# ATT2100 Microprocessor
# Hardware Specification

RELEASE 1.7.2

March 31, