until the bus is granted or the VPU seizes the bus. When the VPU next executes `delay`, the highest-priority DMA request—or the MPU if there are no DMA requests—repeats the bus request process.

`pkgio`

If set, inputs to `ioin` are taken from $\overline{\texttt{IN}}$`[7:0]`. If clear, inputs are taken from `AD[7:0]` when $\overline{\texttt{RAS}}$ is low and $\overline{\texttt{CAS}}$ is high. See *Bit Inputs*, page 111.

`oed`

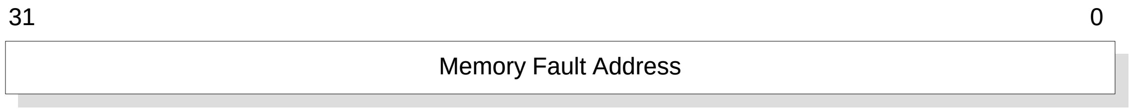
If set, disables $\overline{\texttt{OE}}$ from going active during bus cycles. If clear, $\overline{\texttt{OE}}$ behaves normally. On CPU reset, the $\overline{\texttt{OE}}$ signal is disabled to prevent conventionally connected memory from responding; this allows booting from a device in I/O space. See *Processor Startup*, page 181.

ADVANCE INFORMATION

mgXbw

If clear, the corresponding memory group is cell-wide and is read and written 32-bits per bus cycle. If set, the corresponding memory group is byte-wide and is read and written in a single bus transaction of four bus cycles, one byte per cycle.

120 mfltaddr Memory Fault Address Register



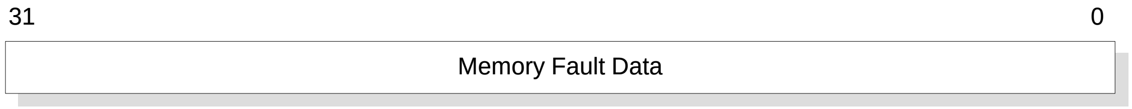
Register is read-only. Reading `mfltaddr` after a memory fault releases the data lock on `mfltaddr` and `mfltdata`, allowing data to flow into the registers. [0]

onchp120.wpg

Figure 35. Memory Fault Address Register

When a memory page-fault exception occurs during a memory read or write, `mfltaddr` contains the address that caused the exception. The contents of `mfltaddr` and `mfltdata` are latched until the first read of `mfltaddr` after the fault. After reading `mfltaddr`, the data in `mfltaddr` and `mfltdata` are no longer valid.

140 mfltdata Memory Fault Data Register

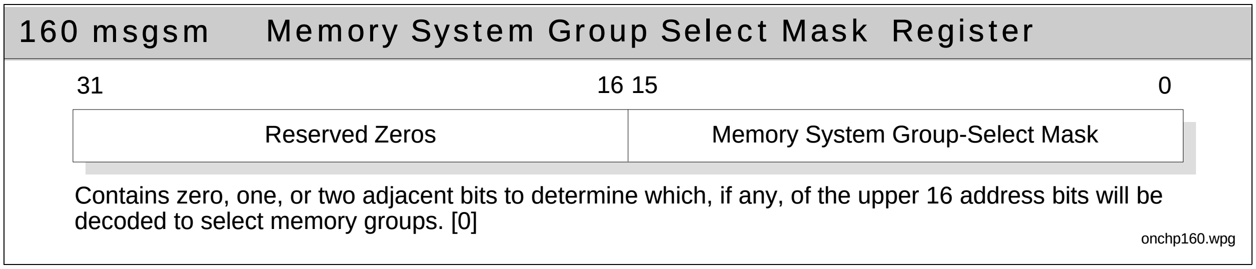


Register is read-only. Reading `mfltaddr` after a memory fault releases the data lock on `mfltaddr` and `mfltdata`, allowing data to flow into the registers. [0]

onchp140.wpg

Figure 36. Memory Fault Data Register

When a memory page-fault exception occurs during a memory write, `mfltdata` contains the data to be stored at `mfltaddr`. The contents of `mfltdata` and `mfltdata` are latched until the first read of `mfltaddr` after the fault.


160 msgsm Memory System Group Select Mask Register
31
16 15
0
Reserved Zeros
Memory System Group-Select Mask

Contains zero, one, or two adjacent bits to determine which, if any, of the upper 16 address bits will be decoded to select memory groups. [0]

onchp160.wpg

Figure 37. Memory System Group-Select Mask Register

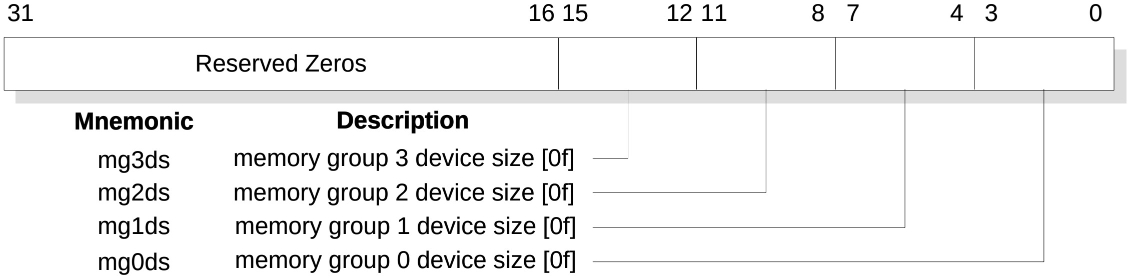
Contains zero, one, or two adjacent bits that locate the memory group-select bits between A16 and A31.

When no bits are set, all memory accesses occur in memory group zero. The memory system high address bits occur in the address bits: for DRAM, above the memory group zero DRAM RAS address; for SRAM, above A21.

When one bit is set, it determines the address bit that selects accesses between memory group zero and memory group one. The memory system high address bits occur in the address bits higher than the bit selected, but always include A31.

When two adjacent bits are set, they are decoded to select one of four memory groups that is accessed. The memory system high address bits occur in the address bits higher than the bits selected, but always include A31.

180 mgds Memory Group Device Size Register



Mnemonic	Description
mg3ds	memory group 3 device size [0f]
mg2ds	memory group 2 device size [0f]
mg1ds	memory group 1 device size [0f]
mg0ds	memory group 0 device size [0f]

Device Sizes

0x00	64K DRAM	0x04	1M DRAM	0x08	8M DRAM	0x0c	64M DRAM†
0x01	128K DRAM	0x05	2M DRAM	0x09	16M DRAM†	0x0d	64M DRAM
0x02	256K DRAM	0x06	4M DRAM†	0x0a	16M DRAM	0x0e	128M DRAM
0x03	512K DRAM	0x07	4M DRAM	0x0b	32M DRAM	0x0f	SRAM

† Asymmetric addressing, the number of RAS and CAS address bits differ.

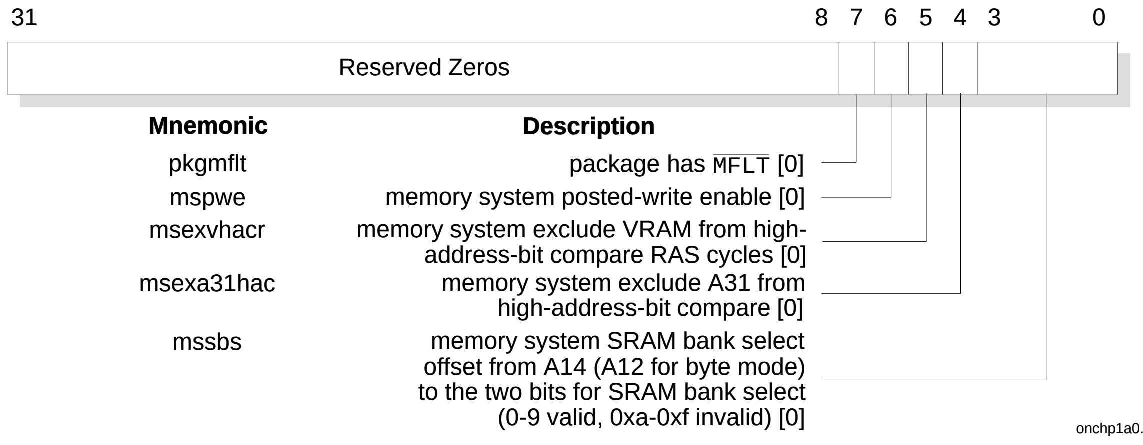
onchp180.wpg

Figure 38. Memory Group Device Size Register

Contains 4-bit codes that select the DRAM address bit configuration, or SRAM, for each memory group. The code determines which bits are used during RAS and CAS addressing and which bits are compared to determine if a RAS cycle is required (due to the DRAM row address changing). See Table 51, page 122, and Table 52, page 123.

ADVANCE INFORMATION

1A0 miscc Miscellaneous C Register



Mnemonic	Description
pkgmflt	package has $\overline{\text{MFLT}}$ [0]
mspwe	memory system posted-write enable [0]
msexvhacr	memory system exclude VRAM from high-address-bit compare RAS cycles [0]
msexa31hac	memory system exclude A31 from high-address-bit compare [0]
mssbs	memory system SRAM bank select offset from A14 (A12 for byte mode) to the two bits for SRAM bank select (0-9 valid, 0xa-0xf invalid) [0]

Figure 39. Miscellaneous C Register

`pkgmflt`
If set, the memory-fault input is sampled from $\overline{\text{MFLT}}$. If clear, the memory-fault input is sampled from AD8 when $\overline{\text{RAS}}$ falls. See Figure 77, page 212.

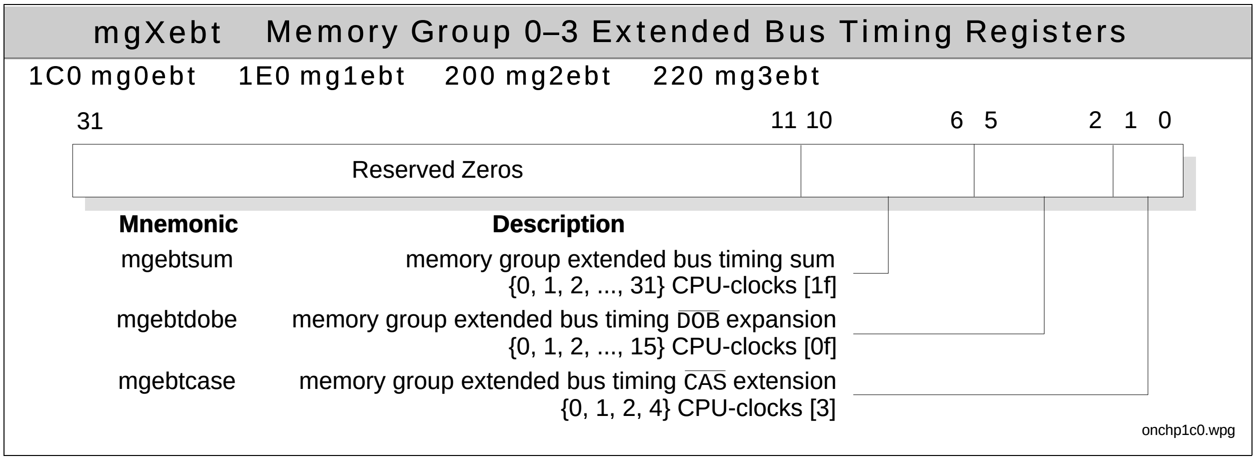
`mspwe`
If set, enables a one-level MPU posted-write buffer, which allows the MPU to continue executing after a write to memory occurs. A posted write has precedence over subsequent MPU reads to maintain memory coherency. If clear, the MPU must wait for writes to complete before continuing.

`msexvhacr`
If set, RAS cycles do not occur in the memory group `msvgrp` when due to a high-address-bit comparison. This prevents unexpected RAS cycles (typically caused by a DMA or VPU initiated bus transaction) from causing a VRAM operation.

`msexa31hac`
If set, A31 is not included in the high-address-bit compare. If clear, A31 is included in the high-address-bit compare. See `mshacd` for more information. The high address bits are typically used for I/O addresses, and require external decoding logic that might require the additional time available in a RAS cycle for decoding. Some bus transactions contain adjacent bus cycles whose high address bits differ by only the state of A31, and could thus require a RAS cycle due solely to the change in this bit. However, some system designs can decode the A31 change in the time available in a CAS cycle, thus speeding I/O access. If this bit is set a RAS cycle does not occur if only address bit A31 changes.

`mssbs`
For multiple memory bank mode only, these bits contain the offset from A14 (A12 for a byte-mode group) to the two address bits used to select banks within any memory group containing SRAM devices. Typically set to place the bits immediately above the address bits of the SRAM devices used.

ADVANCE INFORMATION



mgXebt Memory Group 0–3 Extended Bus Timing Registers

1C0 mg0ebt 1E0 mg1ebt 200 mg2ebt 220 mg3ebt

Mnemonic	Description
mgebtsum	memory group extended bus timing sum {0, 1, 2, ..., 31} CPU-clocks [1f]
mgebtdobe	memory group extended bus timing $\overline{DOB}$ expansion {0, 1, 2, ..., 15} CPU-clocks [0f]
mgebtcase	memory group extended bus timing $\overline{CAS}$ extension {0, 1, 2, 4} CPU-clocks [3]

Figure 40. Memory Group 0–3 Extended Bus Timing Registers

These values compensate for propagation, turn-on, turn-off, and other delays in the memory system. They are specified separately for each memory group. When an I/O-channel bus transaction occurs, the I/O-channel extension, `ioXebt`, is substituted for the corresponding value. The I/O-channel extensions must be sufficient for any memory group into which that I/O channel might transfer.

`mgebtsum`

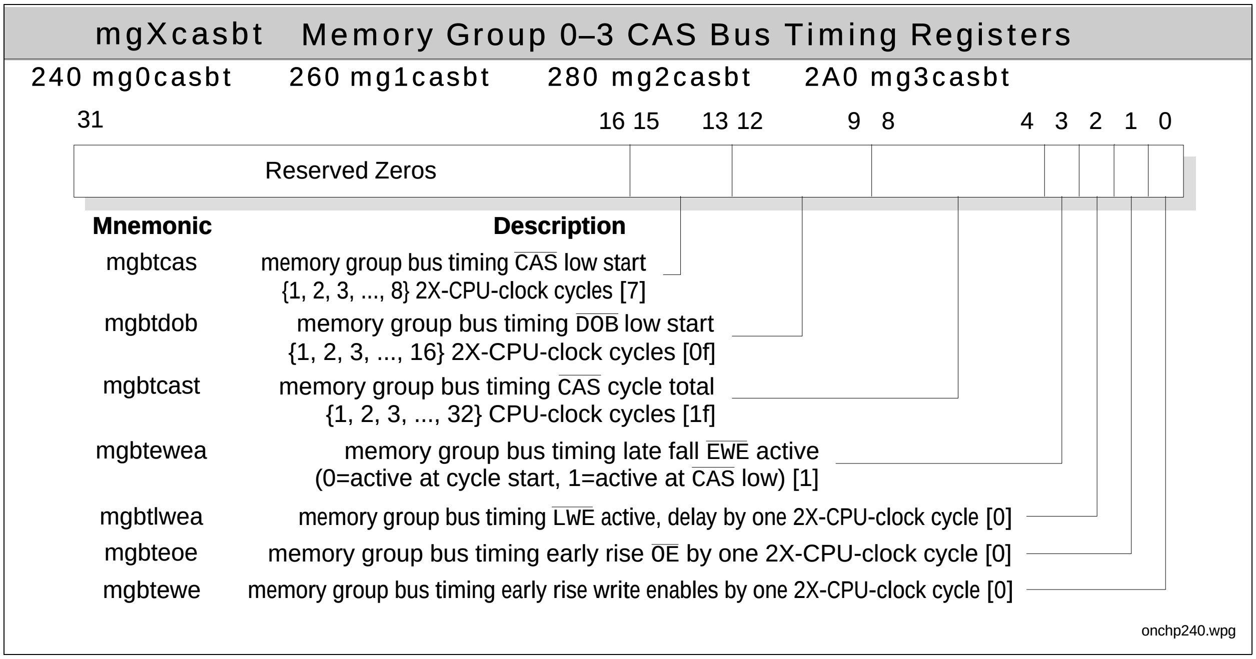
Programmed to contain the sum of `mgebtcase` and `mgebtdobe`. This value is used only during the slot check to compute the total time required for the bus cycle.

`mgebtdobe`

Expands the CAS cycle at $\overline{DOB}$ fall by the specified time. This parameter is used to compensate for memory group buffer delays, device access time, and other operational requirements. If the bus cycle is a memory read cycle, $\overline{OE}$ is expanded. If the bus cycle is a memory write cycle, $\overline{EWE}$ is expanded and $\overline{LWE}$ fall is delayed the specified time.

`mgebtcase`

Extends the CAS cycle by the specified amount after the unextended CAS time. $\overline{DOB}$, $\overline{OE}$, $\overline{EWE}$ and $\overline{LWE}$ rise unextended. This parameter is used to allow for data hold times or to allow for devices to disable their output drivers. When used in combination with `mgbtewe` or `mgbteoe`, hold or disable times can be set in most increments of 2X-CPU-clock cycles.



Mnemonic	Description
mgbtcas	memory group bus timing $\overline{\text{CAS}}$ low start {1, 2, 3, ..., 8} 2X-CPU-clock cycles [7]
mgbtdob	memory group bus timing $\overline{\text{DOB}}$ low start {1, 2, 3, ..., 16} 2X-CPU-clock cycles [0f]
mgbtcast	memory group bus timing $\overline{\text{CAS}}$ cycle total {1, 2, 3, ..., 32} CPU-clock cycles [1f]
mgbtewea	memory group bus timing late fall $\overline{\text{EWE}}$ active (0=active at cycle start, 1=active at $\overline{\text{CAS}}$ low) [1]
mgbtlwea	memory group bus timing $\overline{\text{LWE}}$ active, delay by one 2X-CPU-clock cycle [0]
mgbteoe	memory group bus timing early rise $\overline{\text{OE}}$ by one 2X-CPU-clock cycle [0]
mgbtewe	memory group bus timing early rise write enables by one 2X-CPU-clock cycle [0]

Figure 41. Memory Group 0–3 CAS Bus Timing Registers

Defines the basic timing for CAS-only cycles and the CAS portion of RAS cycles. Timing is specified separately for each memory group. The values that refer to $\overline{\mathtt{CAS}}$ apply to CAS, $\overline{\mathtt{CAS0}}$, $\overline{\mathtt{CAS1}}$, $\overline{\mathtt{CAS2}}$ and $\overline{\mathtt{CAS3}}$, appropriately. The basic CAS cycle timing is augmented by `mgXebt` and `ioXebt` values.

`mgbtcas`
Specifies the CAS-cycle precharge time, the time from the start of the CAS-timed portion of the memory cycle until $\overline{\mathtt{CAS}}$ goes low.

`mgbtdob`
Specifies the end of address time (column address hold) and the beginning of data time on the bus relative to the start of the CAS portion of the memory cycle. This is the time the CPU places write data on the bus or begins accepting read data from the bus.

`mgbtcast`
Specifies the total unexpanded and unextended time of a CAS cycle. $\overline{\mathtt{DOB}}$, $\overline{\mathtt{OE}}$, $\overline{\mathtt{EWE}}$ and $\overline{\mathtt{LWE}}$ rise at this time unless modified by `mgbteoe` or `mgbtewe`. This time value is also used during the slot check to compute the total time required for the bus cycle.

`mgbtewea`
In a system with fast SRAM, $\overline{\mathtt{EWE}}$ fall at cycle start is required to have an adequate write enable. Other devices require their addresses to be valid before write enable falls; in these cases $\overline{\mathtt{CAS}}$ low is required.

`mgbtlwea`
Specifies a delay of zero or one 2X-CPU-clock cycle after $\overline{\mathtt{DOB}}$ fall plus expansion for $\overline{\mathtt{LWE}}$ fall. Expansion refers to the value of `mgebtdobe` or `ioebtdobe`, as appropriate. Allows adjustment for system and device delays. For example, DRAM expects data valid at its write-enable fall. In small systems $\overline{\mathtt{DOB}}$ plus one 2X-CPU-clock cycle (with an expansion of zero) might be appropriate. In a large system with a heavily loaded (or buffered) $\overline{\mathtt{LWE}}$, $\overline{\mathtt{DOB}}$ might be appropriate for the fastest memory cycle. If a larger delay is required, an expansion value can be set. Allows resolution of one 2X-CPU-clock cycle in expansion timing.

`mgbteoe`

If set, $\overline{\texttt{OE}}$ rises one 2X-CPU-clock cycle before the end of the unextended CAS cycle. If clear, $\overline{\texttt{OE}}$ rises with the end of the unextended CAS cycle. One 2X-CPU-clock cycle is sufficient output-driver disable time for some devices; if not, output-driver disable time can be created in most increments of 2X-CPU-clock cycles by combining `mgebtcase` and `mgbteoe`.

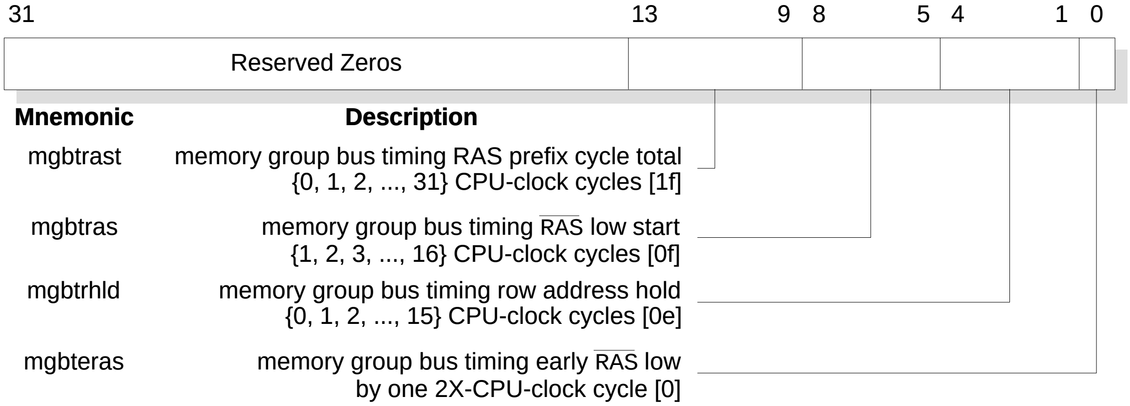
`mgbtewe`

If set, $\overline{\texttt{EWE}}$ and $\overline{\texttt{LWE}}$ rise one 2X-CPU-clock cycle before the end of the unextended CAS cycle. If clear, $\overline{\texttt{EWE}}$ and $\overline{\texttt{LWE}}$ rise with the end of the unextended CAS cycle. One 2X-CPU-clock cycle is sufficient hold time for some devices; if not, hold time can be created in most increments of 2X-CPU-clock cycles by combining `mgebtcase` and `mgbtewe`.

ADVANCE INFORMATION

mgXrasbt Memory Group 0–3 RAS Bus Timing Registers

2C0 mg0rasbt 2E0 mg1rasbt 300 mg2rasbt 320 mg3rasbt



Mnemonic	Description
mgbtrast	memory group bus timing RAS prefix cycle total {0, 1, 2, ..., 31} CPU-clock cycles [1f]
mgbtras	memory group bus timing $\overline{\text{RAS}}$ low start {1, 2, 3, ..., 16} CPU-clock cycles [0f]
mgbtrhld	memory group bus timing row address hold {0, 1, 2, ..., 15} CPU-clock cycles [0e]
mgbteras	memory group bus timing early $\overline{\text{RAS}}$ low by one 2X-CPU-clock cycle [0]

onchp2c0.wpg

Figure 42. Memory Group 0–3 RAS Bus Timing Registers

Defines the timing for the RAS-prefix portion a of RAS memory cycle. Timing is specified separately for each memory group. The values are selected as required for the memory devices used. Timing values that refer to $\overline{\text{RAS}}$ apply to RAS, $\overline{\text{RAS0}}$, $\overline{\text{RAS1}}$, $\overline{\text{RAS2}}$ and $\overline{\text{RAS3}}$, appropriately.

`mgbtrast`

Programmed to contain the sum of the decoded number of CPU-clock cycles represented in `mgbtras` and `mgbtrhld` plus one. At the end of this time the CAS portion of the memory cycle begins. This value is used only during the slot check to compute the total time required for the bus cycle.

`mgbtras`

Specifies the RAS precharge time, the time $\overline{\text{RAS}}$ is high at the beginning of a RAS cycle. The time can be shortened with `mgbteras`.

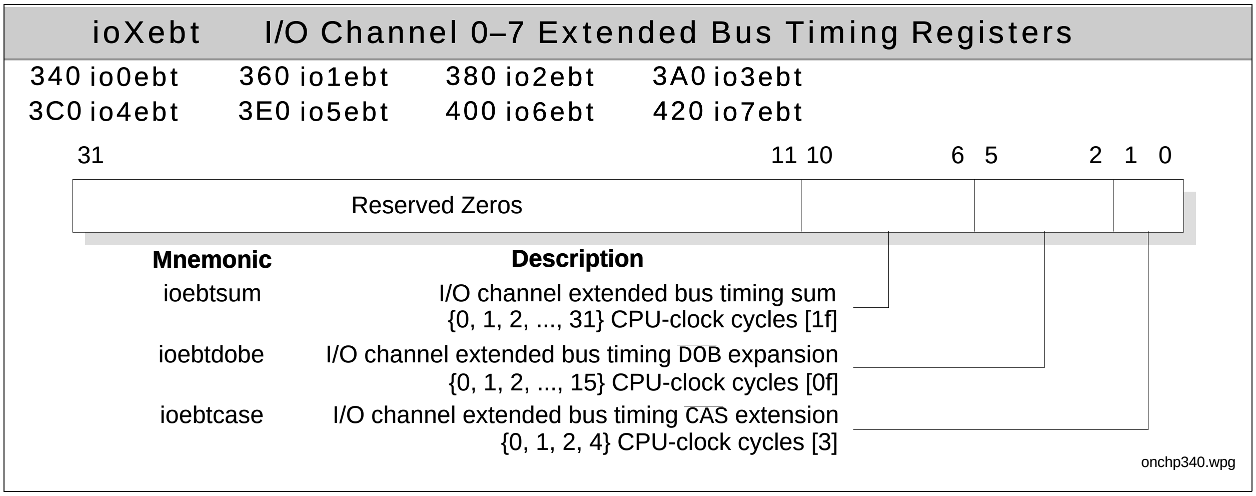
`mgbtrhld`

Specifies the row-address hold time of a RAS cycle, immediately preceding the CAS timing portion of the cycle. The time can be lengthened with `mgbteras`. Immediately following this time the CAS address is placed on the bus, if appropriate.

`mgbteras`

If set, reduces the RAS precharge time (specified by `mgbtras`) and extends the row-address hold time (specified by `mgbtrhld`) by one 2X-CPU-clock cycle.

ADVANCE INFORMATION



ioXebt I/O Channel 0–7 Extended Bus Timing Registers

340 io0ebt	360 io1ebt	380 io2ebt	3A0 io3ebt
3C0 io4ebt	3E0 io5ebt	400 io6ebt	420 io7ebt

Mnemonic	Description
ioebtsum	I/O channel extended bus timing sum {0, 1, 2, ..., 31} CPU-clock cycles [1f]
ioebtdobe	I/O channel extended bus timing $\overline{DOB}$ expansion {0, 1, 2, ..., 15} CPU-clock cycles [0f]
ioebtcase	I/O channel extended bus timing $\overline{CAS}$ extension {0, 1, 2, 4} CPU-clock cycles [3]

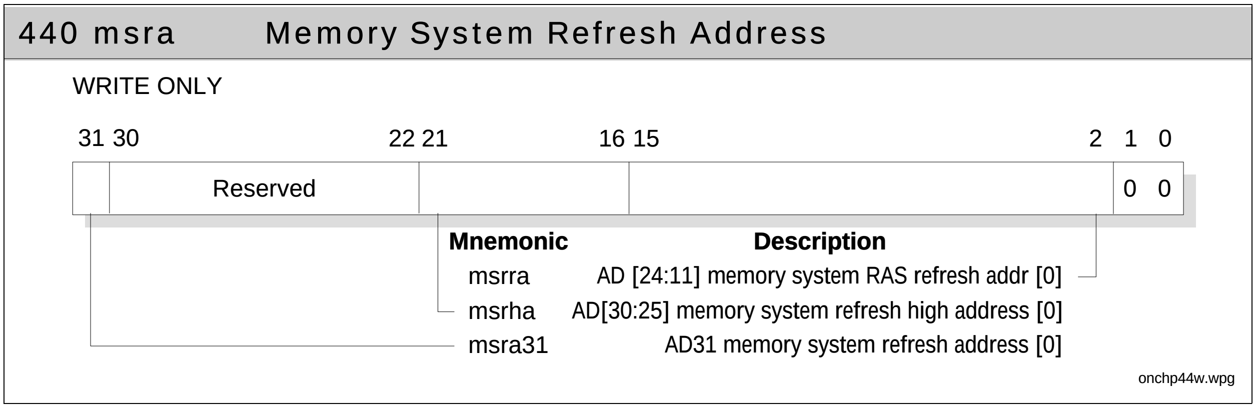
Figure 43. I/O Channel 0–7 Extended Bus Timing Registers

These values compensate for signal propagation, turn-on, turn-off, device, and other delays in the memory and I/O systems. They are substituted for the memory group values, mgXebt, during I/O channel transfers and thus must be sufficient for the I/O device, as well as any memory group with which the I/O device will transfer.

ioebtsum
Programmed to contain the sum of ioebtcase and ioebtdobe. This value is used only during the slot check to compute the total time required for the bus cycle.

ioebtdobe
Expands the CAS cycle at $\overline{DOB}$ fall by the specified time. This parameter is used to compensate for memory group buffer delays, device access time, and other operational requirements. If the bus cycle is a memory read cycle, $\overline{OE}$ is expanded. If the bus cycle a is memory write cycle, $\overline{EWE}$ is expanded and $\overline{LWE}$ fall is delayed the specified time.

ioebtcase
Extends the CAS cycle by the specified amount after the unextended CAS time. $\overline{DOB}$, $\overline{OE}$, $\overline{EWE}$ and $\overline{LWE}$ rise unextended. This parameter is used to allow for data hold times or to allow for devices to disable their output drivers. When used in combination with mgbtewe or mgbteoe, hold or disable times can be set in most increments of 2X-CPU-clock cycles.



Mnemonic	Description
msrra	AD [24:11] memory system RAS refresh addr [0]
msrha	AD[30:25] memory system refresh high address [0]
msra31	AD31 memory system refresh address [0]

Figure 44. Memory System Refresh Address

Contains the next address used for memory-system refresh. The values are placed on the specified pins when `refresh` executes, and `msrra` is incremented by one. The timing for a refresh cycle is set by `msrtg`, and those memory groups that are refreshed are set by `mgXrd`.

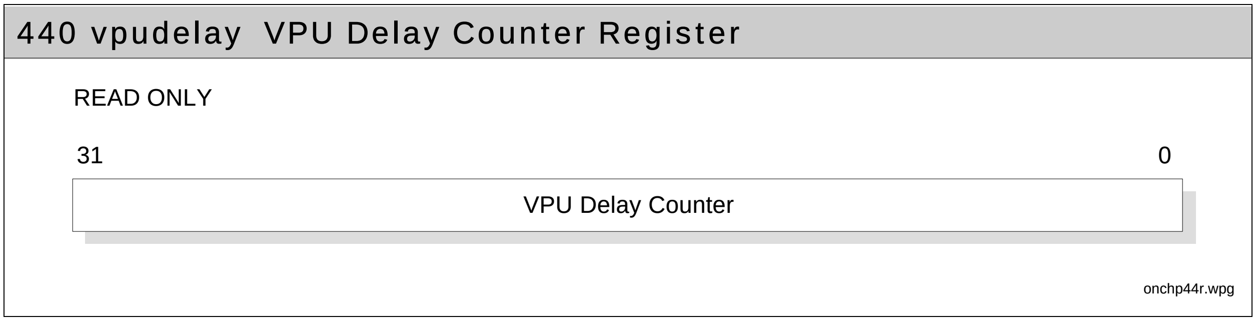


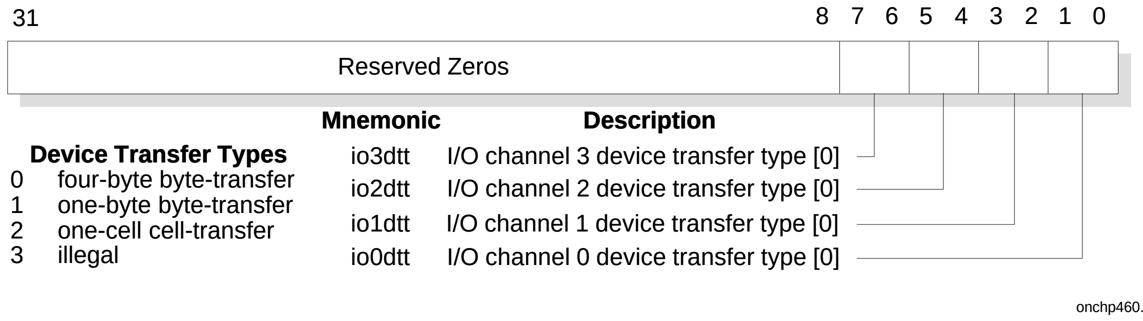
Figure 45. VPU Delay Counter Register

Contains the number of CPU-clock cycles until the VPU seizes the bus. The counter is decremented once each CPU-clock cycle. The counter can be used, for example, to determine if a time-critical task can be completed before the VPU seizes the bus, or to measure time in CPU-clock increments.

ADVANCE INFORMATION

ADVANCE INFORMATION

460 iodtta I/O Device Transfer Types A Register

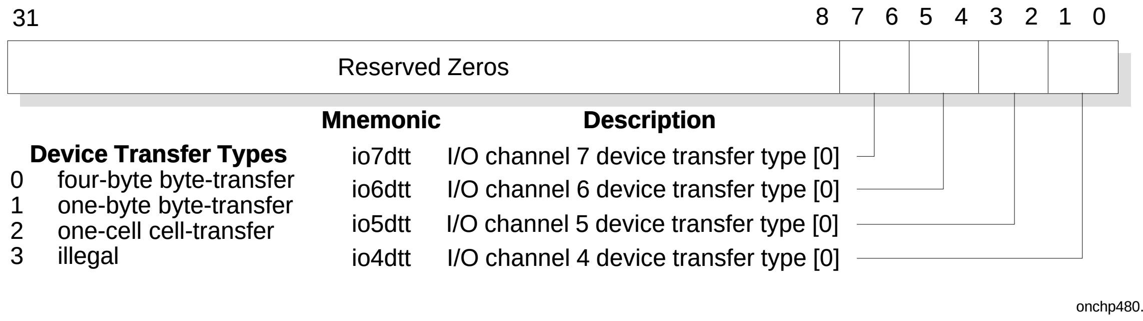


Device Transfer Types	
0	four-byte byte-transfer
1	one-byte byte-transfer
2	one-cell cell-transfer
3	illegal

Mnemonic	Description
io3dtt	I/O channel 3 device transfer type [0]
io2dtt	I/O channel 2 device transfer type [0]
io1dtt	I/O channel 1 device transfer type [0]
io0dtt	I/O channel 0 device transfer type [0]

Figure 46. I/O Device Transfer Types A Register

480 iodttb I/O Device Transfer Types B Register



Device Transfer Types	
0	four-byte byte-transfer
1	one-byte byte-transfer
2	one-cell cell-transfer
3	illegal

Mnemonic	Description
io7dtt	I/O channel 7 device transfer type [0]
io6dtt	I/O channel 6 device transfer type [0]
io5dtt	I/O channel 5 device transfer type [0]
io4dtt	I/O channel 4 device transfer type [0]

Figure 47. I/O Device Transfer Types B Register

Specifies one of three transfer types for the device attached to the corresponding I/O channel.

- Four-Byte Byte-Transfer Type: Transfers four bytes of data, one byte at a time, between the device and memory in a single bus transaction. The transaction consists of four bus cycles accessing the device, plus one additional bus cycle to access memory if the memory is cell-wide. All initial transfer addresses are to cell boundaries.
- One-Byte Byte-Transfer Type: Transfers one byte of data between the device and memory in a single bus transaction. The transaction consists of a single bus cycle. Transfers to cell-wide memory are to byte zero of the addressed cell, with the remaining 24 bits undefined. Transfers to byte-wide memory are to the specified byte.
- One-Cell Cell-Transfer Type: Transfers one cell of data between the device and memory in a single bus transaction. The transaction consists of one bus cycle to access the device, plus four additional bus cycles to access memory if the memory is byte-wide. All initial transfers are to cell boundaries.

Reserved Register Addresses

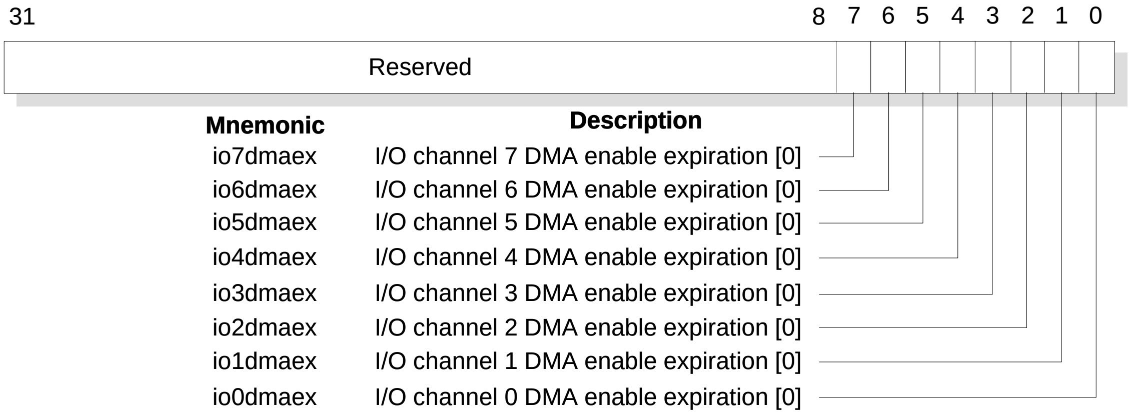
4A0-780

onchp4a0.wpg

Figure 48. Reserved Register Addresses

These addresses are reserved.

7A0 iodmaex DMA Enable Expiration Register



Mnemonic	Description
io7dmaex	I/O channel 7 DMA enable expiration [0]
io6dmaex	I/O channel 6 DMA enable expiration [0]
io5dmaex	I/O channel 5 DMA enable expiration [0]
io4dmaex	I/O channel 4 DMA enable expiration [0]
io3dmaex	I/O channel 3 DMA enable expiration [0]
io2dmaex	I/O channel 2 DMA enable expiration [0]
io1dmaex	I/O channel 1 DMA enable expiration [0]
io0dmaex	I/O channel 0 DMA enable expiration [0]

onchp7a0.wpg

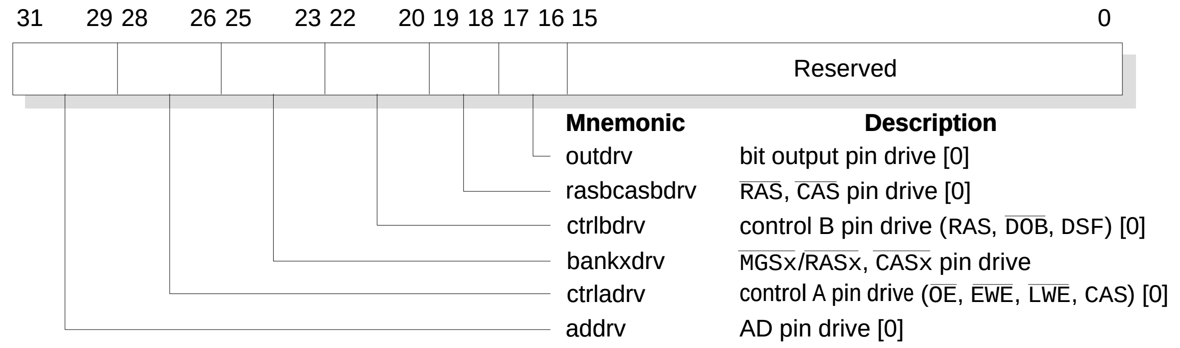
Figure 49. DMA Enable Expiration Register

Clears the corresponding DMA enable bit in `iodmae` after a DMA I/O channel transfer is made to the last location in a 1024-byte memory page. This allows DMA on the corresponding I/O channel to be disabled after transferring a predetermined number of bytes. See *Direct Memory Access Controller*, page 103.

ADVANCE INFORMATION

ADVANCE INFORMATION

7C0 drivers Driver Current Register



Mnemonic	Description
outdrv	bit output pin drive [0]
rasbcasbdrv	$\overline{\text{RAS}}$, $\overline{\text{CAS}}$ pin drive [0]
ctrlbdrv	control B pin drive (RAS, $\overline{\text{DOB}}$, DSF) [0]
bankxdrv	$\overline{\text{MGSx/RASx}}$, $\overline{\text{CASx}}$ pin drive
ctrladrv	control A pin drive ($\overline{\text{OE}}$, $\overline{\text{EWE}}$, $\overline{\text{LWE}}$, CAS) [0]
addrv	AD pin drive [0]

3-Bit Field

00n	1 of 3 drivers
01n	2 of 3 drivers
11n	3 of 3 drivers

2-Bit Field

0n	1 of 3 drivers
1n	2 of 3 drivers

Where n =

0	1 of 2 pre-drivers
1	2 of 2 pre-drivers

onchp7c0.wpg

Figure 50. Driver Current Register

Allows programming the relative amount of current available to drive the various signals out of the package. The programmed driver current has several effects.

- The amount of current selected determines the rise and fall times of the signals into a given load. The rise and fall times, PWB wire lengths, and PWB construction determine whether the signals are to be treated as transmission lines, and whether signal terminations are required.
- The rise and fall of signals affects bus cycle timing since signal switching consumes time. Slower rise and fall times might require a slower bus cycle.
- Greater driver current increases di/dt, and thus increases package and system electrical noise. Though total power consumption does not change when driver current is changed (since the same load is charged, just slower or faster), there is less noise produced when di/dt is decreased. Reducing output driver pre-driver current also reduces package and system electrical noise, and can thus facilitate approval of electromagnetic compliance for products.

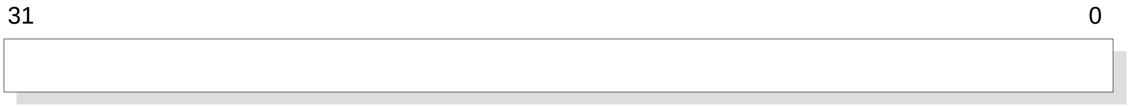
Programmable drivers allow the system designer to trade among system design complexity, system cost, and system performance.

Output drivers consist of a pre-driver and an output driver. The current-supply capability of each part of the output driver can be programmed separately. The low bit of each field selects full- or half-drive capability on the pre-drivers for that set of signals. The upper one or two bits select 1/3-, 2/3- or full-drive capability.

The pre-drivers are supplied by the core logic power, and the noise generated by their operation can affect the performance of the CPU in systems with an inadequate power supply or decoupling. In such systems, lowering pre-driver current can possibly compensate for system design flaws.

The drivers are on two separate power buses: one for AD and one for control signals and all other output pins. As a result, inside the package, electrical noise caused by AD driver switching is prevented from corrupting the quality of the control signals. This separation, however, does not preclude noise coupling between the power pins outside the package. Depending on system loading, the output drivers account for 50% to 95% of the power consumed by the CPU, and thus are a potentially large noise source.

7E0 vpureset VPU Reset Register



write	reset VPU on any write
read	0ffffffff while waiting to reset, zero otherwise

onchp7e0.wpg

Figure 51. VPU Reset Register

Writing any value causes the VPU to begin executing at its software reset executable-code vector (location 0x00000010) at the end of the current memory cycle. This is the mechanism used to clear bit 31 in the VPU PC after hardware reset, and to direct the VPU to execute a new procedure. The value of the register is -1 during the VPU reset process (i.e., from the time `vpureset` is written until the VPU begins execution of the software reset executable-code vector); otherwise, its value is zero.

ADVANCE INFORMATION

Table 54. Bit Field to On-Chip Register Cross-Reference

Bit field	Register	Bit field	Register	Bit-field	Register
addrv	drivers	ioXout_i	ioout	mmb	miscb
bankxdrv	drivers	mfltaddr	mfltaddr	msexa31hac	miscc
casbvras	vram	mfltdata	mfltdata	msexvhacr	miscc
ctrladrv	drivers	mgbtcas	mgXcasbt	msgsm	msgsm
ctrlbdrv	drivers	mgbtcast	mgXcasbt	mshacd	misca
dsfvcas	vram	mgbtdob	mgXcasbt	mspwe	miscc
dsfvras	vram	mgbteoe	mgXcasbt	msra31	msra
fdmap	miscb	mgbteras	mgXrasbt	msras31d	misca
ioebtcase	ioXebt	mgbtewe	mgXcasbt	msrha	msra
ioebtdobe	ioXebt	mgbtewea	mgXcasbt	msrra	msra
ioebtsum	ioXebt	mgbtlwea	mgXcasbt	msrtg	misca
vpudelay	vpudelay	mgbtras	mgXrasbt	mssbs	miscc
vpureset	vpureset	mgbtrast	mgXrasbt	msvgrp	vram
ioXdmae_i	iodmae	mgbtrhld	mgXrasbt	oed	miscb
ioXdmaex	iodmaex	mgebtcase	mgXebt	oevras	vram
ioXdtt	iodtta/b	mgebtdobe	mgXebt	outdrv	drivers
ioXie_i	ioie	mgebtsum	mgXebt	pkgio	miscb
ioXin_i	ioin	mgXbw	miscb	pkgmflt	miscc
ioXip_i	ioip	mgXds	mgds	rasbcasbdrv	drivers
ioXius_i	ioius	mgXrd	misca	wevras	vram

Bus Operation

The MIF handles requests from all sources for access to the system bus. Requests arrive and are prioritized, respectively, from the VPU, DMAC and MPU. This order ensures that the VPU always has predictable memory timing, that DMA has bus availability (because the MPU can saturate the bus), and that memory coherency is maintained for the MPU.

Operation

To gain access to the bus, the bus address must be transferred to the MIF and a check made to see if the bus is available for the time required to complete the bus transaction. The available bus time is called the *slot* and the process checking is called the *slot check*. This bus request process takes two CPU-clock cycles at the beginning of each bus transaction. Memory-reference MPU and VPU instructions always overlap one cycle of instruction execution with the bus access process. DMA operation can overlap both cycles of the bus request process with a preceding MPU bus transaction. Thus, except for DMA overlapped with an MPU bus transaction, there are two CPU-clock cycles of no activity on the bus preceding each bus transaction. Instruction execution times listed include the bus request and programmed bus transaction time as part of the entire memory reference time.

The MIF must always grant the bus to the VPU immediately when requested in order to guarantee temporally deterministic VPU execution. To allow this, the VPU has exclusive access to the bus except when it is executing `delay`. When a DMA or MPU bus request is made, the MIF prioritizes the request, determines the type of bus transaction, computes the slot required (see Table 55), and compares this to `vpudelay`—the amount of time before the VPU seizes the bus. If `vpudelay` is zero, the VPU currently has the bus. If `vpudelay` is larger than the value computed for the bus transaction, the bus is granted to the requestor. Otherwise, the bus remains idle until a bus request occurs that can be satisfied, or until the VPU seizes the bus. Once a bus request has passed the slot check, the bus transaction begins on the next CPU-clock cycle.

Table 56. Bus Access Priorities

(Highest)
VPU
DMA:
- I/O Channel 0
- I/O Channel 1
- I/O Channel 2
- I/O Channel 3
- I/O Channel 4
- I/O Channel 5
- I/O Channel 6
- I/O Channel 7

MPU:
- Posted write
- Instruction pre-fetch
- Local-register stack spill or refill
- Operand stack spill or refill
- `ld/st`
- Instruction fetch

(Lowest)

Table 55. Slot Check Computation

For MPU bus transactions:
((number of RAS cycles) · `mgbtrast`) + ((number of bus cycles) · ((`mgbtcast` + 1) + `mgebtsum`))

For I/O-channel bus transactions†:
((number of RAS cycles) · `mgbtrast`) + ((number of bus cycles) · ((`mgbtcast` + 1) + `ioebtsum`))

Memory values are for the accessed memory group, and I/O-channel values are for the accessed I/O channel.

† To simplify calculation, this value is an estimate of the actual required slot. See text.

The slot check computation is an estimate because for I/O channel bus transactions `ioXebt` is used for all parts of the computation even though a mix of `ioXebt` and `mgXebt` times might be used during the transaction. The effect of this simplified computation is that the slot requested might be larger than the bus time actually used. The bus becomes immediately available for use when the actual bus transaction completes.

The address lines out of the CPU are multiplexed to reduce package pin count and provide an easy interface to DRAM. DRAMs have their addresses split into two pieces: the upper-address bits, or row address, and the lower-address bits, or column address. The two pieces of the address are clocked into the DRAM with two corresponding clock signals: $\overline{\texttt{RAS}}$ and $\overline{\texttt{CAS}}$. `AD[31:0]` also output higher-order address bits than the DRAM row and column addresses during RAS and CAS times, as well as data input or output during the last portion of each bus cycle while $\overline{\texttt{DOB}}$ is active. Bit outputs and bit inputs are also available on `AD[7:0]`.

I/O Addressing

All the address bits above the `msgsm` bits are referred to as the high address bits. These bits are typically used to address I/O devices with external decoding hardware. They can be configured to be included in RAS-cycle determination, or excluded for faster I/O cycles, to match the requirements of the external decoding hardware. See *System Requirements Programming*, page 126, for the available configuration options.

Bus Transaction Types

The CPU supports both cell-wide and byte-wide memory, cell-wide and byte-wide devices, and single- or multi-bus-cycle transactions. Various combinations of these are allowed; they require one, four, or five bus cycles to complete the bus transaction, which can include zero, one, or two RAS cycles. The underlying structure of all bus cycles is the same. Depending on the programmed system configuration, device-memory combination, and current system state, RAS prefix and CAS parts of bus cycles are combined to provide correct address generation and memory device operation. Table 58, page 163, lists the various combinations of RAS and CAS cycles that are possible within a given bus transaction.

MPU and VPU (non-`xfer`) Memory Cycles

The MPU and the VPU can read and execute programs stored in cell-wide or byte-wide memory. The

Table 57. I/O-Channel Transfer Characteristics

Device Width	Device Transfer Type[1]	Memory Width	Flyby[2]/ Buffered[3]	Bus Cycles[4]	Bits Moved
byte	0	byte	F	4	32
byte	0	cell	B	5	32
byte	1	byte	F	1	8
byte	1	cell	F	1	8
cell	2	byte	B	5	32
cell	2	cell	F	1	32

1. Refers to device type specified in `iodtta` or `iodttb`.
2. Data is transferred directly between device and memory.
3. Data is stored in the MIF during part of the transfer.
4. The entire sequence of cycles is an atomic bus transaction.

MPU can also read data from and write data to cell-wide and byte-wide memory. All accesses to cell-wide or byte-wide memory involve an entire cell. Accesses to cell-wide memory thus require one bus cycle, while accesses to byte-wide memory require four bus cycles.

Cell Memory Write from MPU
Cell Memory Read to MPU/VPU
Table 58 and the referenced figures provide details regarding these bus transactions. These transactions require one bus cycle.

Byte Memory Write from MPU
Byte Memory Read to MPU/VPU
Table 58 and the referenced figures provide details regarding these bus transactions. These transactions require four bus cycles. Byte address bits A1 and A0 are incremented from 0 to 3 to address the most-significant through the least-significant byte of the accessed cell.

I/O-Channel Transfers

Depending on the device transfer type and memory device width, a variety of bus cycle combinations occur between I/O devices and memory, as shown in Table 57. The starting address for the transaction comes from the global register that corresponds to the I/O channel involved (g8 corresponds to I/O channel 0, …, g15 corresponds to I/O channel 7). The direction of the transfer relative to memory is indicated by bit one of the same register. See Figure 16, page 104. The device transfer type for the transaction comes from the corresponding field in `iodtta` or `iodttb`. The bus transaction proceeds with the cycles and strobes listed in Table 58.

Cell Memory Write from Four-byte Byte-transfer Device
Table 58 and the referenced figure provide details regarding the bus transaction. The transaction requires five bus cycles. Data is collected from the device and stored in the MIF during the first four bus cycles, and is written to memory by the MIF during the fifth bus cycle. Data that is written to memory while being collected from the device during the first four bus cycles is replaced during the fifth bus cycle. A31 is cleared to deselect the I/O device in order to prevent contention with the MIF during the fifth bus cycle. Byte address bits A1 and A0 are incremented from 0 to 3 to address the most-significant through the least-significant byte of the accessed cell while the data is being transferred from the device.

Cell Memory Read to Four-byte Byte-transfer Device
Table 58 and the referenced figure provide details regarding the bus transaction. The transaction requires five bus cycles. Data is collected from memory and stored in the MIF during the first bus cycle and written to the device by the MIF during the last four bus cycles. $\overline{\texttt{OE}}$ is suppressed during the last four bus cycles to prevent bus contention between memory and the MIF while the device is written. A31 is cleared to deselect the I/O device in order to prevent contention with memory during the first bus cycle. Byte address bits A1 and A0 are incremented from 0 to 3 to address the most-significant through the least-significant byte of the accessed cell while the data is being transferred to the device.

Byte Memory Write from Four-byte Byte-transfer Device
Table 58 and the referenced figure provide details regarding the bus transaction. The transaction requires four bus cycles. Byte address bits A1 and A0 are incremented from 0 to 3 to address the most-significant through the least-significant byte of the accessed cell on both the device and memory. The data is transferred on the bus directly from the device to memory without the intervention of the MIF.

Byte Memory Read to Four-byte Byte-transfer Device
Table 58 and the referenced figure provide details regarding the bus transaction. The transaction requires four bus cycles. Byte address bits A1 and A0 are incremented from 0 to 3 to address the most-significant through the least-significant byte of the accessed cell on both the device and memory. The data is transferred on the bus directly from memory to the device without the intervention of the MIF.

Cell Memory Write from One-byte Byte-transfer Device
Table 58 and the referenced figure provide details regarding the bus transaction. The transaction requires

ADVANCE INFORMATION

one bus cycle. Data is typically supplied by the device on AD[7:0], and is written to the corresponding bits in memory. AD[31:8] are also written to memory, and, if not driven by an external device, still hold the CAS address bits.

Cell Memory Read to One-byte Byte-transfer Device
Table 58 and the referenced figure provide details regarding the bus transaction. The transaction requires one bus cycle. Data is typically taken by the device from AD[7:0], which come from the corresponding bits in memory. The other memory bits are driven by memory, but are typically unused by the device.

Byte Memory Write from One-byte Byte-transfer Device
Table 58 and the referenced figure provide details regarding the bus transaction. The transaction requires one bus cycle. Addresses in the global registers normally address cells because the lowest two bits are unavailable for addressing. However, for this transaction, the address in the global register is a modified byte address. That is, the address is shifted left two bits (pre-shifted in software) to be correctly positioned for the byte-wide memory connected to AD. The address is not shifted again before reaching AD. A31 remains in place, A30 and A29 become unavailable, and the group bits exist two bits to the right of their normal position due to the pre-shifting in the supplied address. This transaction allows bytes to be transferred, one byte per bus transaction, and packed into byte-wide memory.

Byte Memory Read to One-byte Byte-transfer Device
Table 58 and the referenced figure provide details regarding the bus transaction. The transaction requires one bus cycle. Addresses in the global registers normally address cells because the lowest two bits are unavailable for addressing. However, for this transaction, the address in the global register is a modified byte address. That is, the address is shifted left two bits (pre-shifted in software) to be correctly positioned for the byte-wide memory connected to AD. The address is not shifted again before reaching AD. A31 remains in place, A30 and A29 become unavailable, and the groups bits exist two bits to the right of their normal position in the due to the pre-shifting in the supplied address. This transaction allows bytes to be transferred, one byte per bus transaction, and unpacked from byte-wide memory to a device.

Cell Memory Write from One-cell Cell-transfer Device
Table 58 and the referenced figure provide details regarding the bus transaction. The transaction requires one bus cycle.

Cell Memory Read to One-cell Cell-transfer Device
Table 58 and the referenced figure provide details regarding the bus transaction. The transaction requires one bus cycle.

Byte Memory Write from One-cell Cell-transfer Device
Table 58 and the referenced figure provide details regarding the bus transaction. The transaction requires five bus cycles. Data is collected from the device and stored in the MIF during the first bus cycle and written to memory by the MIF during the last four bus cycles. Data that is written to memory while being collected from the device during the first bus cycle is replaced during the second cycle. A31 is cleared to deselect the I/O device in order to prevent contention with the MIF during the last four bus cycles. Byte address bits A1 and A0 are incremented from 0 to 3 to address the most-significant through the least-significant byte of the accessed cell while the data is being transferred from the MIF to memory.

Byte Memory Read to One-cell Cell-transfer Device
Table 58 and the referenced figure provide details regarding the bus transaction. The transaction requires five bus cycles. Data is collected from memory and stored in the MIF during the first four bus cycles and written to the device by the MIF during the last bus cycle. $\overline{OE}$ is suppressed during the fifth bus cycle to prevent a bus contention between the memory and MIF while the device is written. A31 is cleared to deselect the I/O device in order to prevent contention with memory during the first four bus cycles. Byte address bits A1 and A0 are incremented from 0 to 3 to address the most-significant through the least-significant byte of the accessed cell while the data is being transferred from the memory to the MIF.

ADVANCE INFORMATION

Bus Reset

External hardware reset initializes the entire CPU to the power-on configuration, except for `power_fail` in `mode`. While the reset is active (external or power-on self-reset), the AD go to a high-impedance state, OUT`[7:0]` go high, RASes go active, and all other outputs go inactive. See Figure 73, page 205, for waveforms.

Video RAM Support

VRAMs increase the speed of graphics operations primarily by greatly reducing the system memory bandwidth required to display pixels on the video display. A VRAM command is used to transfer an entire row of data from the DRAM array to an internal serial access memory to be clocked out to the video display. VRAMs also support other commands to enhance graphics operations. The VRAM operations are encoded by writing `vram` and performing an appropriate read or write to the desired VRAM memory address. Basic timing for VRAM bus cycles is the same as any similar bus transaction in that memory group. See Figure 32, page 137. Refresh and RAS cycles might also affect VRAM operations. See *Video RAM Support*, page 125. Waveforms representing the effects of the various `vram` options are on page 215.

Virtual-Memory Page Faults Input

The MIF detects memory page faults that are caused by MPU memory accesses by integrating fault detection with RAS cycles. The mapped page size is thus the size of the CAS page. The memory system RAS page address is mapped from a logical page address to a physical page address during RAS precharge through the use of an external SRAM. A memory fault signal supplied from the SRAM is sampled during `RAS` fall and, if low, indicates that a memory page fault has occurred. See Figure 52. The memory fault signal is input from `MFLT` or AD8. See *Alternate Memory Fault Input*, below.

When a memory fault is detected, the bus transaction completes without any of the signals that normally go active during the CAS part of the bus cycle. A memory fault exception is then signaled to the MPU, which executes a trap to service the fault condition. See Figure 77, page 212, for waveforms.

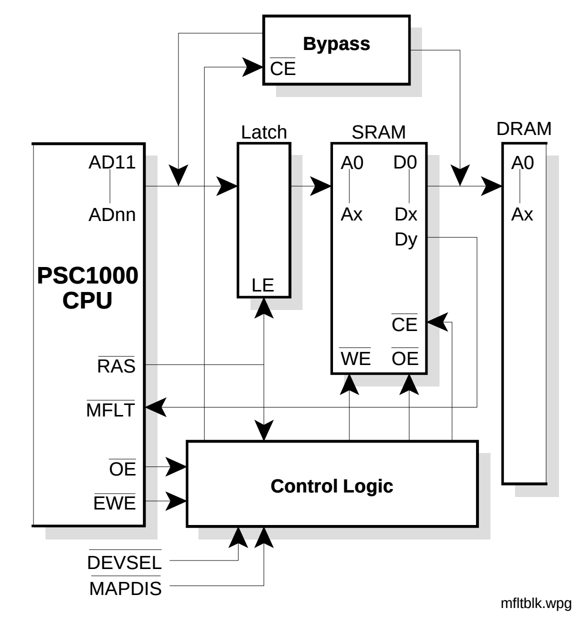


Figure 52. Virtual-Memory Page Mapping Logic

Alternate Inputs and Outputs

The bit inputs, bit outputs, memory fault input, and reset input can be multiplexed on AD rather than using the dedicated pins. This feature can be used to reduce the number of tracks routed on the PWB (to reduce PWB size and cost), and can allow the PSC1000 CPU to be supplied in smaller packages. See Figure 81, page 218, for waveforms.

Alternative Bit Inputs

The bit inputs can be sampled either from `IN[7:0]` or from AD`[7:0]` while `RAS` is low and `CAS` is high. The source is determined by `pkgio`. See Figure 34, page 140, and *Bit Inputs*, page 111.

Alternative Bit Outputs

The bit outputs appear both on OUT`[7:0]` and on AD`[7:0]` while `RAS` is high. Since they appear in both places, no selection bit is required. See *Bit Outputs*, page 115.

ADVANCE INFORMATION

Alternative Memory Fault Input

The memory fault signal can be sampled either from $\overline{\texttt{MFLT}}$ or from AD8 during $\overline{\texttt{RAS}}$ fall. The source is determined by `pkgmflt`. See Figure 39, page 145.

Alternative Reset Input

External hardware reset can be taken either from $\overline{\texttt{RESET}}$ or from AD8; the determination is made at power-on. The power-on and reset sequence is described in detail in *Processor Startup*, page 181.

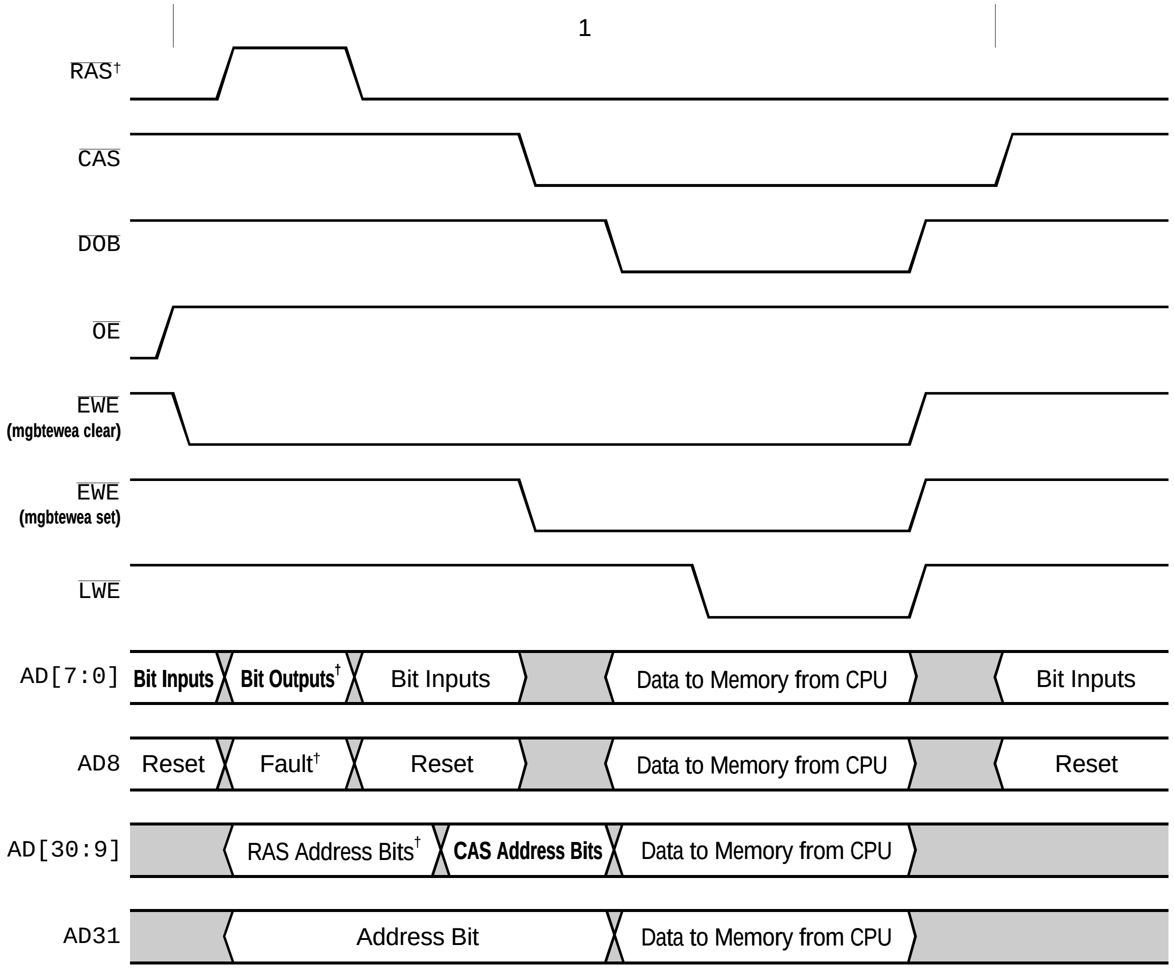
Table 58. RAS/CAS Bus Transactions

			Cell Memory										Byte Memory									
Device	**I/O-Channel Transfer Type[1]**		**Write to Memory**					**Read from Memory**					**Write to Memory**					**Read from Memory**				
		bus cycle #	**1**	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**
MPU/VPU[2]	-	cycle[3]	R_M C_M					R_M C_M					R_M C_M	C_M	C_M	C_M		R_M C_M	C_M	C_M	C_M	
		strobe[4]	w					o					w	w	w	w		o	o	o	o	
		See	Figure 53					Figure 54					Figure 55					Figure 56				
four-byte byte-transfer device	0	cycle[3]	R_I C_I	C_I	C_I	C_I	R_M C_M	R_M C_M	R_I C_I	C_I	C_I	C_I	R_I C_I	C_I	C_I	C_I		R_I C_I	C_I	C_I	C_I	
		strobe[4]	w	w	w	w	w[6]	o	–	–	–	–[7]	w	w	w	w		o	o	o	o	
		A31[5]	a	a	a	a	0	0	a	a	a	a	a	a	a	a		a	a	a	a	
		See	Figure 57					Figure 58					Figure 59					Figure 60				
one-byte byte-transfer device	1	cycle[3]	R_I C_I					R_I C_I					R_I C_I					R_I C_I				
		strobe[4]	w					o					w					o				
		A31[5]	a					a					a					a				
		See	Figure 61					Figure 62					Figure 63					Figure 64				
one-cell byte-transfer device	2	cycle[3]	R_I C_I					R_I C_I					R_I C_I	R_M C_M	C_M	C_M	C_M	R_M C_M	C_M	C_M	C_M	R_I C_I
		strobe[4]	w					o					w	w	w	w	w[8]	o	o	o	o	–[9]
		A31[5]	a					a					a	0	0	0	0	0	0	0	0	a
		See	Figure 65					Figure 66					Figure 67					Figure 68				

Notes:

1. I/O-channel transfer type in `iodtta` and `iodttb`.
2. VPU does not write to memory.
3. Indicates on which bus cycle RAS or CAS cycles are possible. Presence of a RAS cycle depends on system conditions. R_I or C_I indicates that the bus cycle uses `ioXebt` timing values, R_M or C_M indicate that the bus cycle uses `mgXebt` timing values.
4. Active strobe during cycle (w is $\overline{\texttt{EWE}}$/$\overline{\texttt{LWE}}$, o is $\overline{\texttt{OE}}$, – is no active strobe).
5. A31 selects the I/O device when set, deselects the I/O device when clear (a = program-supplied value, 0 = forced to zero).
6. Data is collected from the device and stored in the MIF during the first four cycles, and is written to memory by the MIF during the fifth cycle. Data written during first four cycles is replaced during the fifth cycle.
7. Data is collected from memory into the MIF during the first cycle and written to the device by the MIF during the last four cycles. $\overline{\texttt{OE}}$ is suppressed during the last four cycles to prevent memory from driving the bus.
8. Data is collected from the device and stored in the MIF during the first cycle, and is written to memory by the MIF during the last four cycles. Data written to memory during the first cycle is replaced during the second cycle.
9. Data is collected from memory into the MIF during the first four cycles, and is written to the device by the MIF during the last cycle. $\overline{\texttt{OE}}$ is suppressed on the fifth cycle to prevent memory from driving the bus.

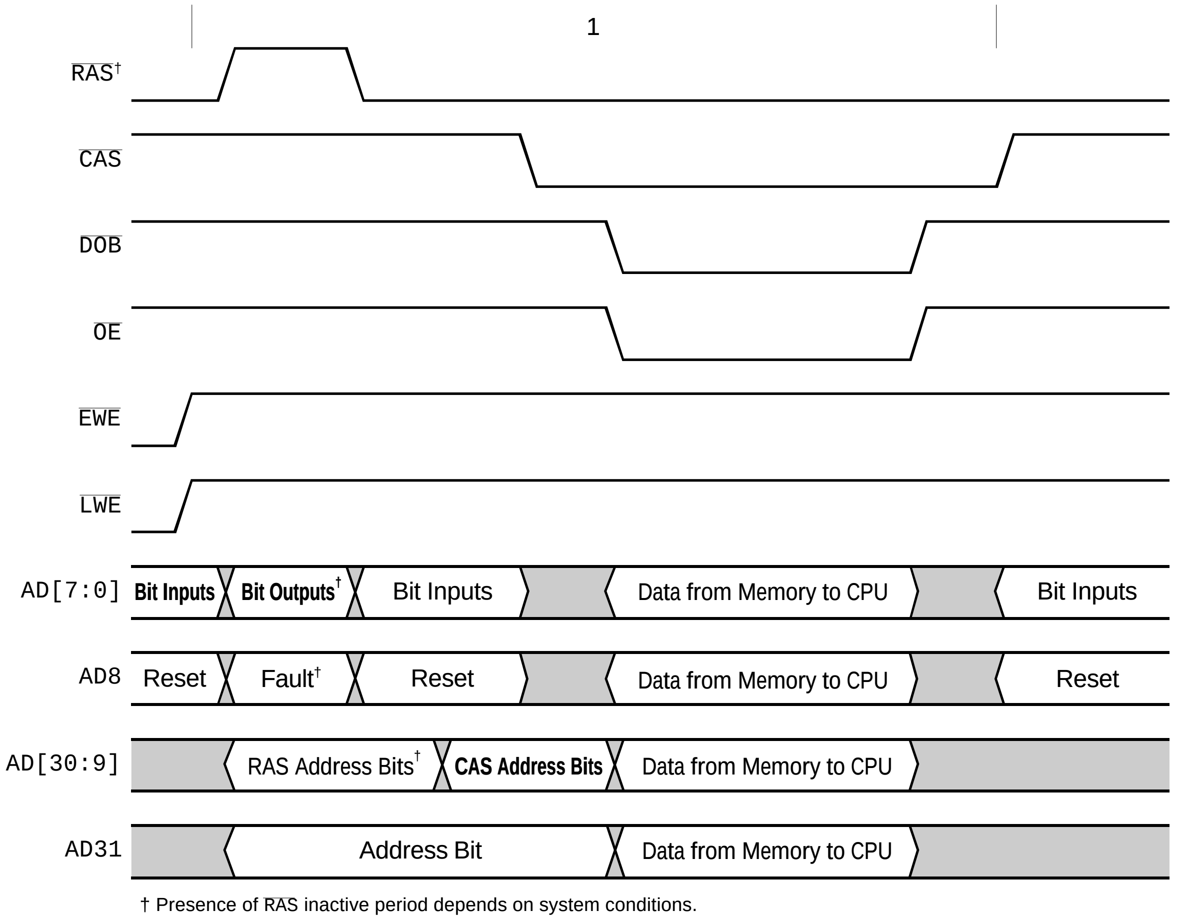
ADVANCE INFORMATION



† Presence of RAS inactive period depends on system conditions.

mpucellw.wpg

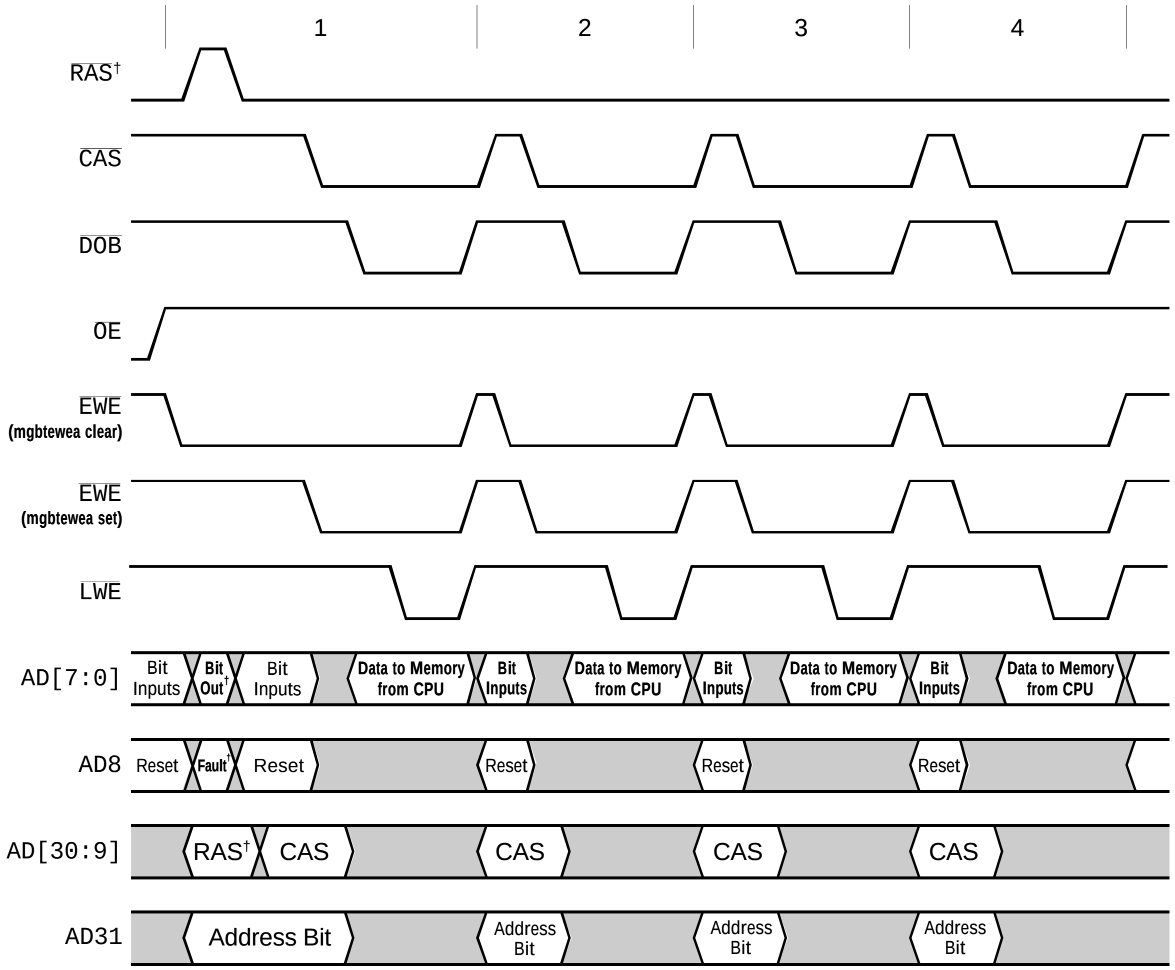
Figure 53. Cell Memory Write from MPU



† Presence of RAS inactive period depends on system conditions.

mpucellr.wpg

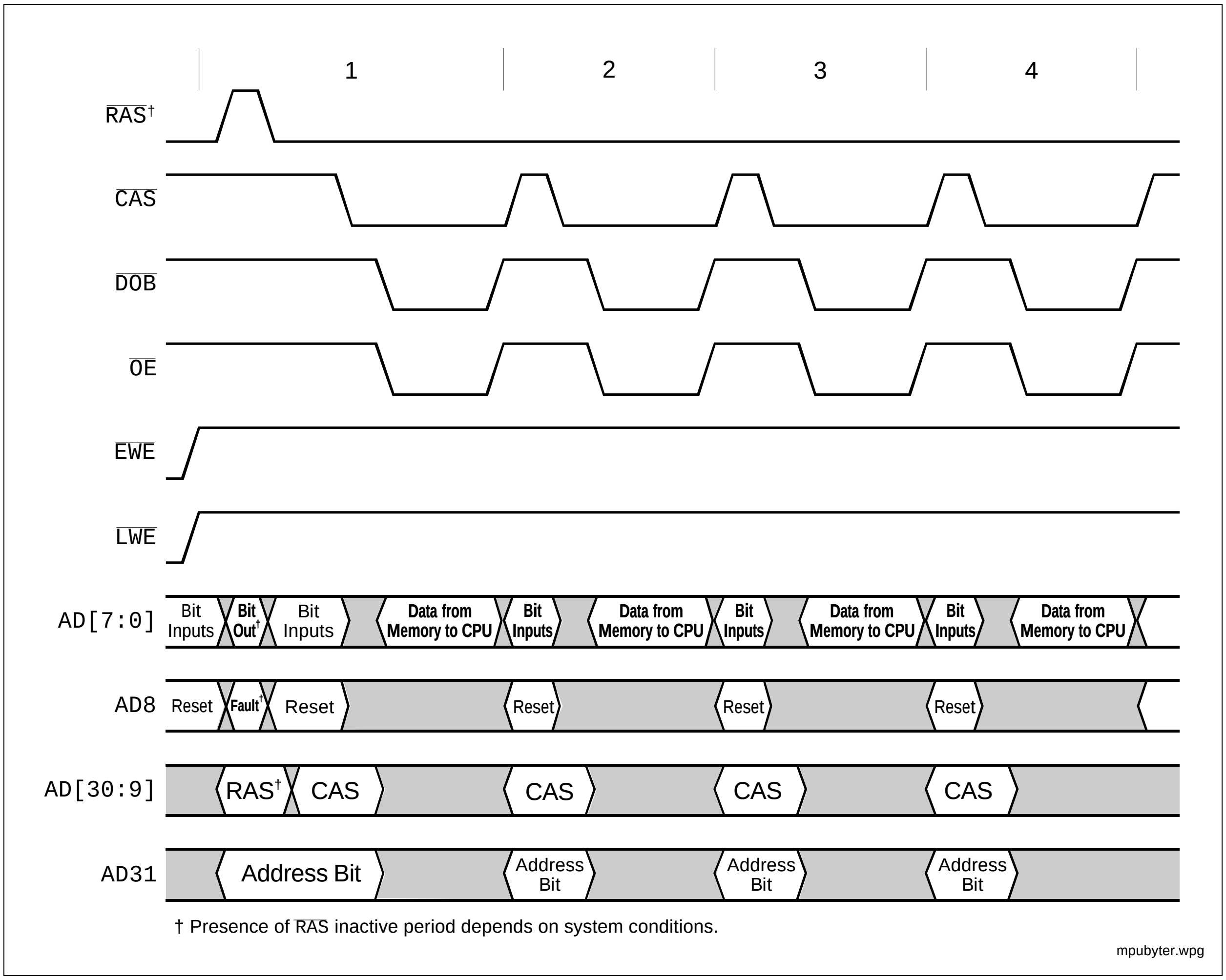
Figure 54. Cell Memory Read to MPU/VPU



† Presence of RAS inactive period depends on system conditions.

mpubytew.wpg

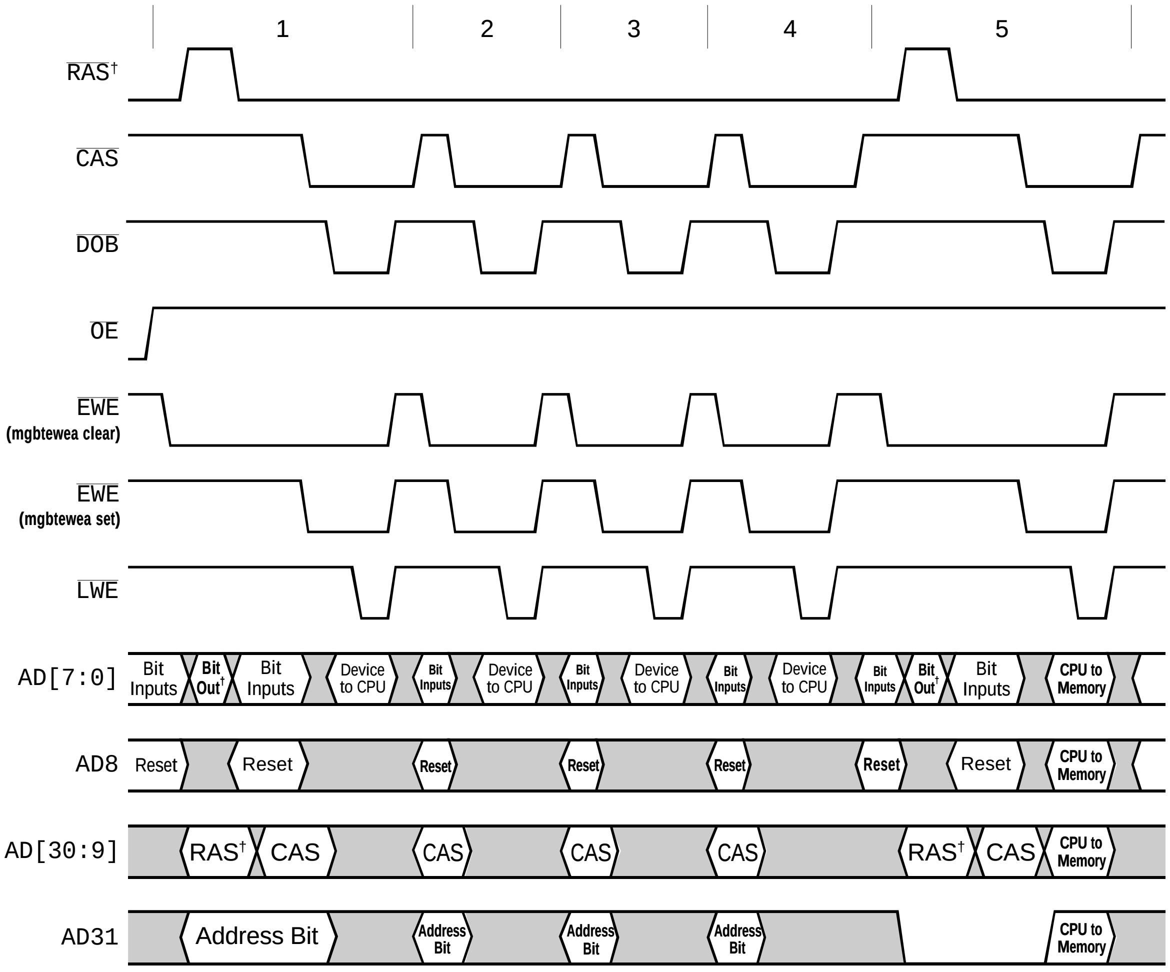
Figure 55. Byte Memory Write from MPU



† Presence of RAS inactive period depends on system conditions.

Figure 56. Byte Memory Read to MPU/VPU

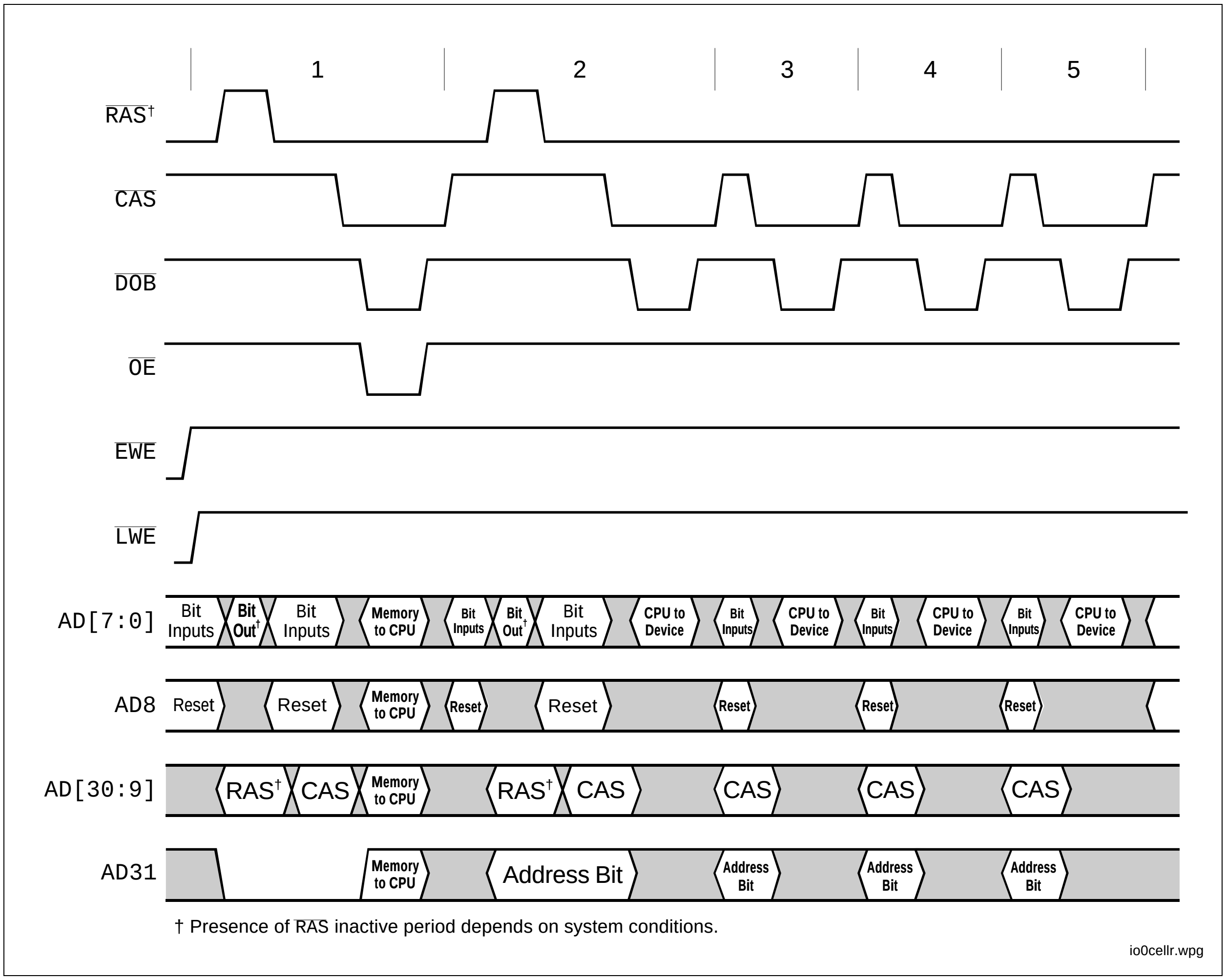
ADVANCE INFORMATION



† Presence of RAS inactive period depends on system conditions.

io0cellw.wpg

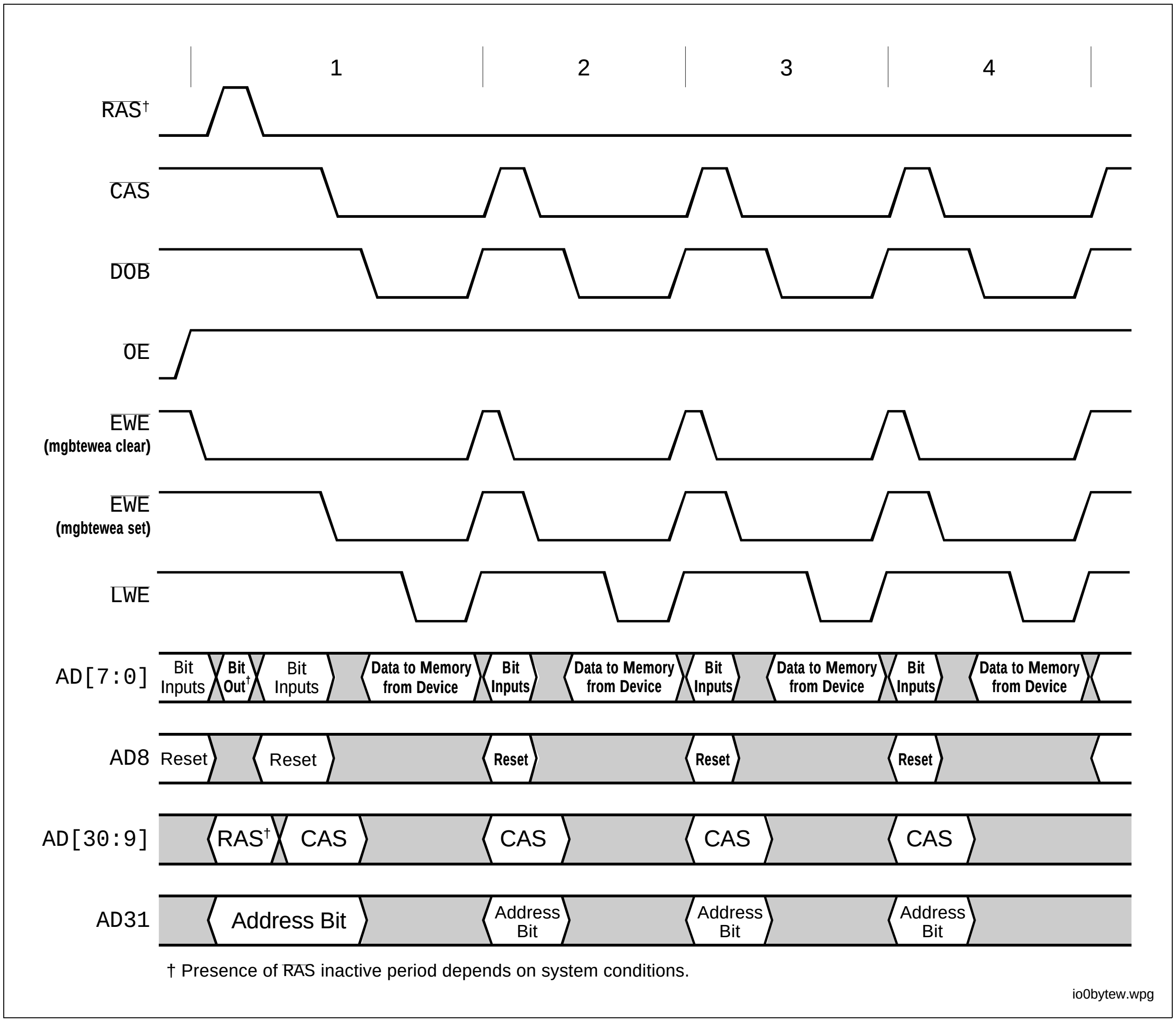
Figure 57. Cell Memory Write from Four-byte Byte-transfer Device



† Presence of RAS inactive period depends on system conditions.

Figure 58. Cell Memory Read to Four-byte Byte-transfer Device

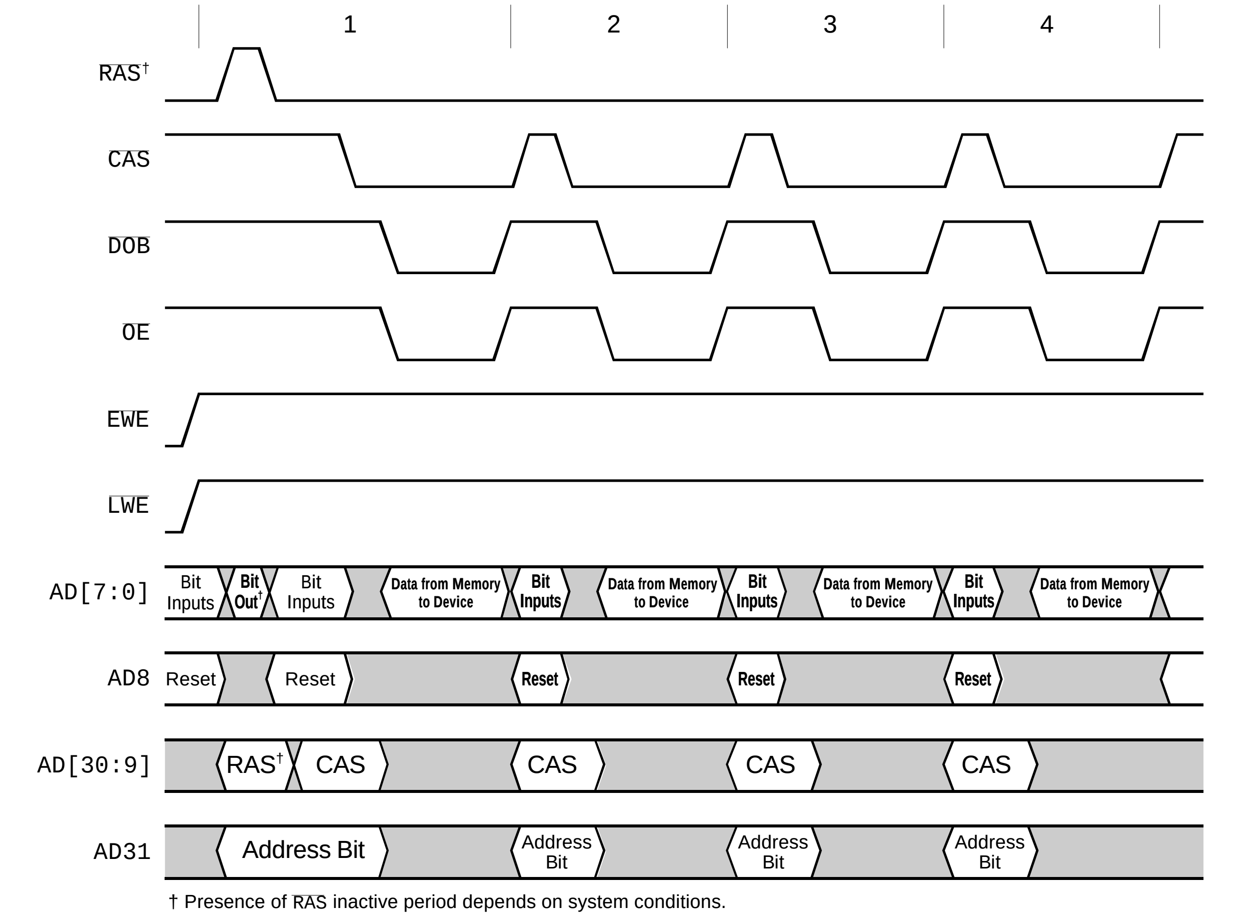
ADVANCE INFORMATION



† Presence of RAS inactive period depends on system conditions.

io0bytew.wpg

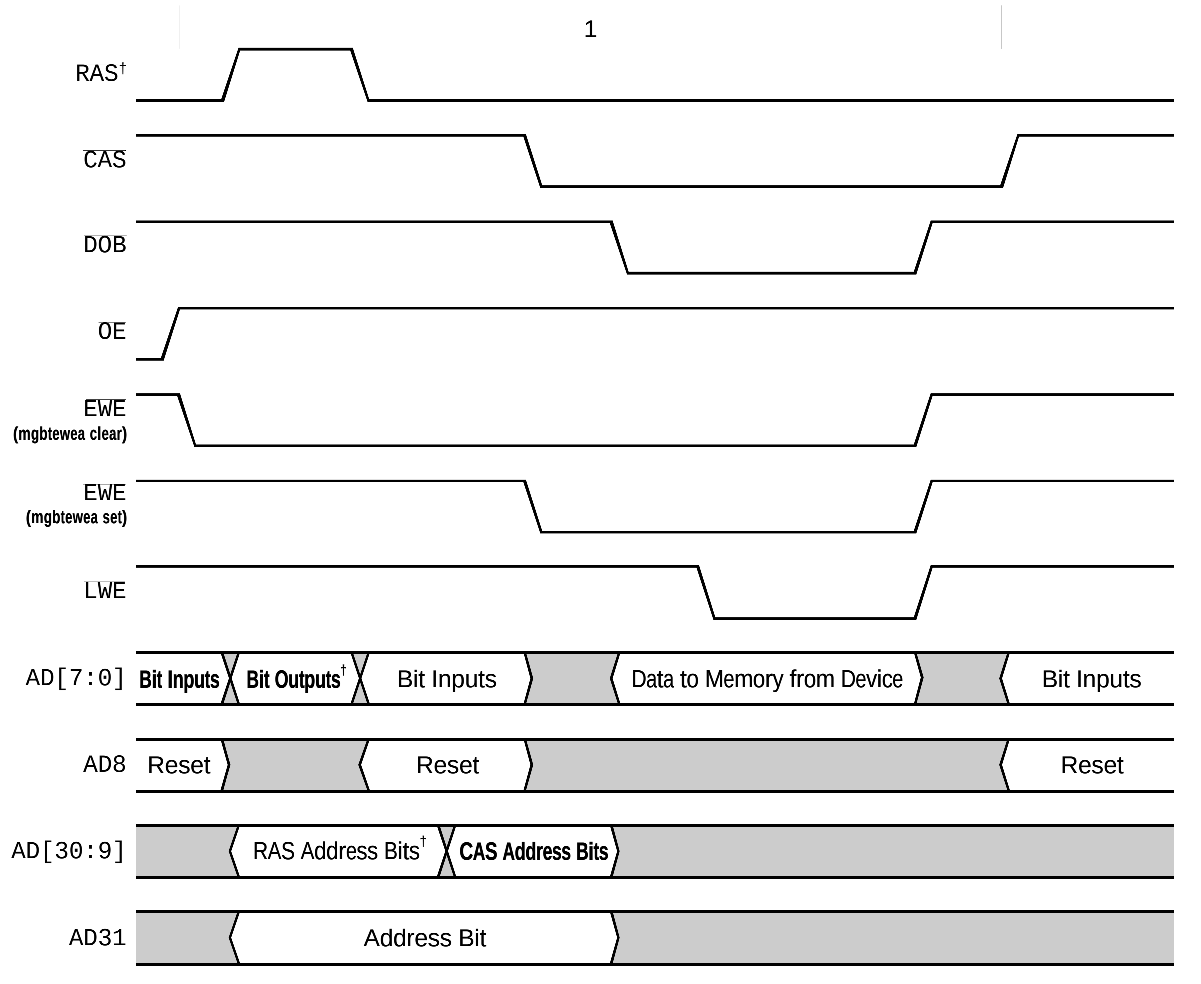
Figure 59. Byte Memory Write from Four-byte Byte-transfer Device



† Presence of RAS inactive period depends on system conditions.

io0byter.wpg

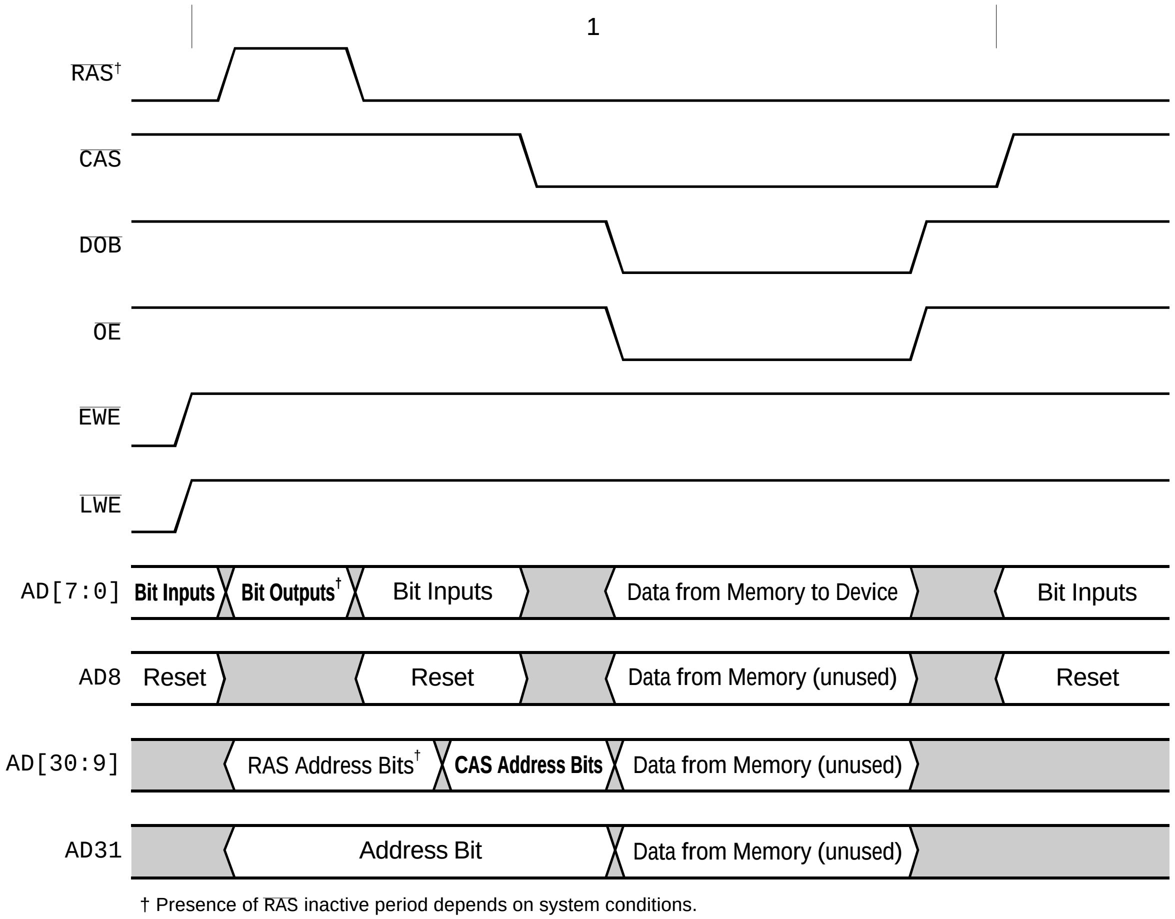
Figure 60. Byte Memory Read to Four-byte Byte-transfer Device



† Presence of RAS inactive period depends on system conditions.

io1cellw.wpg

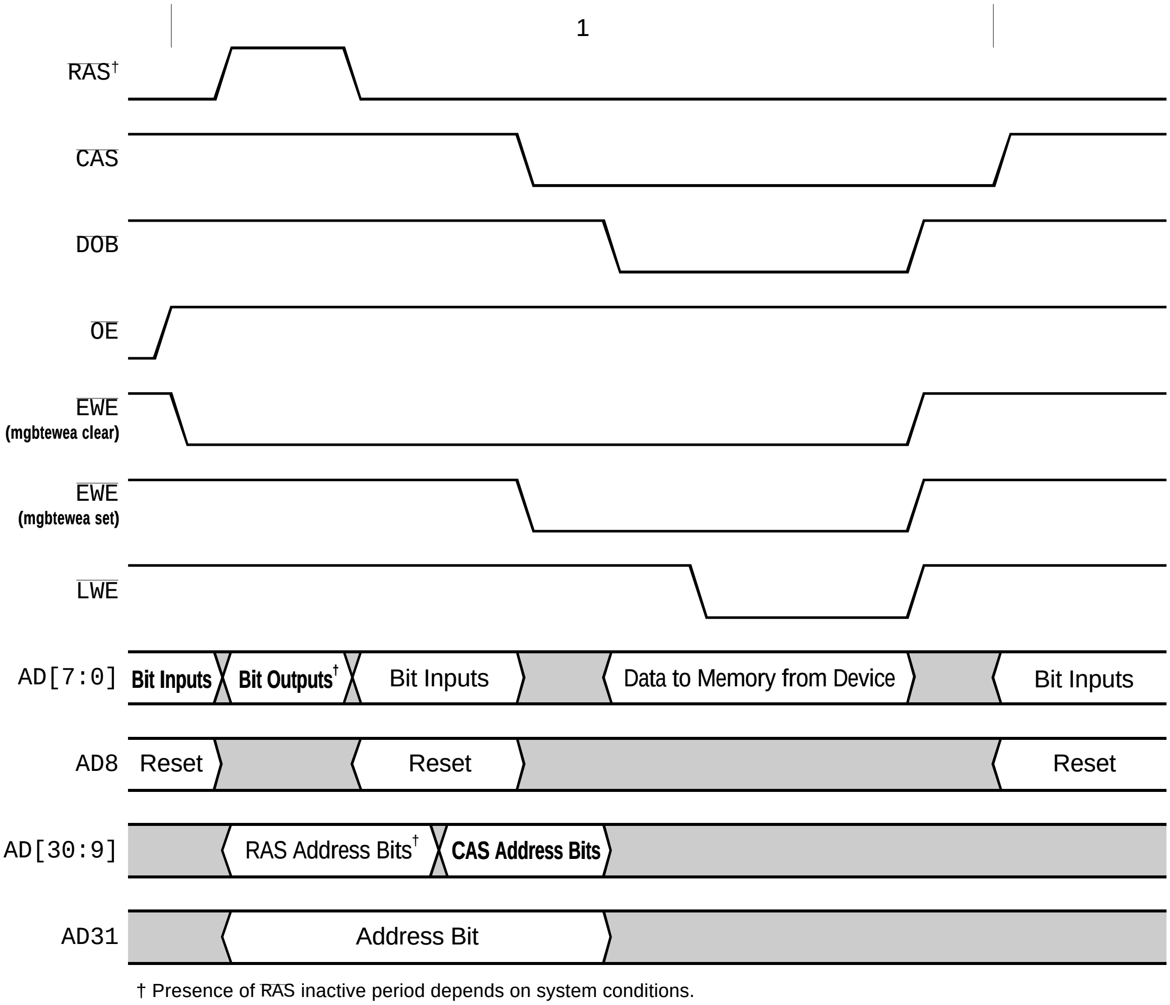
Figure 61. Cell Memory Write from One-byte Byte-transfer Device



† Presence of RAS inactive period depends on system conditions.

io1cellr.wpg

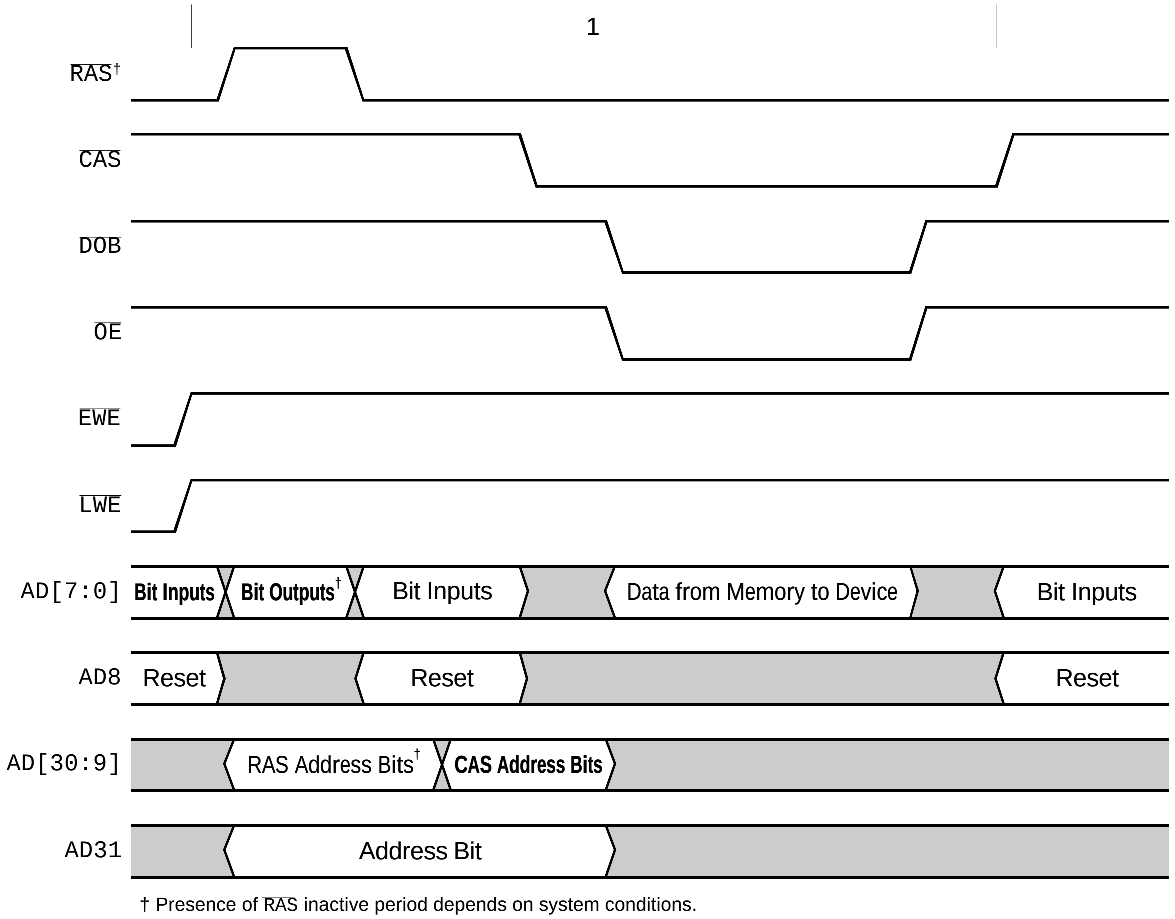
Figure 62. Cell Memory Read to One-byte Byte-transfer Device



† Presence of RAS inactive period depends on system conditions.

io1bytew.wpg

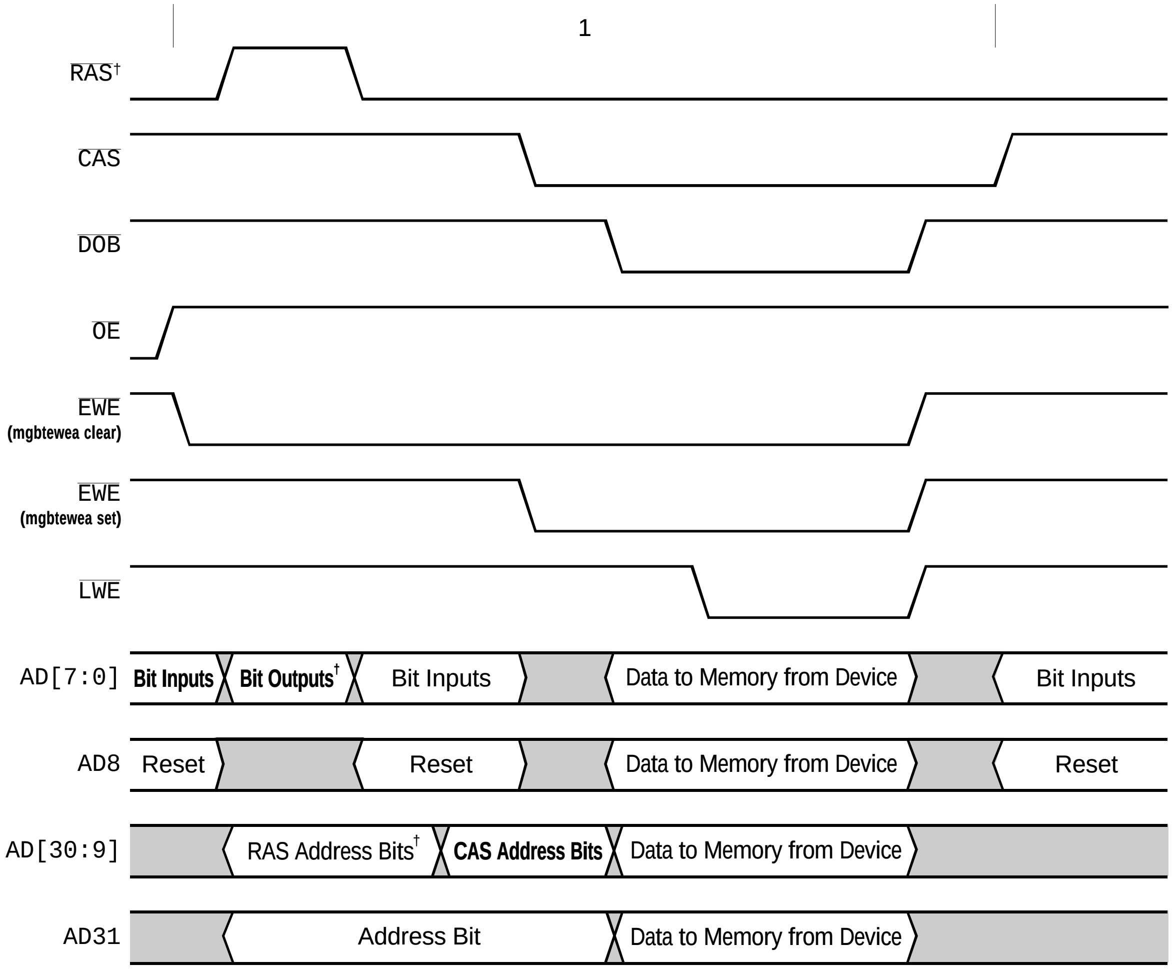
Figure 63. Byte Memory Write from One-byte Byte-transfer Device



† Presence of RAS inactive period depends on system conditions.

io1byter.wpg

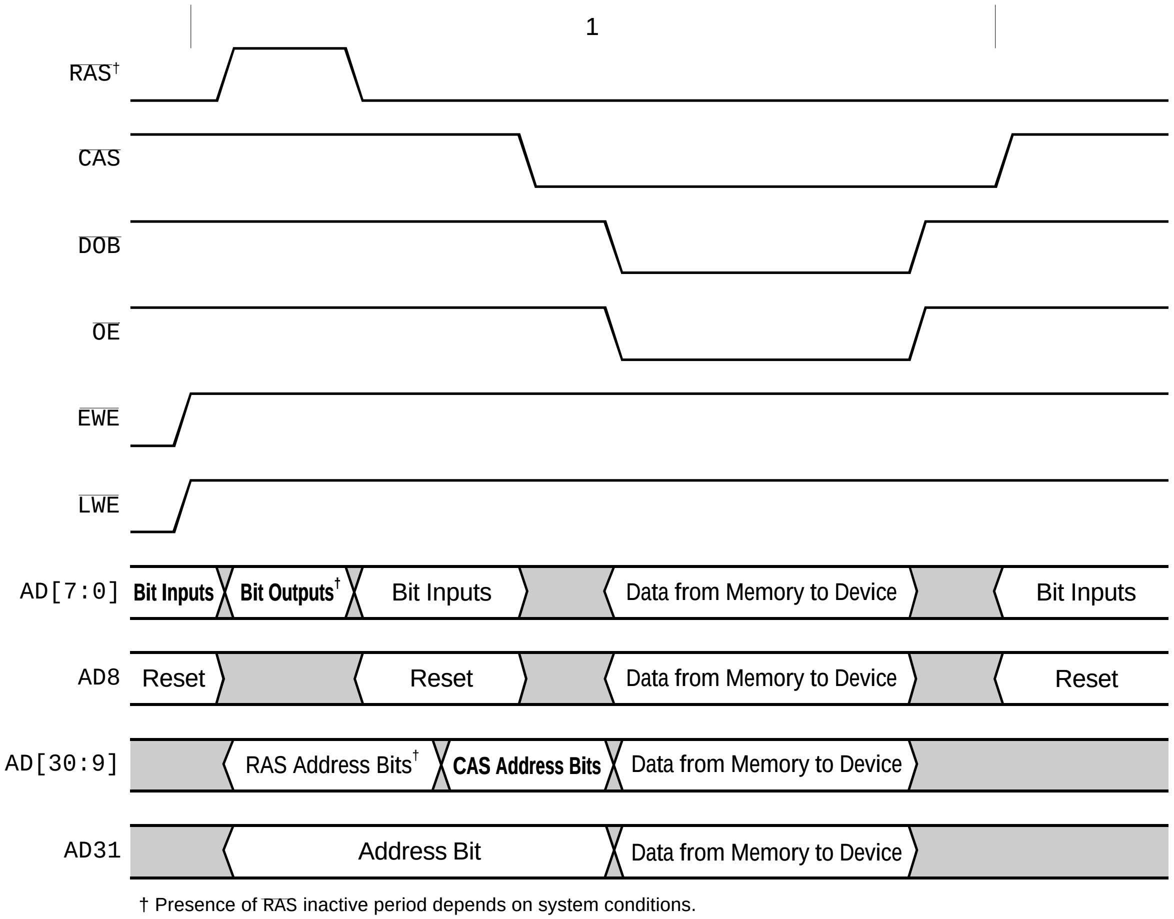
Figure 64. Byte Memory Read to One-byte Byte-transfer Device



† Presence of RAS inactive period depends on system conditions.

io2cellw.wpg

Figure 65. Cell Memory Write from One-cell Cell-transfer Device

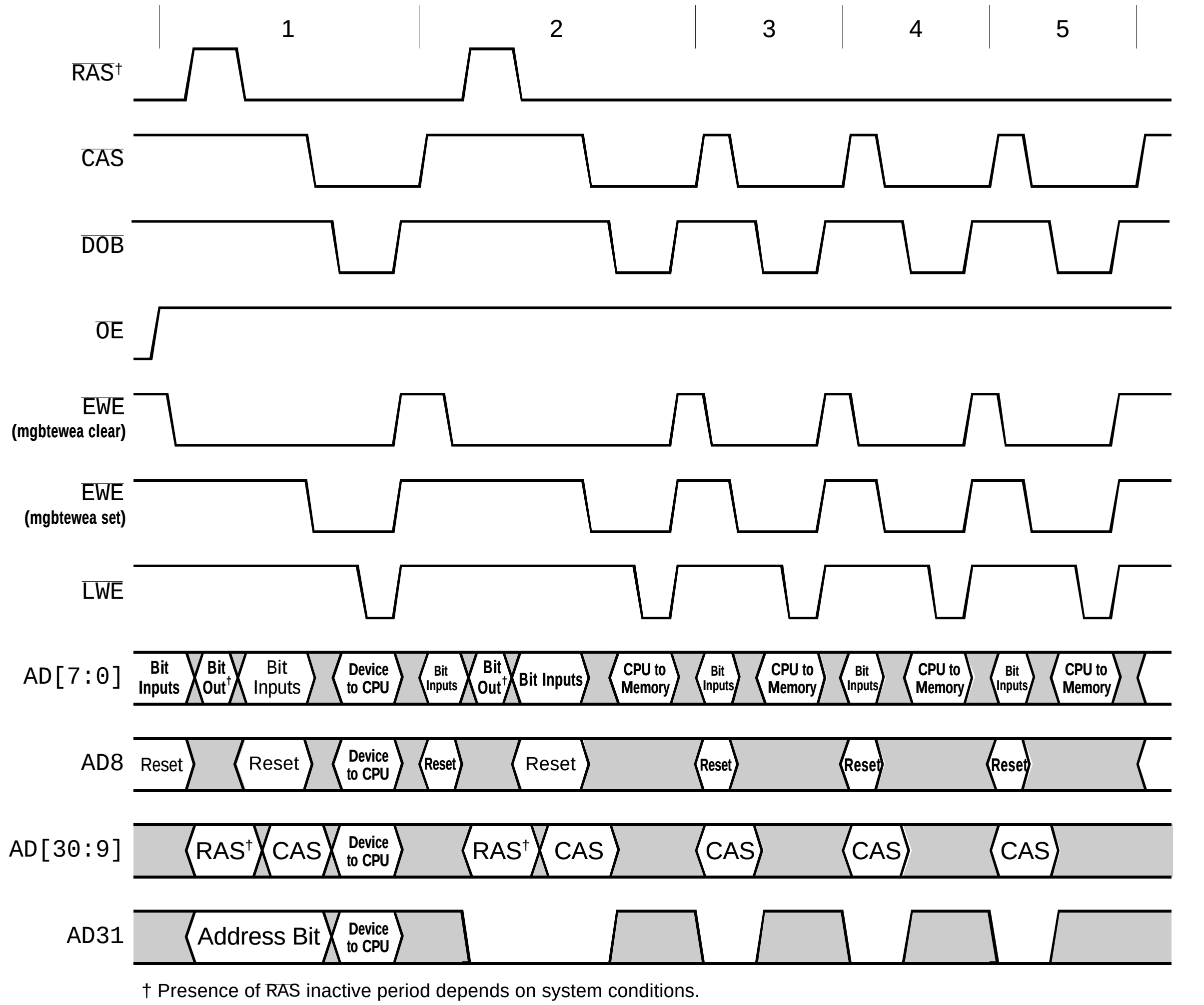


† Presence of RAS inactive period depends on system conditions.

io2cellr.wpg

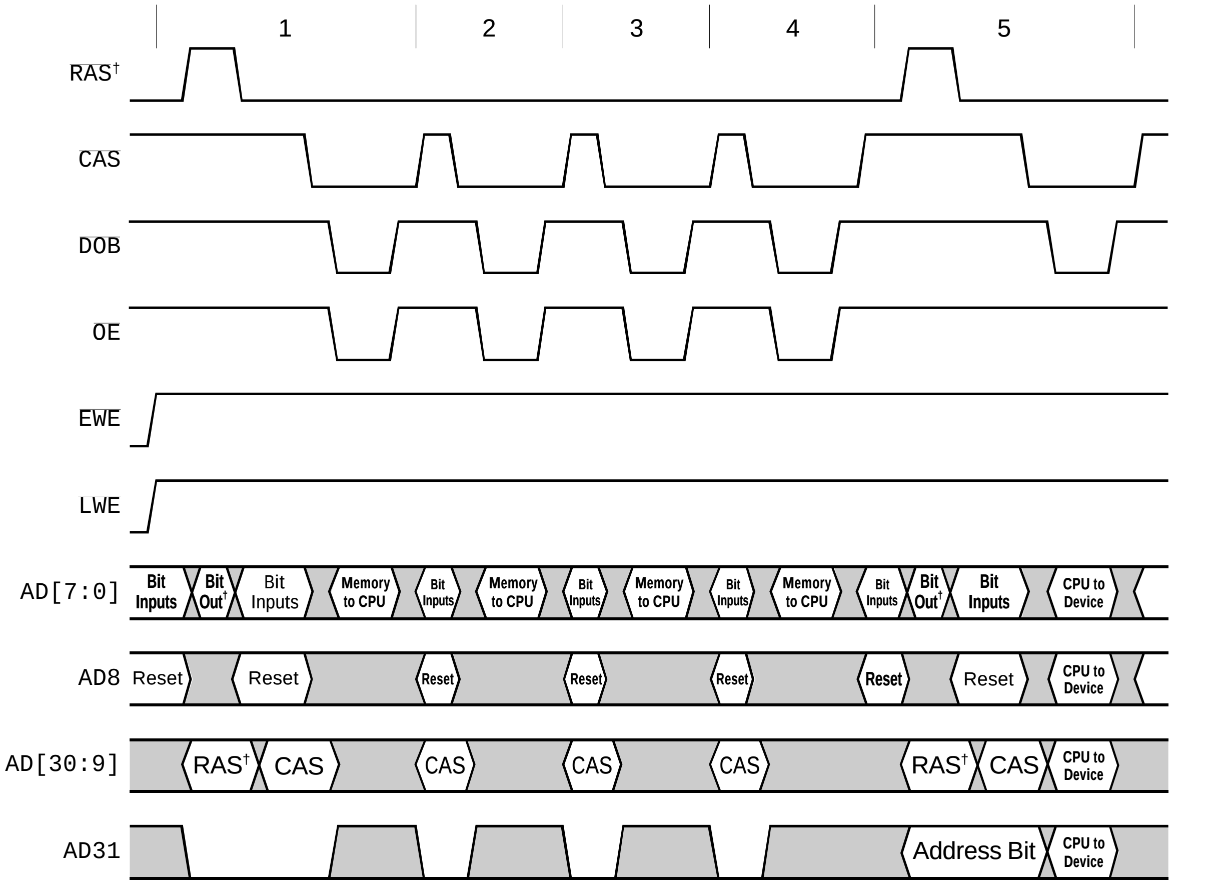
Figure 66. Cell Memory Read to One-cell Cell-transfer Device

ADVANCE INFORMATION



io2bytew.wpg

Figure 67. Byte Memory Write from One-cell Cell-transfer Device



† Presence of RAS inactive period depends on system conditions.

io2byter.wpg

Figure 68. Byte Memory Read to One-cell Cell-transfer Device

ADVANCE INFORMATION

Processor Startup

Power-on Reset

The CPU self-resets on power-up (see *Reset Process*, below). The CPU contains an internal circuit that holds internal reset active and keeps the processor from running, regardless of the state of the external hardware reset, until the supply voltage reaches approximately 3 V. Once the supply reaches 3 V, $\overline{\text{RESET}}$ is sampled and, if active, is used as the source of external reset for the CPU. Otherwise, external reset is multiplexed on AD8. This determination applies until power is cycled again. If one of the resets is active, the CPU waits until that reset goes inactive before continuing. If neither reset source is active, the processor immediately begins the reset sequence. The clock input at CLK, therefore, must be stable before that time.

During the power-on-reset process, the `mode` bit `power_fail` is set to indicate that the power had previously failed. The bit is cleared by any write to `mode`.

Boot Memory

The CPU supports booting from byte-wide memory that is configured as either an $\overline{\text{OE}}$-activated or boot-only memory device. The boot-only memory configuration is primarily used to keep the typically slow boot EPROMs out of the heavily used low-address memory pages.

Boot-only memory is distinct from $\overline{\text{OE}}$-activated memory in that it is wired into the system to place data on the bus without the use of $\overline{\text{OE}}$ or memory bank- or group-specific ($\overline{\text{RASx}}$ or $\overline{\text{CASx}}$) signals. OED is initially set during a CPU reset to disable $\overline{\text{OE}}$ during the boot-up process to allow the described operation. The boot-only memory select signal is externally decoded from the uppermost address bits that contain 0x800…. The number of uppermost address bits used depends on the system's I/O device address decoding requirements. The lowest address bits are connected so as to address individual bytes and cells as they are for a normal memory. Thus the boot-only memory device can be selected regardless of which memory group is accessed.

Reset Process

When reset occurs, the CPU leaves on-chip RAM uninitialized and clears most registers to zero, except for strategically placed bits that assist in the reset sequence. Specifically, the CPU resets to the most conservative system configuration. See Table 59. The `mode` bit `power_fail` is set only by the power-on-reset process and can be checked to determine whether the reset was caused by a power failure or reset going active.

The first bus transaction after reset is a cell read of four bytes from byte-wide memory in memory group zero, memory bank zero, starting from addresses 0x80000000, with $\overline{\text{OE}}$ disabled, in SMB mode. This address consists of I/O device address 0x800… and memory device address 0x…N. Because $\overline{\text{OE}}$ is disabled, $\overline{\text{OE}}$-activated memory does not respond, thus allowing a boot-only memory to respond.

The CPU tests the byte returned from address 0x80000003. If the byte is 0xa5 then a boot-only memory responded and execution continues with $\overline{\text{OE}}$ disabled. Otherwise, a boot-only memory did not respond, and the CPU assumes booting occurs from $\overline{\text{OE}}$-activated memory. The CPU then clears OED to activate $\overline{\text{OE}}$ for this memory to respond on subsequent bus cycles.

Bootstrap Programs

With either boot-only or $\overline{\text{OE}}$-activated memory, bus accesses continue in SMB mode from the byte-wide memory device. The second bus transaction is to the hardware reset address for the VPU at 0x80000004. This typically contains a `jump` to a small `refresh/delay` loop. The `delay` makes the bus available and allows the MPU to begin executing at its reset address, 0x80000008. The programmer must ensure that the delay value programmed in the VPU is sufficient to allow the MPU on the bus with the very slow byte-wide bus transactions that default after reset.

If the system is wired in MMB mode, booting is simpler from a boot-only memory. Booting from $\overline{\text{OE}}$-activated memory is also possible, but requires external gating to prevent bank zero of memory

ADVANCE INFORMATION

groups one, two, and three from being selected when memory group zero is accessed.

Next, the MPU begins executing and typically is programmed to branch to the system bootstrap routine. The MPU bootstrap is programmed to:

- set the configuration registers required for the system hardware,
- set the software reset vector for the VPU,
- copy the initial MPU and VPU application programs from the boot device into memory (if required),
- branch to the application program for the MPU, and
- reset the VPU in software to begin VPU program execution (if required).

System startup is now complete.

The following pages describe several startup configurations. For actual code see *Example PSC1000 CPU System*, page 187. The configurations described below are:

- Boot from byte-wide boot-only memory and copy the application program to cell-wide R/W memory.
- Boot from cell-wide boot-only memory and copy the application program to cell-wide R/W memory.
- Boot and run from byte-wide memory.
- Boot and run from cell-wide memory.

Boot from Byte-Wide Boot-Only Memory and Copy the Application Program to Cell-Wide R/W Memory
This process requires external decoding hardware to cause the boot-only memory to activate as previously described.

To indicate that boot-only memory is present, the memory must have 0xa5 at location 0x80000003 (typically 0x000000a5 in the cell at 0x80000000). This signature byte must be detected at startup to continue the boot process from a boot-only memory.

Construct the boot program execution sequence to be as follows:

1. The VPU executes `JUMP` from its power-on-reset location to code that performs eight RAS cycles on each memory group (by performing refresh cycles) to initialize system DRAM. It then enters a micro-loop that includes `refresh` for DRAM, and `delay` to allow the MPU to execute. The micro-loop repeats `refresh` and `delay`, and eliminates VPU accesses to the bus for further instructions during configuration. `delay` allows the MPU bus access to begin configuring the system before more refresh cycles are required. The refresh cycles are not required if the system does not contain DRAM.

2. The MPU executes `br` from its reset location to the program code to configure the system. The `br` should contain bits that address memory group three. This later allows the configuration for memory group three to be used for boot-only device access timing while memory groups zero, one and two are programmed for the system timing requirements. Although memory group one or two could be used instead of three in the manner described herein, only memory group three is discussed for simplicity.

The MPU configuration program code must be arranged to hold off instruction pre-fetch so that the configurations of the current memory group and the global memory system are not changed during a bus cycle. See the supplied example boot code on page 191.

3. When programming `miscb`, set `mmb` if required. In systems wired for MMB mode this allows RAS-type cycles to occur properly on all memory groups.

4. Set `msgsm` to define four memory groups, even if the system ultimately does not have them. During the next instruction fetch the boot-only memory is again selected. However, the address bits for memory group three placed in the PC by `br` in step two cause the configuration for memory group three to be used.

5. Program the timing of memory group three to optimize access to the boot-only memory. Then program the remainder of the system configuration. During this process the VPU typically performs three or so sets of refresh cycles. Though it is possible that the MPU will be changing pertinent configuration registers during the refresh cycles, it is very unlikely

due to the long bus cycle times of EPROMs typically used for boot-only memory. Further, the worst result is inappropriate timing on a single refresh cycle, which is of little actual consequence since there is no data yet in DRAM to be protected.

If memory group three is used by the application, it must be configured later from the loaded application code.

6. Read the final boot code (if any) and the application program from the boot-only memory and write them to the appropriate locations in R/W memory. The entire application program can be loaded into R/W RAM, except for that part, if any, that is destined for memory group three, where the boot-only memory is running. This must be copied by the application once it is running.

7. Layout a single instruction group that contains programming to clear `OED` and to branch to the application program. Using `br[]` clears A31 so that the boot-only memory does not activate at the branch destination.

8. Now the application program is executing. Configure memory group three, if required. If loading memory group three from the boot-only memory is necessary, then arrange the code between two instruction groups to first ensure pre-fetch is complete, then set `OED`, then execute a micro-loop to transfer the application to memory group three, and reenable `OED` when the micro-loop completes.

9. Reset the VPU in software to begin execution of its application program. A software reset of the VPU causes it to begin executing at 0x10, and as a result clears A31 from the VPU PC so the boot-only memory is no longer selected.

The boot process is complete.

Boot from Cell-Wide Boot-Only Memory and Copy the Application Program to Cell-Wide R/W Memory
This process requires external decoding hardware to cause the boot-only memory to activate as previously described.

The CPU always initially boots from byte-wide memory since this is the reset configuration. The CPU executes instructions from the low byte of each address until the configuration for the current memory group is programmed to be cell wide. Up to this point, the upper 24 bits of the boot-device data are unused. The boot process is otherwise the same as booting from byte-wide boot-only memory, except that at step 3, when writing `miscb`, also set memory groups zero and three to be cell-wide. In the instruction group with the `sto` to `miscb` place a `br` to the next instruction group. This holds off pre-fetch so that the next instruction fetch is cell-wide. Note that the boot program must be carefully programmed so that the instructions before the `br` are represented as byte-wide and after the `br` are represented as cell-wide. The Patriot linker has a section directive, `CELLBOOT`, to create the appropriate initial section.

Boot and Run from Byte-Wide Memory
This process requires the boot/run memory device to be activated by $\overline{MGS0}$/$\overline{RAS0}$/$\overline{CAS0}$. A31 is not used when selecting the boot/run memory.

To indicate that $\overline{OE}$-activated memory is present, the memory must not respond with 0xa5 at location 0x80000003 when $\overline{OE}$ is not asserted. The lack of this signature byte is detected at startup to indicate that $\overline{OE}$ is required to continue the boot process. `OED` is set during a CPU reset to disable `OED` during the boot-up process, and cleared when the signature byte 0xa5 is not detected, re-enabling $\overline{OE}$.

Construct the boot program execution sequence to be as follows:

1. The VPU executes `JUMP` from its power-on-reset location to code that performs eight RAS cycles on each memory group (by performing refresh cycles) to initialize system DRAM. It the enters a micro-loops that includes `refresh` for DRAM, and `delay` to allow the MPU to execute. The micro-loop repeats `refresh` and `delay`, and eliminates accesses by the VPU to the bus for further instructions during configuration. `delay` allows the MPU bus access to begin configuring the system before more refresh cycles are required. The refresh cycles are not required if the system does not contain DRAM.

ADVANCE INFORMATION

2. The MPU executes `br` from its reset location to the program code to configure the system.

The MPU configuration program code must be arranged to hold off instruction pre-fetch so that the configurations of the current memory group and the global memory system are not changed during a bus cycle. See the supplied example boot code on page 191.

3. When programming `miscb`, set `mmb` if required. In systems wired for MMB mode this allows RAS-type cycles to occur properly on all memory groups.

4. Program the timing of memory group zero to optimize access to the memory. Then program the remainder of the system configuration. During this process the VPU typically performs three or so sets of refresh cycles. Though it is possible for the MPU to be changing pertinent configuration registers during a refresh cycle, it is very unlikely due to the long bus cycle times of EPROMs. Further, the worst result is inappropriate timing on a single refresh cycle, which is of little actual consequence since there is no data yet in DRAM to be protected.

5. Reset the VPU in software to begin execution of its application program, if needed. A software reset of the VPU causes it to begin executing at 0x10, and as a result clears A31 from the VPU PC.

6. Begin execution of the application program.

The boot process is complete.

Boot and Run from Cell-Wide Memory

This process requires the boot/run memory device to be activated by $\overline{\texttt{MGS0}}$/$\overline{\texttt{RAS0}}$/$\overline{\texttt{CAS0}}$. A31 is not used when selecting the boot/run memory.

The CPU always initially boots from byte-wide memory since this is the reset configuration. The CPU executes instructions from the low byte of each address until the configuration for the current memory group is programmed to be cell wide. Up to this point, the upper 24 bits of the boot-device data are unused. The boot process is otherwise the same as booting and running from byte-wide memory, except that at step 3, when writing `miscb`, also set memory group zero to be cell- wide. In the instruction group with the `sto` to `miscb` place a `br` to the next instruction group. This holds off pre-fetch so that the next instruction fetch is cell-wide. Note that the boot program must be carefully programmed so that the instructions before the `br` are represented as byte-wide and after the `br` are represented as cell-wide. The Patriot linker has a section directive, `CELLBOOT`, to create the appropriate initial section.

Stack Initialization

After CPU reset both of the MPU stacks are uninitialized until the corresponding stack pointers are loaded. This should be one of the first operations performed by the MPU.

After a reset, the operand stack is abnormally empty. That is, `s2` has not been allocated, and is allocated on the first push operation. However, popping this item causes the stack to be empty and require a refill. The first pushed item should therefore be left on the stack, or `sa` should be initialized, before the operand stack is used further.

Table 59. System Configuration after CPU Reset

Uninitialized

s2–s17	sdepth	r1–r15	ldepth	g1–g15

Initialized Zero

s0–s1	r0	g0	x	ct	ioip	ioius	ioie
iodmae	misca	mfltaddr	mfltdata	msgsm	miscc	msra	vpudelay
iodtta	iodttb	iodmaex	drivers				

Initialized Non-zero

sa	0xfffffffc	la	0xfffffffc	ioin	0xff	ioout	0xff	mode	0x00004000

vram							
msvgrp	0x03	dsfvcas	zero	dsfvras	zero	casbvras	zero
wevras	zero	oevras	zero				
miscb							
mmb	zero	fdmap	zero	pkgio	zero	oed	one
mg3bw	one	mg2bw	one	mg1bw	one	mg0bw	one
mgds							
mg3ds	0x0f	mg2ds	0x0f	mg1ds	0x0f	mg0ds	0x0f
mgXebt							
mgebtsum*	0x1f	mgebtdobe	0x0f	mgebtcase	0x03		
mgXcasbt							
mgbtcas	0x07	mgbtdob	0x0f	mgbtcast	0x1f	mgbtewea	one
mgbtlwea	zero	mgbteoe	zero	mgbtewe	zero		
mgXrasbt							
mgbtrast	0x1f	mgbtras	0x0f	mgbtrhld	0x0e	mgbteras	zero
ioXebt							
ioebtsum*	0x1f	ioebtdobe	0x0f	ioebtcase	0x03		

The CPU reset conditions produce the following configuration:

Stacks uninitialized.
All interrupts, traps, faults, DMAs, and DMA expirations disabled.
VRAM memory group set to memory group three, no VRAM options set.
VRAM memory group included in high address bit compare caused RAS cycles.
Refresh enabled on all groups using memory group zero timing, refresh address bits starting at zero.
Posted writes disabled.
Cause RAS cycle when A31 = 1.
Cause RAS cycle when high address bits change.
A31 included in high-address-bit compare.
Single memory bank per memory group (SMB) mode with one memory group.
Input bits taken from the bus.
Memory fault taken from AD8.
$\overline{\text{OE}}$ disabled.
All memory groups are byte-wide.
All memory device sizes set to SRAM.
Memory bus cycles set for maximum length, actual CAS cycle length set to 51 CPU-clock cycles with CAS precharge of eight 2X-CPU-clock cycles (* xxebtsum registers set to maximum, which requests a slot larger than actually required), CAS address hold time of eight 2X-CPU-clock cycles, $\overline{\text{EWE}}$ fall at $\overline{\text{CAS}}$ fall, memory write data setup time to $\overline{\text{LWE}}$ fall of 15 CPU-clock cycles, memory write data setup time to $\overline{\text{EWE}}$ and $\overline{\text{LWE}}$ rise of 39 CPU-clock cycles, $\overline{\text{OE}}$ active time of 39 CPU-clock cycles, data hold time/buffer disable time of four CPU-clock cycles, RAS precharge of 16 CPU-clock cycles, RAS address hold time of 14 CPU-clock cycles.
All I/O-channel timings set for maximum length (the same bus cycles as memory above).
Revolving DMA priorities.
Device transfer types all set to four-byte byte-transfer devices.

ADVANCE INFORMATION

Example PSC1000 CPU Systems

Example System 1

Figure 69 depicts a minimal system with an 8-bit wide EPROM in memory group zero, and 256K of 8-bit-wide DRAM in memory group one. Memory group zero and memory group one must be configured with timing appropriate for the devices used, and `mg1ds` set to 0x02 (256K DRAM). Otherwise, the default system configuration is suitable. The system can boot and run directly from the EPROM, or, since EPROMs are generally slower than DRAM, can copy the EPROM into DRAM for faster code execution.

Example System 2

Figure 70 depicts a minimal system with 32-bit-wide DRAM in memory group zero, an 8-bit-wide EPROM as a boot-only memory device, and an I/O address decoder. The I/O address decoding is performed by a 74HC137, a 3-to-8 decoder with latch. The decoder is wired to supply four device selects when A31 is set, and another four when A31 is clear. The sets of four selects are latched during RAS precharge and enabled during $\overline{\text{CAS}}$ active. They are decoded from A30 and A29 when a 32-bit-wide memory group is involved and from A28 and A27 when an 8-bit-wide memory group is involved. The device select with A31 set and the other decoded address bits clear is used to select the EPROM as a boot-only memory device.

The EPROM must be programmed with 0xa5 at location 0x80000003 (typically 0x000000a5 at location 0x80000000). Memory group zero must be configured with timing appropriate for the devices used, `mg0bw` set to zero (cell wide), and `mg0ds` set to 0x02 (256K DRAM). Since RAS is used to latch the I/O address, `msras31d`, `mshacd` and `msexa31hac` must remain in their default configuration of clear.

Example System 3

Figure 71 depicts a system with 32 KB of 32-bit-wide SRAM in memory group zero, 1 MB of 32-bit-wide DRAM in memory group one, an 8-bit-wide EPROM as a boot-only memory device, and an I/O address decoder. Address latching of the CAS address for the SRAM is performed by two 74ACT841 transparent latches. The address inputs of the DRAM and EPROM are also connected to the outputs of the latches, though they could have been connected to the corresponding AD instead. The I/O address decoding is performed by a 74FCT138A, a 3-to-8 decoder, using the latched CAS address bits. The decoder is wired to supply eight device selects when A31 is set. The selects are enabled during $\overline{\text{CAS}}$ active. They are decoded from A30 and A29 when the DRAM memory group is involved and from A20 and A21 when the SRAM memory group is involved. Since the EPROM is 8-bit-wide, the selects are decoded from A18 and A19 when accessing the EPROM. The device select with A31 set and the other decoded address bits clear is used to select the EPROM as a boot-only memory device.

The EPROM must be programmed with 0xa5 at location 0x80000003 (typically 0x000000a5 at location 0x80000000). The memory groups must be configured with timing appropriate for the devices used, `mg0bw` and `mg1bw` set to zero (cell wide), `mg0ds` set to 0x0f (SRAM), and `mg1ds` set to 0x02 (256K DRAM). Since RAS is not used to latch the I/O address, `msras31d`, `mshacd` and `msexa31hac` can be set to reduce the number of RAS cycles involved in I/O.

ADVANCE INFORMATION

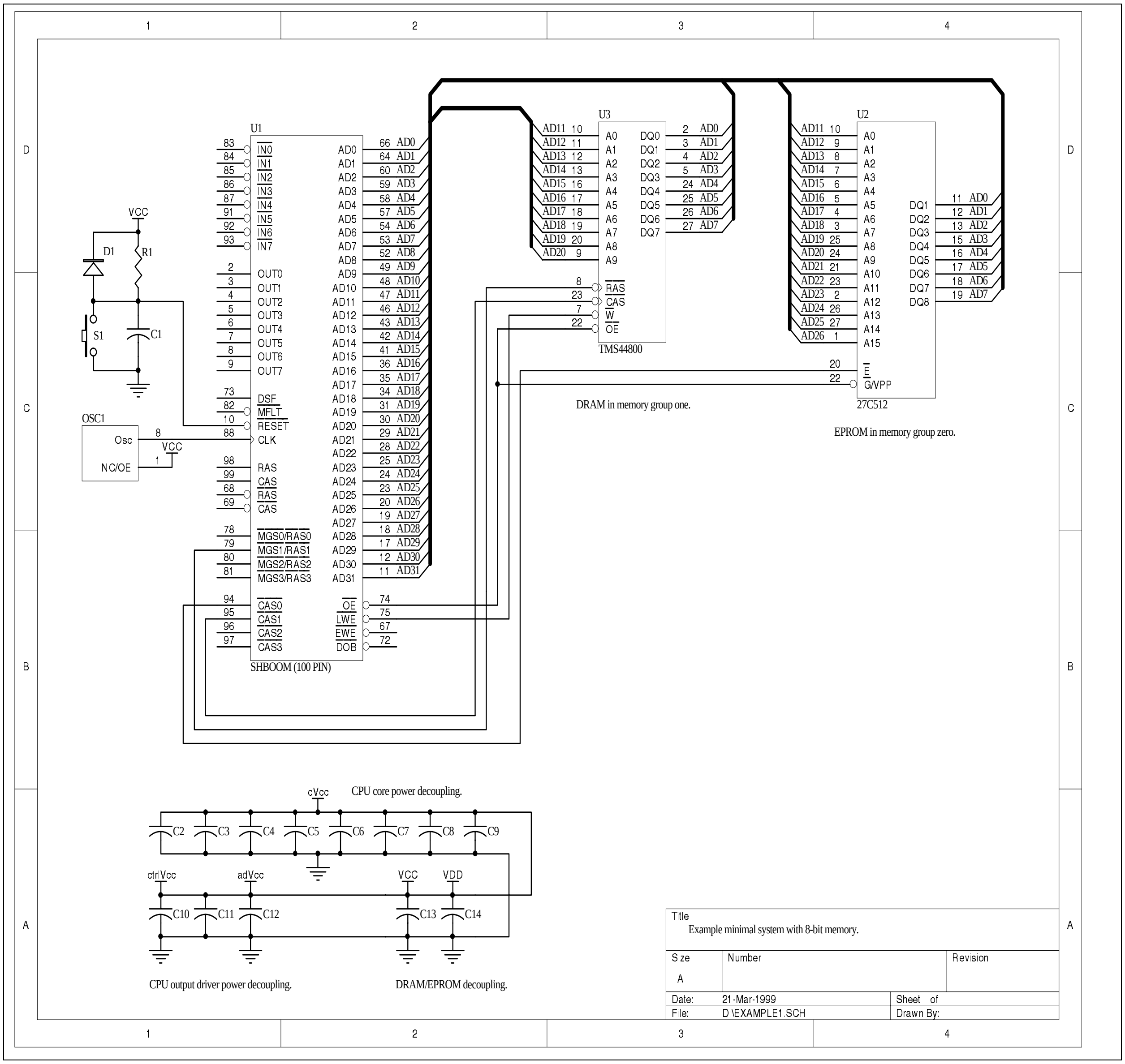
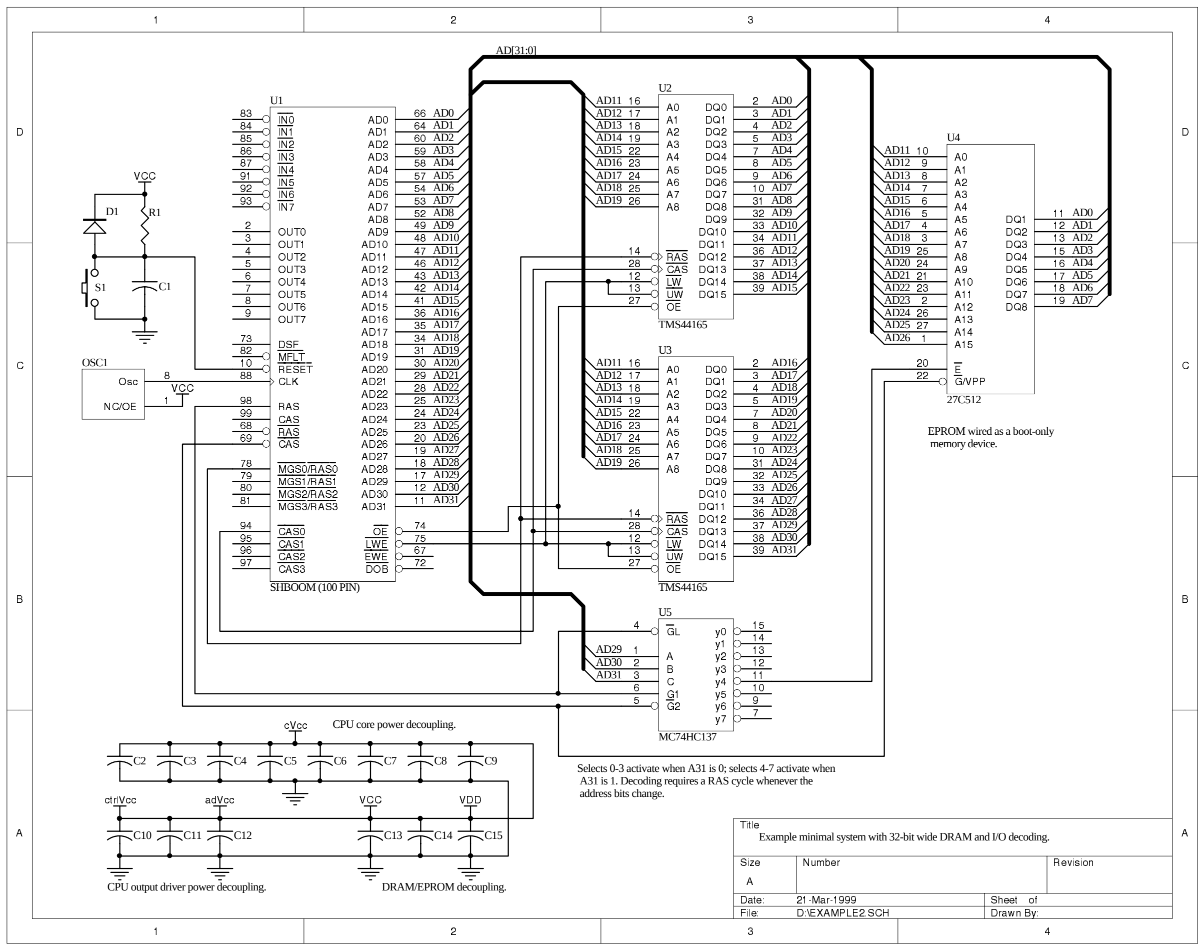


Figure 69. Example Minimal System with 8-bit Memory



EPROM wired as a boot-only memory device.

Selects 0-3 activate when A31 is 0; selects 4-7 activate when A31 is 1. Decoding requires a RAS cycle whenever the address bits change.

CPU core power decoupling.

CPU output driver power decoupling.

DRAM/EPROM decoupling.

Title
Example minimal system with 32-bit wide DRAM and I/O decoding.

Size A
Number
Revision
Date: 21-Mar-1999
Sheet of
File: D:\EXAMPLE2.SCH
Drawn By:

Figure 70. Example Minimal System with 32-bit DRAM and I/O Decoding

ADVANCE INFORMATION

ADVANCE INFORMATION

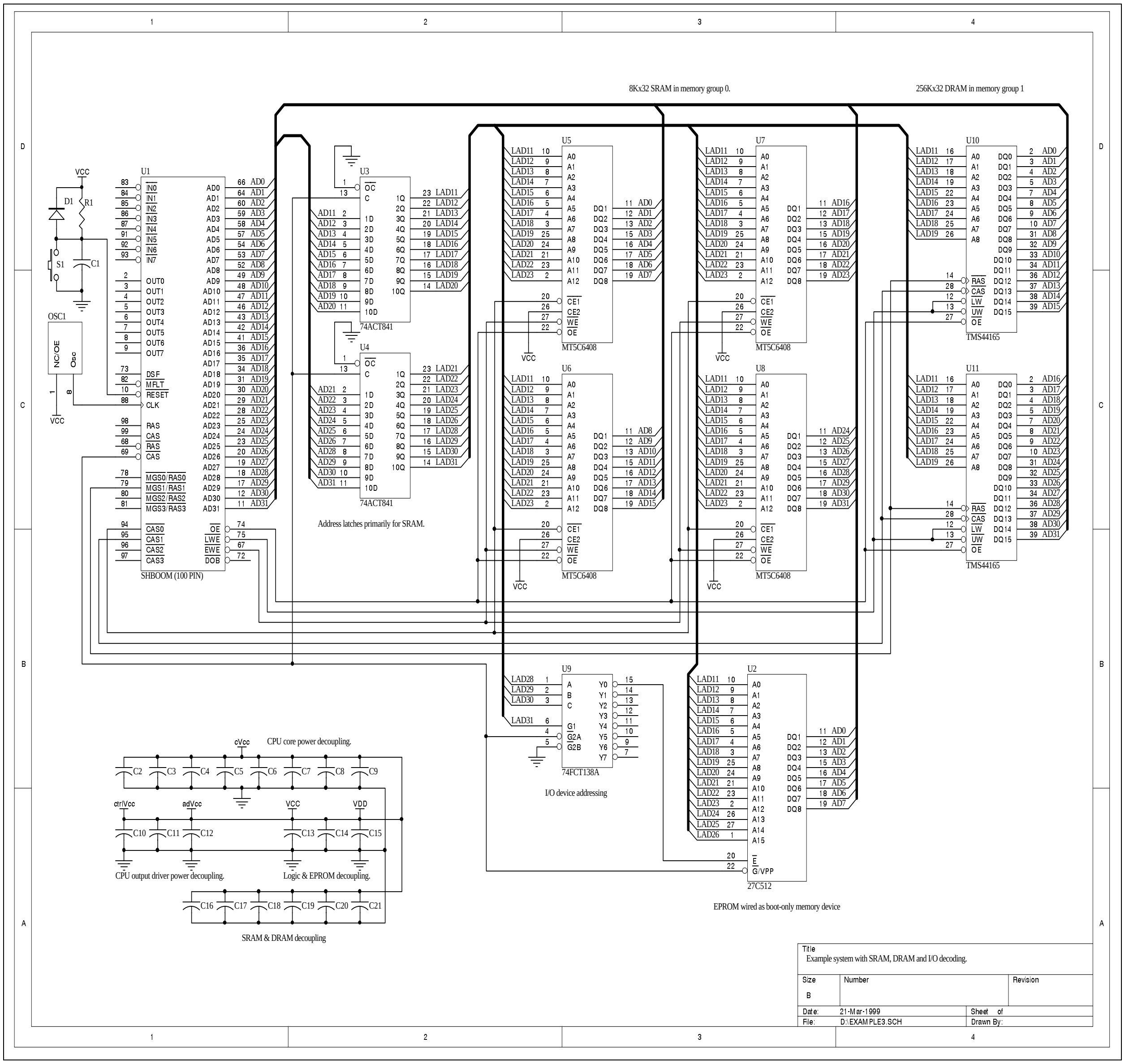


Figure 71. Example System with SRAM, DRAM and I/O Decode

```
                        ;;------------------------------------------------------------------------------
                        ;;====================== PSC1000 Boot-only EPROM loader ======================
                        ;;------------------------------------------------------------------------------
                        ;;
                        ;; bootcode.s
                        ;;
                        ;; Copyright (c) 1996 Patriot Scientific Corporation.  All rights reserved.
                        ;;
                        ;; This file contains the programming to configure a PSC1000 system and copy
                        ;; the system programming from EPROM into RAM.
                        ;;
                        ;;------------------------------------------------------------------------------

                                .opt            b.1,llbl

                                .opt            noincl
                                .include        "onchip.def"
                                .include        "sysmem.def"
                                .opt            incl,llbl

                                .import         start, load_tbl, config_data, config_entries
                                .import         lstk_org, ostk_org, astk_org

          [=000186a0]           tenKHz_per_ns   = 100000        ; scaling factor

                                ; Memory timing for VPU calculations (adjust to match DEMOCFG.S)
                                ; The "+ 2" below is for bus request in addition to the programmed
                                ; bus transaction time. These times depend on the system design and
                                ; memories used.

          [=00000006]           CAS0 = (4 + 2)                  ; clocks per CAS for group 0
          [=0000000b]           RAS0 = CAS0 + 5                 ; clocks per RAS for group 0

          [=00000007]           CAS1 = (5 + 2)                  ; clocks per CAS for group 1
          [=0000000d]           RAS1 = CAS1 + 6                 ; clocks per RAS for group 1

          [=0000000d]           RRAS = RAS1                     ; clocks for refresh
```

```
41 T                             ;;-----------------------------------------------------------------------------
42 T                             ;;
43 T                             ;; Reset Vectors
44 T                             ;;
45 T                             ;;-----------------------------------------------------------------------------
46 T
47 T                             ; On startup, PSC1000 looks at the first cell of the EPROM to decide if the
48 T                             ; boot device is a boot-only device (accessed first), or a memory device.
49 T                             ; PSC1000 performs a byte-wide memory bus transaction (four byte reads)
50 T                             ; starting at address 0x80000000 (without using -OE) to look for a 0xA5 from
51 T                             ; a boot device at address 0x80000003. If 0xA5 is not found then the boot
52 T                             ; device did not respond, OED is cleared (to allow _OE active on subsequent
53 T                             ; bus transactions) and the boot device is thus assumed to be normal memory
54 T                             ; that requires -OE for access.
55 T
56 T                             ; In either case, the VPU then begins execution at memory location 0x80000004.
57 T                             ; The VPU must initialize DRAM with refresh cycles and maintain refresh while
58 T                             ; the data and programs are distributed to memory. The VPU will be reset to
59 T                             ; run its application program long before the MLOOP it is executing expires.
60 T
61 T                             ; Once the VPU executes DELAY the MPU will begin execution at location
62 T                             ; 0x80000008. The branch goes to group 3 so that groups 0,1, & 2 can be
63 T                             ; configured for the application while execution continues out of group 3.
64 T                             ; Group 3 can be reconfigured later, if needed. Until msgsm is set during
65 T                             ; configuration, group3 cannot be selected.
66 T
67 T                             ; The group3 addressing for the EPROM is set in the linker.
68 T
69 T                                     .text   boot
     SECTION: boot
70 T 8c000000  00 00 00 a5               .long   boot_only           ; boot_only device activates w/o -OE
71 T
72 T                                     .opt    vpu
73 T 8c000004  33 00 00 04               jump    VPU_POR_init        ; VPU hardware reset vector
74 T
75 T                                     .opt  mpu
76 T 8c000008  4f 4b                     br      [#MPU_init]         ; MPU hardware reset vector
77 T 8c00000a        30 30               .quad   4
   T 8c00000c  8c 00 00 24
78 T
79 T
```

```
 80 T                                ;;------------------------------------------------------------------------
 81 T                                ;;
 82 T                                ;; Power-On-Reset VPU code
 83 T                                ;;
 84 T                                ;;------------------------------------------------------------------------
 85 T
 86 T                                ; DRAM typically requires 8 refresh cycles before it can be used, plus
 87 T                                ; periodic refresh cycles to continue operating. For 1Mb DRAM and
 88 T                                ; larger sizes, the typical refresh requirement is equivalent to 512 refresh
 89 T                                ; cycles every 8 ms. The code below supplies 8 refresh cycles to initialize
 90 T                                ; the DRAM, and then executes refresh cycles with appropriate delays to
 91 T                                ; maintain the DRAM data.
 92 T
 93 T                                ; NOTE FOR MMB WIRED SYSTEMS REGARDING REFRESH INITIALIZATION
 94 T                                ; The system boots in SMB mode. If the system is wired for MMB mode, RAS
 95 T                                ; cycles (refresh and otherwise) will not occur on the the selected memory
 96 T                                ; group (due to the MMB wiring) until MMB is set by the MPU. However, despite
 97 T                                ; the MMB wiring, non-selected memory groups will experience properly formed
 98 T                                ; RAS cycles. After MMB is set, due to the default configuration causing a
 99 T                                ; RAS cycle on every memory access, RAS cycles occuring due to memory
100 T                                ; accesses on the selected memory group will supply the required RAS or
101 T                                ; refresh initialization cycles. Of course, once MMB mode is set, subsequent
102 T                                ; refresh cycles will occur properly on all enabled memory groups.
103 T
104 T
105 T                                        .text
    SECTION: .text
106 T                                        .opt    vpu
107 T
108 T 8c000010                       VPU_POR_init::
109 T
110 T 8c000010  10                           refresh
111 T 8c000011     10                        refresh
112 T 8c000012        10                     refresh
113 T 8c000013           10                  refresh
114 T
115 T 8c000014  10                           refresh
116 T 8c000015     10                        refresh
117 T
118 T                                        ; During power-on configuration by the MPU, the described timing
119 T                                        ; below does not apply; the memory cycles are much slower until
120 T                                        ; configured. That means there will be fewer refresh cycles than
121 T                                        ; described, but, that's ok because there is no data in the DRAM to
122 T                                        ; maintain yet anyway.
123 T
124 T                                        ; This code below is executed only from the boot EPROM. A different
125 T                                        ; copy or version is used to run the VPU application from R/W RAM.
126 T
127 T                                        ; A typical 256Kx4 DRAM requires 512 refreshes every 8 ms. That means
```

```
128 T                                    ; we need a refresh every 15.625 us, or a total loop time below of
129 T                                    ; 31.250 us since we do two refreshes per loop below.
130 T
131 T                                    ; Assuming a RAS cycle with the bus request takes 13 clocks, the loop
132 T                                    ; below takes 13 + 13 + 2 + delay + 1, or 29 + delay CPU-clock
133 T                                    ; cycles to execute. 31.250 us - 29 CPU-clocks is the delay time we
134 T                                    ; must use.
135 T
136 T                                    ; Compute values for POR_Refresh_Loop.
137 T
138 T                                    ; Total time to be taken by one loop iteration in nanoseconds.
139 T                                    ; rounded down.
140 T           [=007a1200]              refresh_interval = 8000000      ; time in ns
141 T           [=00000200]              refreshes_per_interval = 512
142 T           [=00000002]              refreshes_per_loop = 2
143 T
144 T           [=00007a12]              POR_loop_ns = (refresh_interval * refreshes_per_loop) / refreshes_per_interval
145 T
146 T                                    ; Number of CPU_clock cycles during POR_loop_ns.
147 T           [=000009c4]              POR_loop_clocks = (POR_loop_ns * (__CLOCK_KHZ__/10)) / tenKHz_per_ns
148 T
149 T                                    ; Number of CPU-clocks cycles required by instructions except delay
150 T                                    ; time.
151 T           [=0000001d]              POR_overhead_clocks = RRAS * 2 + 2 + 1
152 T
153 T                                    ; CPU-clock cycle delay value required.
154 T           [=000009a7]              POR_delay = POR_loop_clocks- POR_overhead_clocks
155 T
156 T 8c000016        27                 ld      #POR_delay,g7
157 T 8c000017           2f              ld      #-1,g15                 ; sufficient to load system
    T 8c000018  00 00 09 a7
    T 8c00001c  ff ff ff ff
158 T
159 T 8c000020                       POR_refresh_loop::
160 T
161 T 8c000020  10                       refresh                         ; RRAS
162 T 8c000021     10                    refresh                         ; RRAS
163 T 8c000022        57                 delay   g7                      ; 2 + delay
164 T 8c000023           7f              mloop   g15,POR_refresh_loop    ; 1
165 T
166 T
```

```
167 T                             ;;--------------------------------------------------------------------------------
168 T                             ;;
169 T                             ;; MPU startup code
170 T                             ;;
171 T                             ;;--------------------------------------------------------------------------------
172 T
173 T                                     .text
    SECTION: .text
174 T                                     .opt    mpu
175 T
176 T 8c000024                    MPU_init::
177 T
178 T                             ;
179 T                             ; Set up stacks.
180 T                             ;
181 T 8c000024  4f bd                     move    #lstk_org - 8, la
182 T 8c000026        4f bc               move    #ostk_org - 8, sa
    T 8c000028  00 03 bf f8
    T 8c00002c  00 03 af f8
183 T 8c000030  4f 58                     move    #astk_org, g8           ; auxillary stack for C
184 T
185 T                             ; Now load up to configure the chip
186 T
187 T 8c000032        4f b8               move    #config_data,x
    T 8c000034  00 03 a0 00
    T 8c000038  8c 00 00 a4
188 T 8c00003c  90 b4                     move    #(config_entries - 1).b,ct
189 T
190 T                                     ; Set edge rates on strobes (special case, the data goes to d16..31 not 0..16)
191 T
192 T 8c00003e        49 11               ld      [x++]
193 T 8c000040  99                        split
194 T 8c000041     b2                     xcg
195 T 8c000042        b0                  sto     []
196 T 8c000043           b3               pop
197 T
198 T                                     ; Configure all the other registers
199 T
200 T 8c000044                    config_loop::
201 T
202 T 8c000044  49                        ld      [x++]
203 T 8c000045     99                     split
204 T
205 T                                     ; group sto[] with dbr so we don't change a memory
206 T                                     ; control register during an access (dbr holds off prefetch)
207 T 8c000046        30 30               .quad   4
208 T 8c000048  b0                        sto     []
209 T 8c000049     b3                     pop
210 T 8c00004a        1f ff               dbr     config_loop
```

```
 211 T
 212 T                             ;
 213 T                             ; Distribute EPROM to memory
 214 T                             ;
 215 T
 216 T 8c00004c  08 00 00 06               call.3  load_tbl
 217 T
 218 T
 219 T                             ; Prepare to clear OED bit and start application
 220 T
 221 T 8c000050  4f                        push.l  #start                  ; MPU application code start
 222 T
 223 T 8c000051     4f                     push    #miscb
 224 T 8c000052        92                  push                            ; keep addr for sto []
 225 T 8c000053           96               ldo     []
     T 8c000054  00 00 04 00
     T 8c000058  00 00 01 00
 226 T 8c00005c  90 e9                     iand    #oed                    ; reset OE disable bit
 227 T
 228 T 8c00005e        b2 10               xcg                             ; addr on top for below
 229 T
 230 T 8c000060                            .quad   3                       ; Keep up to br [] together
 231 T 8c000060  b0                        sto     []                      ; turn on OE
 232 T 8c000061     b3                     pop
 233 T 8c000062        4b                  br      []                      ; go to MPU application
 234 T
 235 T                                     ; We do the above steps in one instruction group so that if there is
 236 T                                     ; memory in group 3 it won't collide with our EPROM
 237 T
 238 T 8c000063                            .end

Total Errors:      0
Total Warnings:    0
```

```
 1 T
 2 T                          ;;--------------------------------------------------------------------------
 3 T                          ;;
 4 T                          ;; Example configuration data
 5 T                          ;;
 6 T                          ;; democfg.s
 7 T                          ;;
 8 T                          ;; Note tht very little of the configuration data below is "typical".
 9 T                          ;; The exact values always depend on the board design, device types, and
10 T                          ;; speeds of the memory components used.
11 T                          ;;
12 T                          ;; Copyright (c) 1998 Patriot Scientific Corporation.  All rights reserved.
13 T                          ;;
14 T                          ;;--------------------------------------------------------------------------
15 T
16 T                                  .opt            noincl
17 T                                  .include        "onchip.def"
18 T                                  .opt            incl
19 T                                  .export         config_data, config_entries
20 T
21 T                                  .text
   SECTION: .text
22 T
23 T 8c0000a4                config_data::
24 T                                  ; Must be first entry in table.
25 T 8c0000a4  00 00 07 c0            .long (drivers | 0x00000000)
26 T
27 T 8c0000a8  01 60 0c 00            .long (msgsm << 16 | 0x0c00)
28 T 8c0000ac  01 00 00 38            .long (miscb << 16 | pkgio | oed | mg3bw)
29 T
30 T 8c0000b0  00 e0 00 91            .long (misca << 16 | 0x91)
31 T 8c0000b4  01 80 f7 4f            .long (mgds << 16 | 0xf74f)
32 T 8c0000b8  01 a0 00 93            .long (miscc << 16 | msexa31hac | pkgmflt | 3)
33 T
34 T 8c0000bc  02 40 24 30            .long (mg0casbt << 16 | 1 << mgbtcas_pos | 2 << mgbtdob_pos | 3 << mgbtcast_pos | 0 << mgbt
                          ewea_pos | 0 << mgbtlwea_pos | 0 << mgbteoe_pos | 0 << mgbtewe_pos)
35 T 8c0000c0  01 c0 00 00            .long (mg0ebt   << 16 | 0 << mgebtsum_pos | 0 << mgebtdobe_pos | 0 << mgebtcase_pos)
36 T 8c0000c4  02 c0 0a 42            .long (mg0rasbt << 16 | 5 << mgbtrast_pos | 2 << mgbtras_pos | 1 << mgbtrhld_pos | 0 << mgb
                          teras_pos)
37 T
38 T 8c0000c8  02 60 46 44            .long (mg1casbt << 16 | 2 << mgbtcas_pos | 3 << mgbtdob_pos | 4 << mgbtcast_pos | 0 << mgbt
                          ewea_pos | 1 << mgbtlwea_pos | 0 << mgbteoe_pos | 0 << mgbtewe_pos)
39 T 8c0000cc  01 e0 00 00            .long (mg1ebt   << 16 | 0 << mgebtsum_pos | 0 << mgebtdobe_pos | 0 << mgebtcase_pos)
40 T 8c0000d0  02 e0 0c 62            .long (mg1rasbt << 16 | 6 << mgbtrast_pos | 3 << mgbtras_pos | 1 << mgbtrhld_pos | 0 << mgb
                          teras_pos)
41 T
42 T 8c0000d4  02 80 48 60            .long (mg2casbt << 16 | 2 << mgbtcas_pos | 4 << mgbtdob_pos | 6 << mgbtcast_pos | 0 << mgbt
                          ewea_pos | 0 << mgbtlwea_pos | 0 << mgbteoe_pos | 0 << mgbtewe_pos)
43 T 8c0000d8  02 00 00 00            .long (mg2ebt   << 16 | 0 << mgebtsum_pos | 0 << mgebtdobe_pos | 0 << mgebtcase_pos)
44 T 8c0000dc  03 00 0c 62            .long (mg2rasbt << 16 | 6 << mgbtrast_pos | 3 << mgbtras_pos | 1 << mgbtrhld_pos | 0 << mgb
                          teras_pos)
45 T
46 T 8c0000e0  02 a0 6b 70            .long (mg3casbt << 16 | 3 << mgbtcas_pos | 5 << mgbtdob_pos | 23 << mgbtcast_pos | 0 << mgb
                          tewea_pos | 0 << mgbtlwea_pos | 0 << mgbteoe_pos | 0 << mgbtewe_pos)
47 T 8c0000e4  02 20 01 03            .long (mg3ebt   << 16 | 4 << mgebtsum_pos | 0 << mgebtdobe_pos | 3 << mgebtcase_pos)
```

```
  48 T 8c0000e8  03 20 08 23            .long (mg3rasbt << 16 | 4 << mgbtrast_pos | 1 << mgbtras_pos | 1 << mgbtrhld_pos | 1 << mgb
                                teras_pos)
  49 T
  50 T           [=00000012]            config_entries  = ((. - config_data)/4)
  51 T
  52 T
  53 T

Total Errors:      0
Total Warnings:    0
```

Electrical Characteristics

Power and Grounding

The PSC1000 CPU is implemented in CMOS for low average power requirements. However, the high clock-frequency capability of the CPU can require large switching currents of as much as eleven amperes, depending on the output loading. Thus, all V_{CC} and V_{SS} must be connected to planes within the PWB (printed wire board) for adequate power distribution.

The switching current required by cV_{CC} and cV_{SS} is characterized by the internal clock and output driver pre-drivers. The internal clock requires approximately 500 mA with significant 5-GHz frequency components every clock transition. The output driver pre-drivers require as much as 3 A with significant 1-GHz frequency components every output transition. The CPU has on-chip capacitance to supply the high-frequency components. Package diagrams indicate which of cV_{CC} and cV_{SS} are closest to the internal clock drivers and PLL.

The switching current required by ctrlV_{CC} and ctrlV_{SS} is characterized by the supplied output drivers and externally attached loads. Assuming a worst-case average load of 100 pF and 16 pins switching at once, these drivers require 2.67 A with significant 300-MHz frequency components every output transition. Switching-current requirements reduce substantially linear manner with a reduction in external loading.

The switching power required by adV_{CC} and adV_{SS} is characterized by the supplied output drivers and externally attached loads. Assuming a worst-case average load of 100 pF and 32 pins switching at once, these drivers require 5.33 A with significant 300-MHz frequency components every output transition. Switching-current requirements reduce substantially linear manner with a reduction in external loading.

Power Decoupling

Due to the switching characteristics discussed above, power decoupling at the CPU is typically required. Surface-mount capacitors with low ESR are preferred. Generally, smaller-physically-sized capacitors have better frequency characteristics (i.e., lower series inductance, resulting in higher self-resonance frequency) than larger physically-sized capacitors. PWB board construction using FR-4 with power and ground layer spacing of 10 mils or less supplies the best high-frequency decoupling (typically about 100 pF/in^2). Connections to the power and ground planes must be as short as possible. Proper power and ground plane connections and appropriate decoupling also reduces EMC problems.

The charge supply required from the decoupling capacitors can be calculated from the relation $C = I/(f\Delta V)$, where I is the current required, f is the frequency, and ΔV is the allowed voltage drop, typically .1 V. Thus, cV_{CC} and cV_{SS} require 1000 pF for the internal clock and .03 μF for the output driver pre-drivers, while ctrlV_{CC} and ctrlV_{SS} together with adV_{CC} and adV_{SS} require .24 μF. These requirements can be met with four .1 μF X7R capacitors, one on each side of, and on the same side of the PWB, as the CPU, as close to the package as practical.

Note that mounting capacitors on the same PWB surface as the PSC1000 CPU package can allow connecting traces of about 25 mils in length, while mounting capacitors on the opposite PWB surface requires traces of over 100 mils in length. At the switching frequencies listed, the difference in trace lengths creates significant differences in decoupling effectiveness. The package and capacitor power and ground connections would optimally be fabricated with VIP (via-in-pad), if possible, for the same reasons.

Connection Recommendations

All output drivers are designed to directly drive the heavy capacitive loads of memory systems, thus minimizing the external components and propagation delays associated with buffering logic. However, with increased loading comes increased power dissipation, and trade-offs must be made to ensure that the PSC1000 CPU operating temperature does not exceed operating limitations. Systems with heavy CPU bus loads might require heat sinks or forced air ventilation. Note that reducing output driver current does not reduce total power dissipation because power consumption is dependent on output loading and not

ADVANCE INFORMATION

on signal transition edge rates. See Figure 50, page 154.

To reduce system cost, most inputs have internal circuitry to provide a stable input voltage if the input is unused. Thus, most unused inputs do not require pull-ups.

Clock

The PSC1000 CPU requires an external CMOS oscillator at one-half the processor frequency. The oscillator is doubled internally (CPU-clock cycle) to operate the MPU and the VPU, and doubled again to provide fine-granularity programmable bus timing (2X-CPU-clock cycle).

Inexpensive oscillators typically have guaranteed duty cycles of only 55/45 or 60/40. The narrower half of the clock cycle represents the clock period at which the CPU appears to be operating. An 80-MHz CPU is thus be limited with a 60/40 oscillator to 64 MHz (32 MHz externally), because with a 64 MHz CPU-clock the 40% clock period is 12.5 ns. Thus oscillator selection and qualification is an important factor in processor performance.

The CPU-clock frequency selected depends on application and system hardware requirements. A clock frequency might be selected for the VPU to produce appropriate application timing, or for the MIF to optimize bus timing. For instance, if the system requires a 40 ns bus cycle, it might be more efficient to operate at 75 MHz with a three CPU-clock cycle long bus cycle (40 ns) than to operate at 80 MHz with a four CPU-clock cycle long bus cycle (50 ns).

Absolute Maximum Ratings

Table 60. Absolute Maximum Ratings

Characteristic	Symbol	Min	Max	Unit	Notes
Core Logic Supply Voltage	cV_{CC}	-0.5	+7.0	V	1
Control Driver Supply Voltage	$ctrlV_{CC}$	-0.5	+7.0	V	1
AD Driver Supply Voltage	adV_{CC}	-0.5	+7.0	V	1
DC Input Voltage	V_I	-0.5	+7.0	V	
DC Output Voltage	V_O	-0.5	+7.0	V	output Hi-Z
		-0.5	V_{CC}+0.5	V	output driven
DC Input Diode Current	I_{IK}		-50	mA	$V_I < V_{SS}$
DC Output Diode Current	I_{OK}		-50	mA	
			+50	mA	
Storage Temperature	T_{STG}	-65	+150	°C	
Case Temperature Under Bias	T_C	-65	+125	°C	
Operating Junction Temperature	T_J	-65	+150	°C	

Notes:
Stressing the device beyond Absolute Maximum Ratings can cause the device to sustain permanent damage. Operating the device beyond Operating Conditions is not recommended and can reduce device reliability. Functional operation at Absolute Maximum Ratings is not guaranteed.
1. cV_{SS}, $ctrlV_{SS}$ and adV_{SS} are required to be at the same potential.

Operating Conditions

Table 61. Operating Conditions

Characteristic	Symbol	Min	Max	Unit	Notes
Core Logic Supply Voltage	cV_{CC}	3.0	5.5	V	
Control Driver Supply Voltage	$ctrlV_{CC}$	3.0	5.5	V	
AD Driver Supply Voltage	adV_{CC}	3.0	5.5	V	
Input Voltage	V_I	0	5.5	V	
Output Voltage	V_O	0	5.5	V	output Hi-Z
		0	V_{CC}	V	output driven
Output Current	I_{OH}		180	mA	1
	I_{OL}		180	mA	1
Input Clock	f_C		80	MHz	
Case Temperature Under Bias	T_C	0	+85	°C	
Free-Air Operating Temperature	T_A	-40	+85	°C	
Input Edge Rate	$\Delta t/\Delta V$	0	.1	ns/V	2

Notes:
1. Assumes the maximum of three driver sections enabled (at 60 ma each) during signal transitions only.
2. $V_{IN} = V_{IH\text{-}MIN} - V_{IL\text{-}MAX}$ monotonic

DC Specifications

Table 62. DC Specifications

Characteristic	Sym-bol	Min	Max	Unit	Notes
Input Low Voltage	V_{IL}	0	0.8	V	TTL
		0	1.8		CMOS
Input High Voltage	V_{IH}	2.0	cV_{CC}	V	TTL
		3.0	cV_{CC}	V	CMOS
Output Low Voltage	V_{OL}		0.4	V	I_{OL} = 12 mA
Output High Voltage	V_{OH}	2.4		V	I_{OL} = 45 mA
		V_{CC} - 0.4			I_{OL} = 12 mA
Input Leakage Current	I_{LI}		±10	µA	$0<=V_{IN}<=V_{CC}$
Output Leakage Current	I_{OL}		±10	µA	$0.4<V_{OUT}<V_{CC}$
Power Supply Current	I_{CC}		100	mA	1
Input Capacitance	C_{IN}		8	pF	2
I/O or Output Capacitance	C_{OUT}		10	pF	2

Notes:

1. Under normal operation. Specially constructed programs can draw substantially more current.
2. f_C = 1 MHz. Capacitance values are not tested.

Table 63. Input Characteristics

PIN	Input Level	Impedance, Ohms		Notes
		Minimum	Maximum	
$\overline{\text{AD}}$[31:0]	TTL	25K	50K	repeater, V_{IH}
		15K	30K	repeater, V_{IL}
CLK	CMOS	1M		must be driven
$\overline{\text{IN}}$[7:0]	TTL	1M		
$\overline{\text{MFLT}}$	TTL Schmitt trigger	250K	500K	pull-up
$\overline{\text{RESET}}$	CMOS Schmitt trigger	250K	500K	pull-up

ADVANCE INFORMATION

AC Characteristics

Table 64. CPU-Clock and 2X-CPU-Clock

No.	Characteristic	Symbol	Min	Max	Unit	Notes
1	Clock period		12.5		ns	
	Stabilization of PLL			10,000	CPU-Clocks	3

Notes:
1. CPU-Clock generated from $\overline{\text{CLK}}$ edges.
2. 2X-CPU-Clock intermediate pulse generated with a PLL.
3. Required after $\overline{\text{RESET}}$ inactive before dependance on 2X-CPU-Clock timing.

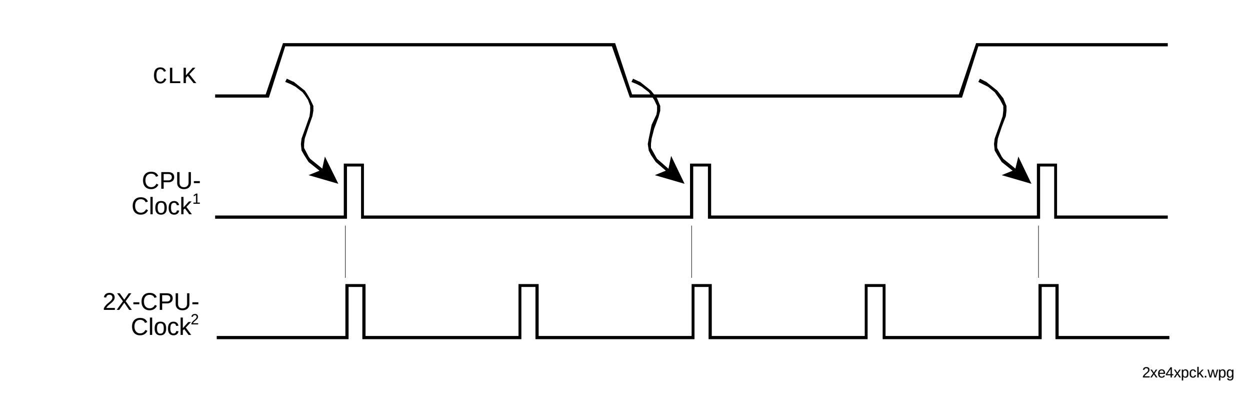


Figure 72. CPU-Clock and 2X-CPU-Clock

Table 65. CPU Reset Timing

No.	Characteristic	Symbol	Min	Max	Unit	Notes
1	Reset active time, pin		2		CPU-clocks	
2	Reset active to AD Hi-Z		3	4	CPU-clocks	2
			10	11	CPU-clocks	3
3	Reset inactive to AD active and start of RAS prefix for first bus cycle		4	5	CPU-clocks	
4	Reset active to signals inactive		3	4	CPU-clocks	2
			10	11	CPU-clocks	3

Notes:
1. AD have bus repeaters that hold the last bus state when not driven by the CPU or an external device.
2. When reset is sampled from $\overline{\text{RESET}}$.
3. When reset is sampled from AD8.
4. States occur from subsequent bus cycle and program execution.

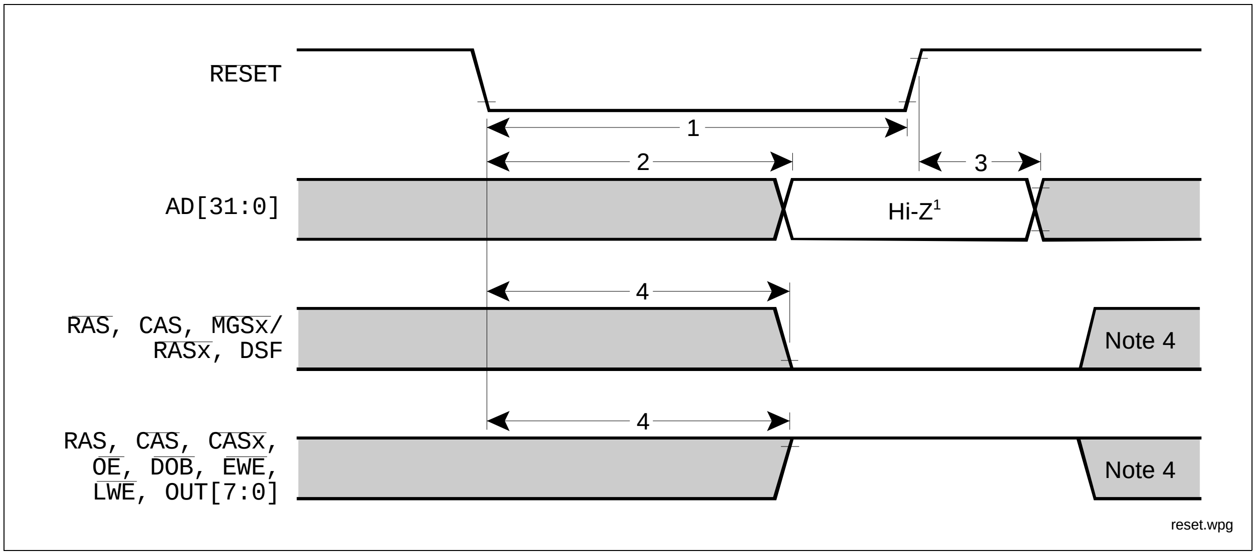


Figure 73. CPU Reset Timing

ADVANCE INFORMATION

Table 66. Memory Read and Write Timing

No.	Characteristic	Min	Max	Unit	Notes
1	RAS Prefix	1 + mgbtras + mgbtrhld		CPU-clocks	5
2	$\overline{\text{RAS}}$ inactive	(mgbtras · 2) – mgbteras		2X-CPU-clocks	5
3	RAS address hold	(mgbtrhld · 2) + mgbteras		2X-CPU-clocks	5
4	RAS prefix start to $\overline{\text{RAS}}$ rise	1		CPU-clocks	
5	End of bus cycle to start of next	0		CPU-clocks	
6	CAS part	mgbtcast + mgebtdobe + mgebtcase		CPU-clocks	5
		mgbtcast + ioebtdobe + ioebtcase		CPU-clocks	5,9
7	CAS part start to $\overline{\text{CAS}}$ fall	mgbtcas		2X-CPU-clocks	5
8	CAS part start to $\overline{\text{MGSx}}$ rise	time 6		CPU-clocks	
9	CAS part start to $\overline{\text{MGSx}}$ fall		3.75	ns	
10	$\overline{\text{MGSx}}$ inactive pulse width, RAS cycle	0		ns	3
11	RAS cycle start to $\overline{\text{MGSx}}$ fall		3.0	ns	
12	$\overline{\text{MGSx}}$ inactive pulse width, CAS cycle	0		ns	3
13	CAS part start to $\overline{\text{DOB}}$ rise, memory read	(mgbtcast · 2) – mgbteoe		2X-CPU-clocks	5
14	CAS part start to $\overline{\text{DOB}}$ fall	mgbtdob		2X-CPU-clocks	5
15	CAS part start to CAS address valid		5.0	ns	
16	$\overline{\text{DOB}}$ fall to address invalid	1.5		ns	
17	RAS prefix to address valid		2.25	ns	
18	RAS prefix end to RAS address invalid	2.0		ns	
19	RAS prefix end to CAS address valid		5.75	ns	
20	Data setup before $\overline{\text{DOB}}$ rise	16.0		ns	4,6
21	Data hold after $\overline{\text{DOB}}$ rise	0		ns	4,6
22	CAS part start to $\overline{\text{OE}}$ rise	time 13		2X-CPU-clocks	
23	CAS part start to $\overline{\text{OE}}$ fall	time 14		2X-CPU-clocks	
24	Previous cycle end to $\overline{\text{EWE}}$ rise		1.75	ns	
25	Previous cycle end to $\overline{\text{LWE}}$ rise		1.75	ns	

Table 66. Memory Read and Write Timing (continued)

No.	Characteristic	Min	Max	Unit	Notes
26	CAS part start to $\overline{\mathtt{DOB}}$ rise, memory write	(mgbtcast · 2) – mgbtewe		2X-CPU-clocks	5
27	$\overline{\mathtt{DOB}}$ fall to data valid		3.25	ns	4
28	$\overline{\mathtt{DOB}}$ rise to data not driven	1.0		ns	4
29	CAS part start to $\overline{\mathtt{EWE}}$ rise	time 26		CPU-clocks	
30	CAS part start to $\overline{\mathtt{EWE}}$ fall		8.5	ns	
31	$\overline{\mathtt{EWE}}$ inactive pulse width, RAS	2.5		ns	3
32	RAS prefix start to $\overline{\mathtt{EWE}}$ fall		6.0	ns	
33	$\overline{\mathtt{EWE}}$ inactive pulse width, CAS	2.5		ns	3
34	CAS part start to $\overline{\mathtt{EWE}}$ fall	mgbtcas		2X-CPU-clocks	5
35	CAS part start to $\overline{\mathtt{LWE}}$ rise	time 26		2X-CPU-clocks	
36	CAS part start to $\overline{\mathtt{LWE}}$ fall	mgbtdob + mgbtlwea + (mgebtdobe · 2)		2X-CPU-clocks	5
		mgbtdob + mgbtlwea + (ioebtdobe · 2)		2X-CPU-clocks	5,9
37	Previous cycle end to $\overline{\mathtt{OE}}$ rise		2.25	ns	

Notes:

1. AD have bus repeaters that hold the last bus state when not driven by the CPU or an external device.
2. Does not apply to byte-wide data transfers. See note 1.
3. Minimum applies when time 5 is minimum.
4. Time applies only to data transfers to the CPU.
5. Use decoded value of register fields for calculations.
6. If mgbteoe is set, data must be held until specified time relative to the next CPU-clock timing boundary. See Note 1.
7. $\overline{\mathtt{MGSx}}$ applies when mmb is set. $\overline{\mathtt{RASx}}$ applies when mmb is clear.
8. All CASes and RASes move appropriately.
9. Applies to bus cycles of I/O-channel bus transactions that involve the I/O device.

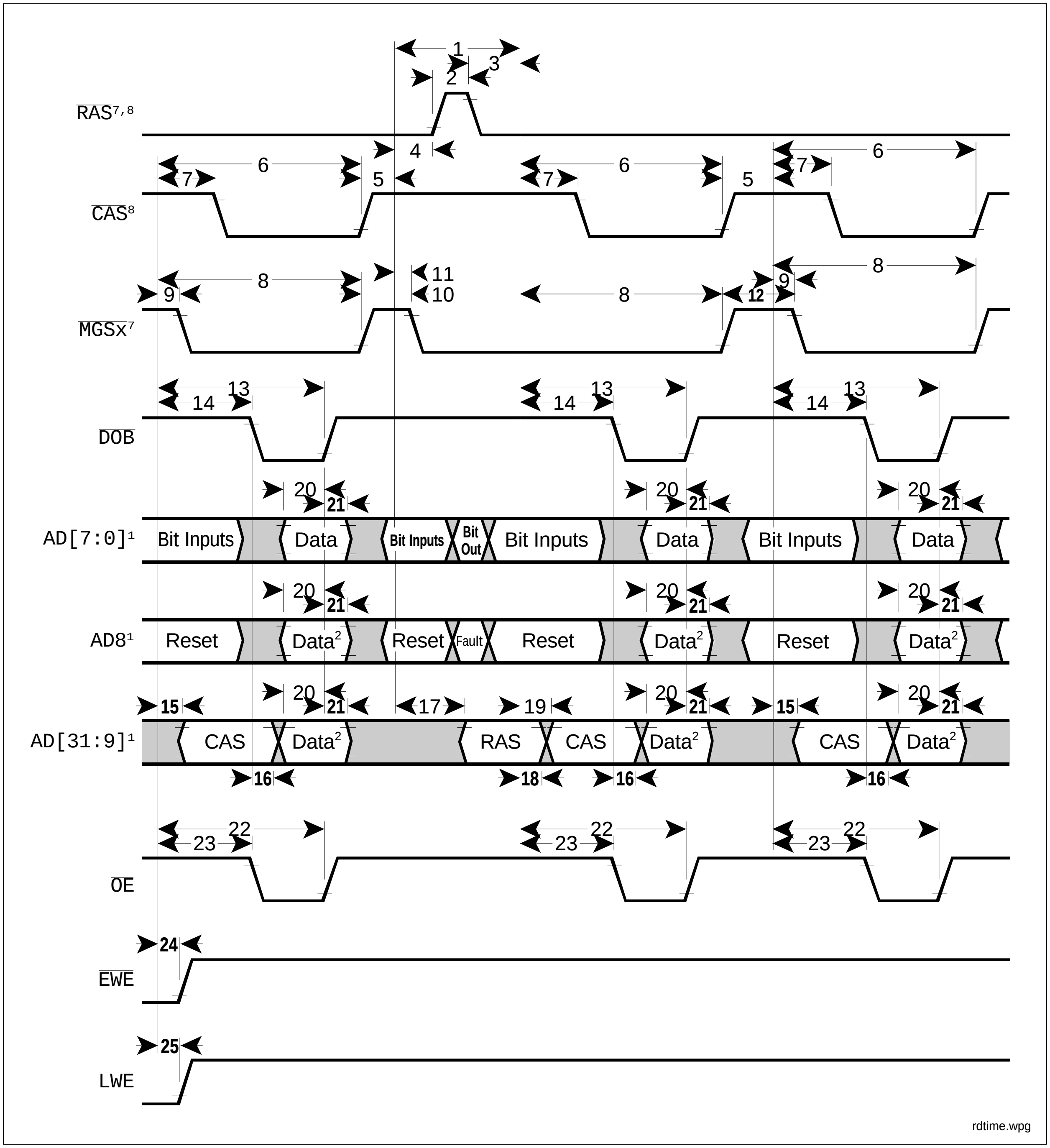


Figure 74. Memory Read Timing

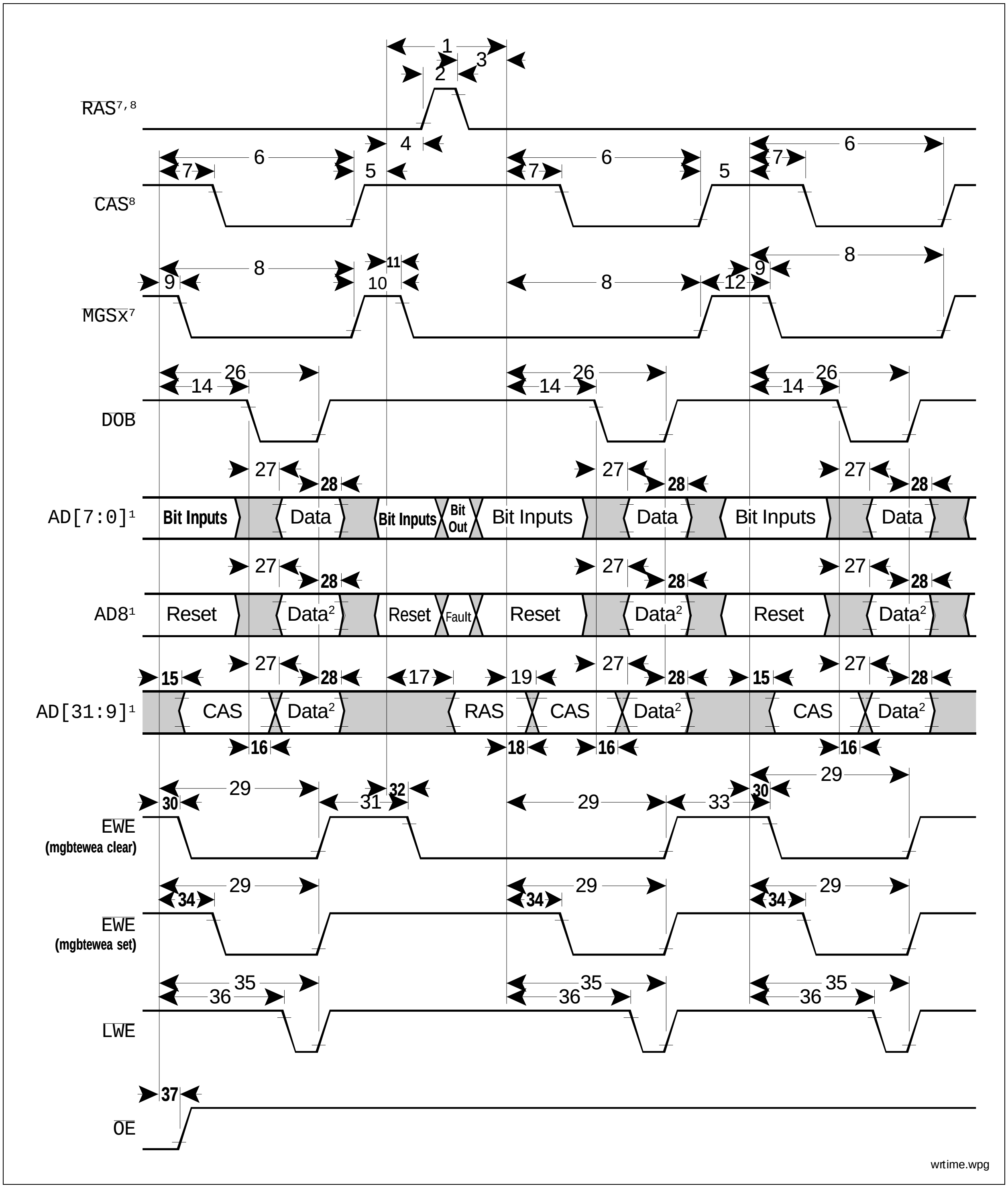


Figure 75. Memory Write Timing

ADVANCE INFORMATION

Table 67. Signal Coincidence Timing

No.	Characteristic	Symbol	Min	Max	Unit	Notes
1	$\overline{CAS}$ rise to $\overline{CASx}$ rise			3.0	ns	
2	$\overline{CAS}$ fall to $\overline{CASx}$ fall			3.25	ns	
3	$\overline{CAS}$ rise to CAS fall			.5	ns	
4	$\overline{CAS}$ fall to CAS rise			1.0	ns	
5	$\overline{RAS}$ rise to $\overline{RASx}$ rise			3.25	ns	
6	$\overline{RAS}$ fall to $\overline{RASx}$ fall			3.75	ns	
7	$\overline{RAS}$ rise to RAS fall			1.0	ns	
8	$\overline{RAS}$ fall to RAS rise			.5	ns	

Notes:

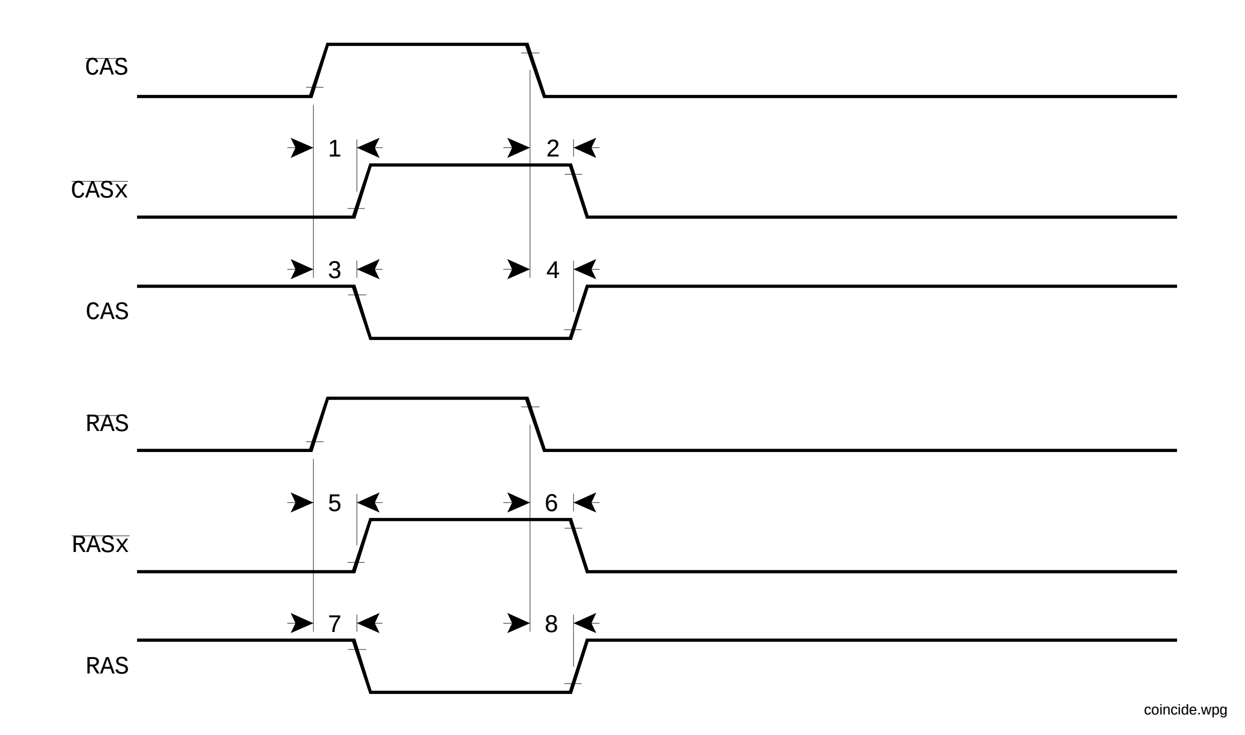


Figure 76. Signal Coincidence Timing

Table 68. Memory Fault Timing

No.	Characteristic	Symbol	Min	Max	Unit	Notes
1	$\overline{\texttt{MFLT}}$ setup		4.5		ns	7
2	$\overline{\texttt{MFLT}}$ hold		0		ns	7
3	Fault request setup		9.0		ns	7
4	Fault request hold		0		ns	7
5	$\overline{\texttt{EWE}}$ rise after $\overline{\texttt{RAS}}$ fall		(mgbtrhld · 2) + mgbteras		2X-CPU-clocks	8, 9

Notes:

1. $\overline{\texttt{MGSx}}$ applies when mmb is set.
2 $\overline{\texttt{RASx}}$ applies when mmb is clear.
3. $\overline{\texttt{MFLT}}$ is used for memory fault requests when pkgmflt is set.
4. AD8 is used for memory fault requests when pkgmflt is clear.
5. Appropriate timing references of $\overline{\texttt{RAS}}$ apply to RAS.
6. Conditions exist for time equivalent to the entire bus transaction.
7. Applies as if RAS had fallen at the next CPU-clock timing boundary.
8. Applies only to memory write cycles.
9. Use decoded value of register fields for calculation.

ADVANCE INFORMATION

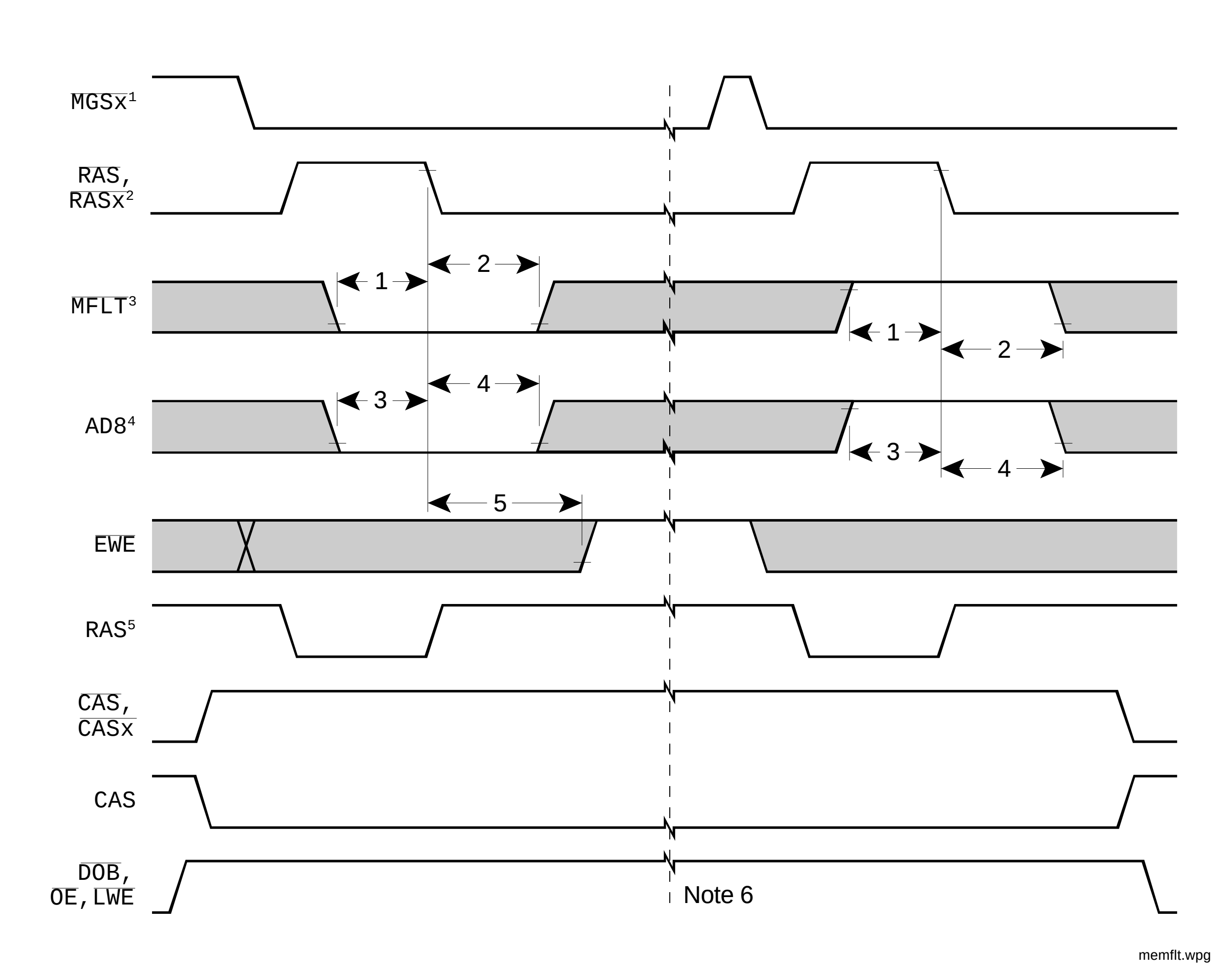


Figure 77. Memory Fault Timing

Table 69. Refresh Timing

No.	Characteristic	Symbol	Min	Max	Unit	Notes
1	Refresh cycle length		`1 + mgbtras + mgbtrhld + mgbtcast + mgebtdobe + mgbtcase`		CPU-clocks	4, 5
2	RAS cycle precharge		(`mgbtras` · 2) - `mgbteras`		2X-CPU-clocks	4, 5

Notes:

1. $\overline{\texttt{MGSx}}$ applies when `mmb` is set.
2 $\overline{\texttt{RASx}}$ applies when `mmb` is clear.
3. Appropriate timing references of $\overline{\texttt{RAS}}$ apply to RAS.
4. Timing is for memory group `msrtg`.
5. Use decoded values of register fields for calculation. Sum is the same as for a RAS cycle.

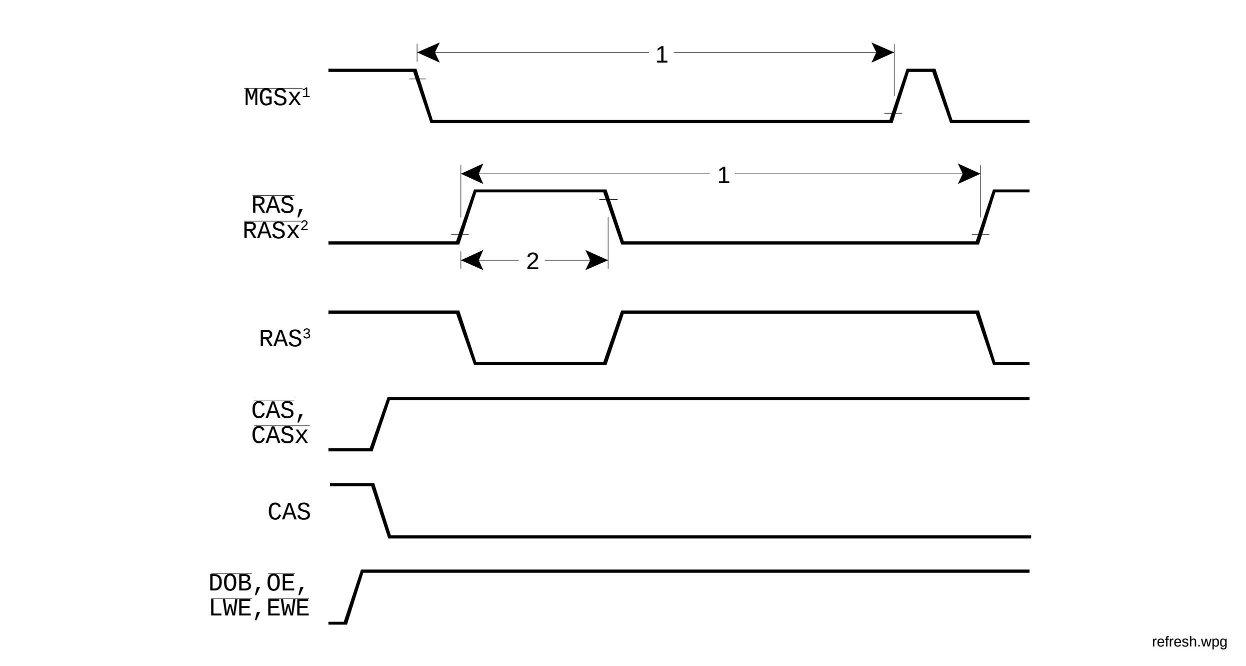


Figure 78. Refresh Timing

Table 70. VRAM Timing

No.	Characteristic	Symbol	Min	Max	Unit	Notes
1	$\overline{\texttt{RAS}}$ rise to `DSF` in `dsfvras` state		0	2	CPU-clocks	9
2	$\overline{\texttt{RAS}}$ fall to `DSF` changing to `dsfvcas` state		(`mgbtrhld` · 2) + `mgbteras`		2X-CPU-clocks	
3	`DSF` changing to `dsfvcas` state before $\overline{\texttt{CAS}}$ fall		`mgbtcas` + 1		2X-CPU-clocks	10
4	`DSF` in `dsfvcas` state after $\overline{\texttt{CAS}}$ rise		0	1	CPU-clocks	11
5	$\overline{\texttt{RAS}}$ rise to signal active		2		CPU-clocks	
6	$\overline{\texttt{RAS}}$ fall to signal inactive		time 2		2X-CPU-clocks	

Notes:

1. During an access to the VRAM memory group when `casbvras` is clear.
2. During an access to the VRAM memory group when `casbvras` is set.
3. During an access to the VRAM memory group when `oevras` is set.
4. During an access to the VRAM memory group when `wevras` is set.
5. Active during a memory read.
6. Active during a memory write.
7. `DSF` is low during non-VRAM memory group accesses.
8. All CASes move appropriately.
9. If the previous memory cycle was to the VRAM memory group then `DSF` might not go low between memory cycles.
10. Applies to RAS cycles and CAS cycles.
11. If the next memory cycle is to the VRAM memory group then `DSF` might not go low between memory cycles.

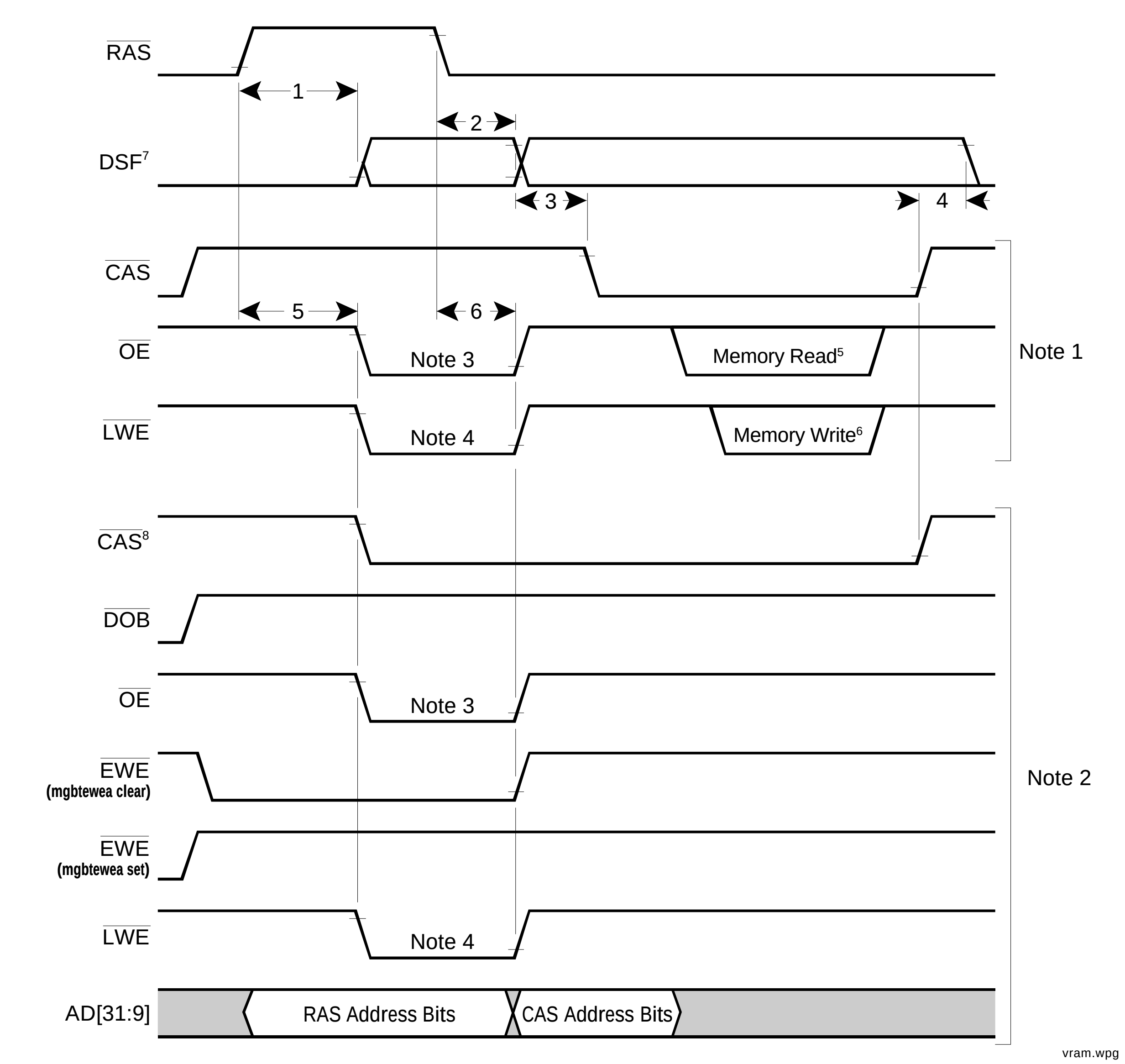
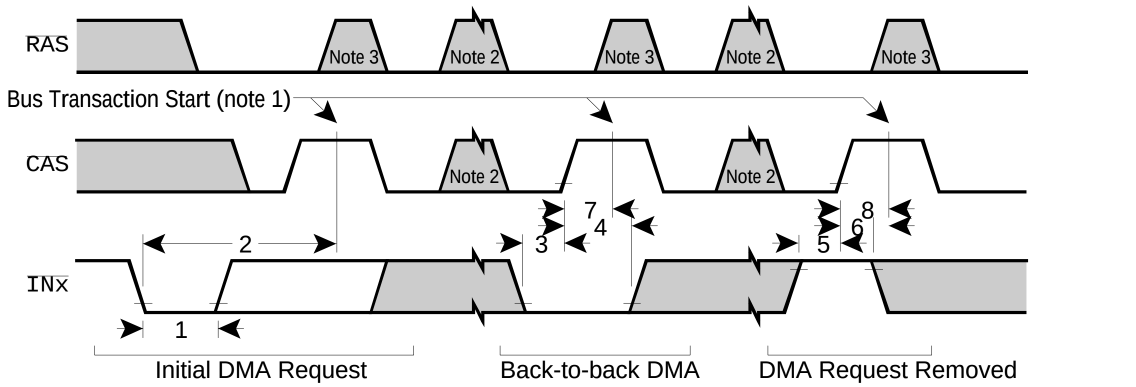


Figure 79. VRAM Timing

Table 71. DMA Request Timing

No.	Characteristic	Min	Max	Unit	Notes
1	Initial DMA request	>4		CPU-clocks	
2	Initial DMA request to first DMA I/O-channel bus cycle start	>3.25ns + 5 CPU-cycles	∞		4
3	DMA request setup before end of DMA I/O-channel bus transaction	6.75ns + 2 CPU-clocks			2
4	DMA request hold after end of DMA I/O-channel bus transaction	0		ns	2
5	DMA request high setup before end of DMA I/O-channel bus cycle	6.75ns + 2 CPU-clocks			2
6	DMA request high hold after end of DMA I/O-channel bus cycle	0		ns	2
7	End of DMA bus cycle to start of next DMA I/O-channel bus cycle		2	CPU-clocks	2,5
8	End of DMA bus cycle to start of next non-DMA I/O channel bus cycle	2		CPU-clocks	2,5

Notes:
Timings assume `pkgio` is set. When `pkgio` is clear, bus sampling timings predominate.
1. Bus transaction start can be for a RAS or CAS cycle and occurs after bus request overhead.
2. Timings are only relevant on the last bus cycle of a DMA bus transaction. Noted areas can contain 0, 3 or 4 bus cycles to complete the bus transaction. Some cycles might be RAS cycles.
3. Bus cycle could be either RAS or CAS.
4. The max condition occurs if the VPU never executes `delay` or if there are continuous DMA bus transactions from higher priority devices.
5. Value represents bus request overhead.



dmareq.wpg

Figure 80. DMA Request Timing

ADVANCE INFORMATION

ADVANCE INFORMATION

Table 72. I/O on Bus Timing

No.	Characteristic	Symbol	Min	Max	Unit	Notes
1	$\overline{\texttt{RAS}}$ rise to outputs valid			9.0	ns	
2	$\overline{\texttt{RAS}}$ fall to outputs not driven		.75		ns	
3	$\overline{\texttt{RAS}}$ fall to `AD[7:0]` bit inputs valid			2	CPU-clocks	1,4
4	`AD[7:0]` bit inputs setup before $\overline{\texttt{CAS}}$ fall		5.0		ns	1
5	`AD[7:0]` bit inputs hold after $\overline{\texttt{CAS}}$ fall		0		ns	1
6	$\overline{\texttt{CAS}}$ rise to `AD[7:0]` bit inputs valid			4	CPU-clocks	1,5
7	AD8 fault input setup to $\overline{\texttt{RAS}}$ fall		16.25		ns	2
8	AD8 fault input hold after $\overline{\texttt{RAS}}$ fall		0		ns	2
9	AD8 reset input setup before $\overline{\texttt{CAS}}$ fall		1.75		ns	3
10	AD8 reset input hold after $\overline{\texttt{CAS}}$ fall		0		ns	3

Notes:

1. `AD[7:0]` are used for inputs when `pkgio` is clear.
2. AD8 is used for memory fault requests when `pkgmflt` is clear.
3. AD8 is used for reset when $\overline{\texttt{RESET}}$ is not low at power-up.
4. If $\overline{\texttt{RAS}}$ fall to $\overline{\texttt{CAS}}$ fall is less than maximum, time 4 applies.
5. If $\overline{\texttt{CAS}}$ rise to $\overline{\texttt{CAS}}$ fall is less than maximum, time 4 applies..

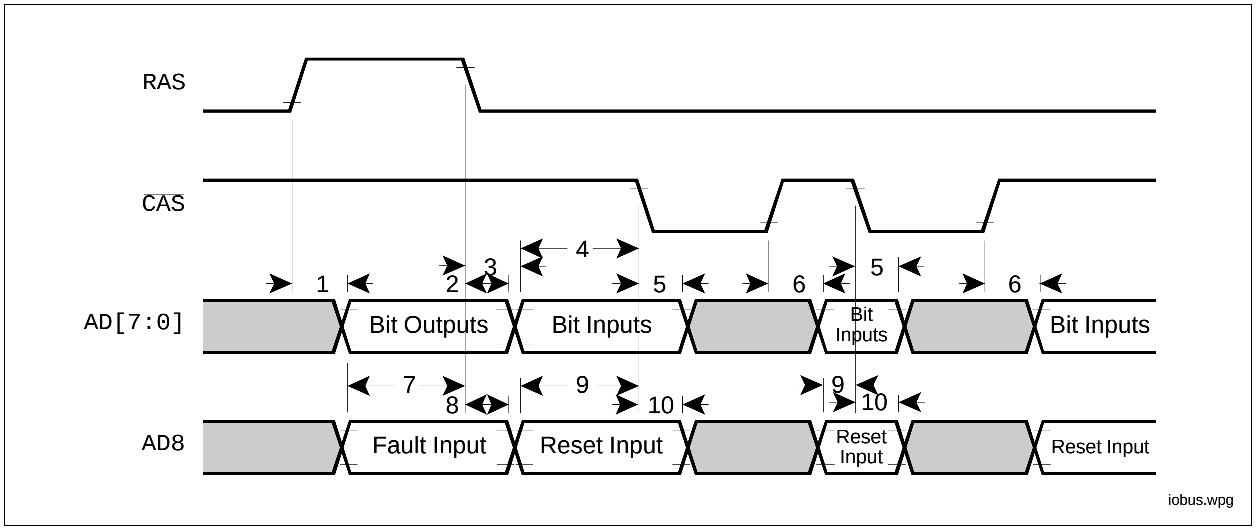


Figure 81. I/O on Bus Timing

Table 73. Bit Input Sample Timing

No.	Characteristic	Symbol	Min	Max	Unit	Notes
1	Sample clock period		4		CPU-clocks	1
2	$\overline{\texttt{INx}}$ to sample delay		.75	1.5ns + 4 CPU-clocks	ns	1
3	Low data sampled to `ioXin` delay		4		CPU-clocks	1,2,5
4	High data sampled to `ioXin` delay		4		CPU-clocks	1,2,4,5
5	$\overline{\texttt{INx}}$ to `ioXin` delay			1.5	ns	1,3

Notes:
1. $\overline{\texttt{IN}}$`[7:0]` are used for inputs when `pkgio` is set.
2. Allows data sampled in a metastable state to resolve to stated level.
3. Only during a DMA bus transaction on the corresponding I/O channel.
4. Minimum is exceeded when `ioin` is a persisting zero.
5. Except during a DMA bus transaction on the corresponding I/O channel.

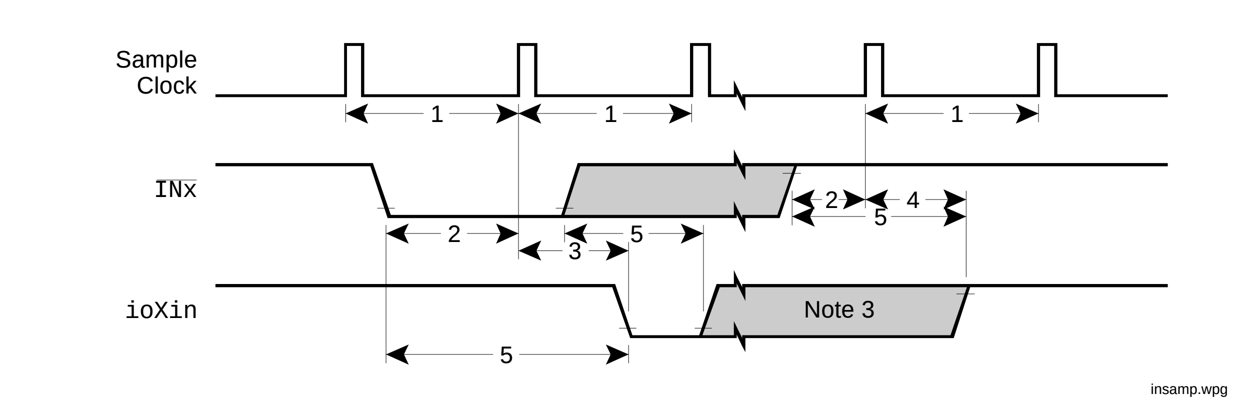


Figure 82. Bit Input Sample Timing

Table 74. Bit Input from Bus Sample Timing

No.	Characteristic	Symbol	Min	Max	Unit	Notes
1	$\overline{\text{RAS}}$ fall to first sample		2		CPU-clocks	1
2	Continued sample clock while $\overline{\text{CAS}}$ remains high		4		CPU-clocks	
3	Sample clock to $\overline{\text{CAS}}$ fall			5.0	ns	2
4	$\overline{\text{CAS}}$ rise to first sample		4		CPU-clocks	
5	$\overline{\text{CAS}}$ inactive		4		CPU-clocks	
6	$\overline{\text{CAS}}$ inactive			<4	CPU-clocks	
7	External input change to AD change			50.5	CPU-clocks	3
8	AD to sample delay			4	CPU-clocks	4
9	Low data sampled to `ioin` delay		4		CPU-clocks	5
10	High data sampled to `ioin` delay		4	note 5	CPU-clocks	5, 6

Notes:

1. If $\overline{\text{RAS}}$ fall to $\overline{\text{CAS}}$ fall is less than maximum, time 3 applies.
2. Applies only when four or more CPU-clock cycles have elapsed since the last sample.
3. Does not include external buffer delay.
4. Minimum is specified only to allow meeting specific sampling events.
5. Allows data sampled in metastable state to resolve.
6. Minimum is exceeded when `ioin` is a persisting zero.

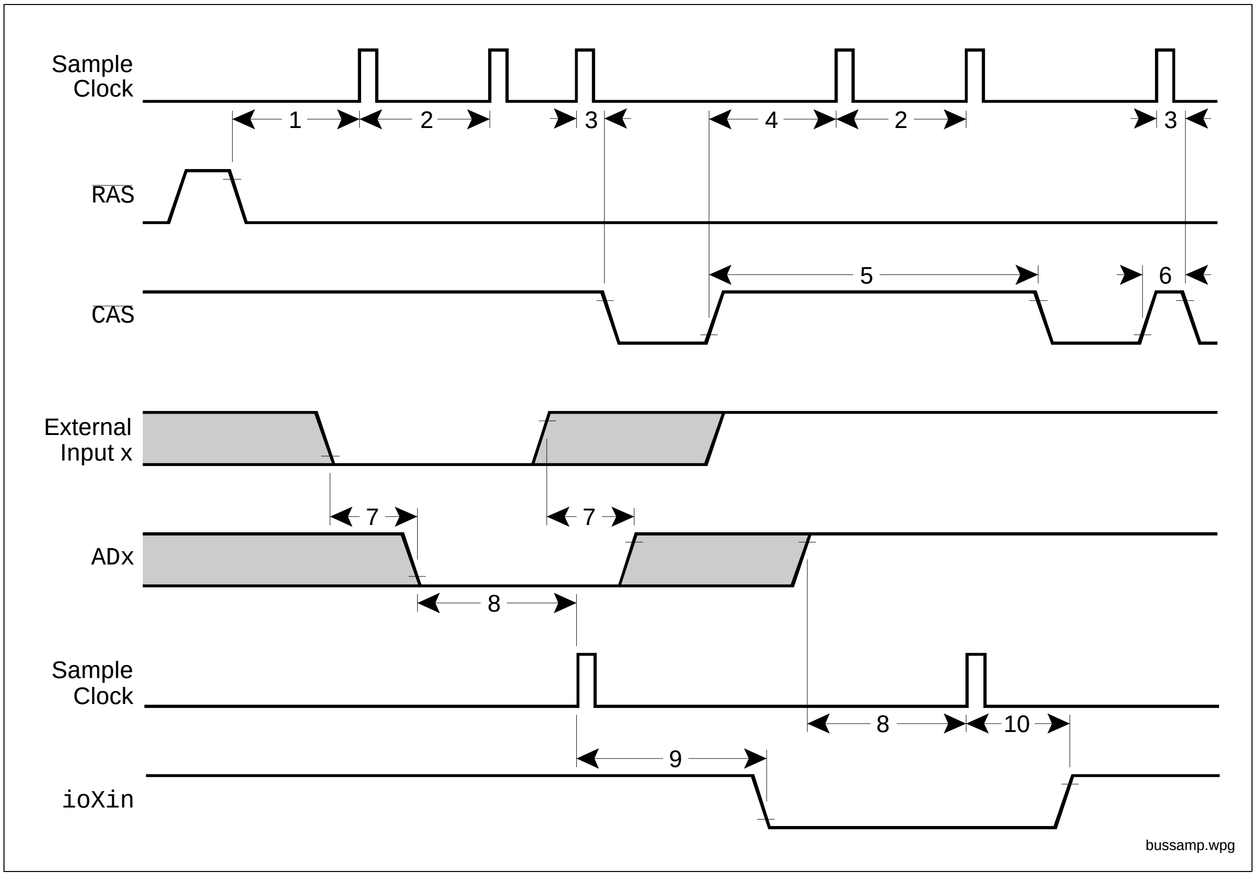


Figure 83. Bit input from Bus Sample Timing

ADVANCE INFORMATION

Mechanical Characteristics

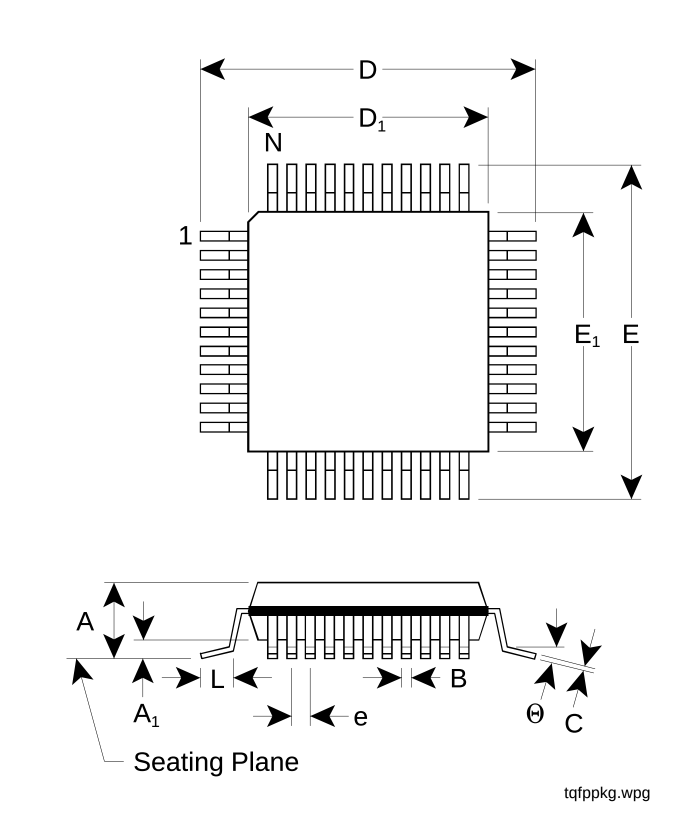


Figure 84. 100-Pin TQFP Package Dimensions

Table 75. 100-Pin TQFP Package Dimensions

Symbol	Millimeters		
	Min.	Nom.	Max.
A	—	—	1.60
A_1	.05	—	.15
B	.17	.20	.27
C	—	—	.17
D	16.00 BSC.		
D_1	14.00 BSC.		
E	16.00 BSC.		
E_1	14.00 BSC.		
L	.45	.60	.75
N	100		
e	.50 BSC.		
coplanarity	—	—	.08
θ	0°	3.5°	7.0°

Note: JEDEC SPEC MS-026

ADVANCE INFORMATION

Table 76. 100-Pin TQFP Package Thermal Characteristics

Characteristic	Symbol	Value @ Airflow LFM				Unit	Notes
		0	225	500	1000		
Thermal Resistance, Junction to Ambient	θ_{JA}	42	37	32	28	°C/W	
Thermal Resistance, Junction to Case	θ_{JC}	10				°C/W	

Notes:

Revision History

ADVANCE INFORMATION

Distributors and Sales Offices

Asia

JAPAN
RealVision
3-1-1 Shin-Yokohama
Kouhoku-Ku, Yokohama
2220033 JAPAN
Mac Sano
Tel: 81 (45)473-7331
Fax: 81 (45)473-7330
e-mail: sano@realvision.co.jp

KOREA
Acetronix
5th Fl Namhan Bldg
76-42 Hannam-Dong
Yongsan-Ku Seoul 140-210, Korea
Tel: +822-796-4561
Fax: +822-796-4563
Shane Rhee
e-mail: ace@ace-tronix.co.kr

SINGAPORE, MALAYSIA, THAILAND, PHILIPPINES, INDONESIA
Microtronics Associates PTE LTD
8 Lorong Baker Bantu, #30-01
Kolam Ayer Industrial Pakr,
Singapore 348743
Tel: 65-748-1835
Fax: 65-743-3065
Samuel Tan
e-mail: microapl@pacific.net.sg
web site: www.microtronics-associate.com

TAIWAN
Pantek Technology Corp.
11F, No. 156 Sec. 5 Nan-King E. Rd
Taipei, Taiwan R.O.C.
Tel: +886-225-2749-5909
Fax: +886-225-2749-4053
Victor Shen
e-mail: pantek@gcn.net.tw

Europe

FINLAND
Inegrated Electronics Oy Ab
Laurinmaenkuja 3 A, 00440 Helsinki
PL31 00441 Helsinki, Finland
Tel: 90-2535-4400
Fax: 90-2535-4450
Ilpo Hamunen
e-mail: ilpo@ieoy.fi
web site: www.ieoy.fi

GERMANY & AUSTRIA
Ineltek Gmbh
Haupststr. 45
D-85922 Heidenheim, Germany
Tel: 49-7321-9385-0
Fax: 49-7321-9385-95
Roland Becker
e-mail: becker@ineltek.com
web site: www.ineltek.com

Middle East

ISRAEL
Iridium Data Ltd.
1 Shwartz St. Eliave Center
P.O. Box 677
Ra'anana 43000
Tel: +972-9-74505555
Fax: +972-9-7451515
Yossi Gabbay
e-mail: iridium@netvision.net.il
web site: www.iridium.co.il

USA

Patriot Scientific Corporation
10989 Via Frontera
San Diego, CA 92127
1 (619) 674 5000 (voice)
1 (619) 674 5005 (fax)
www.ptsc.com

ADVANCE INFORMATION

Index

ADVANCE INFORMATION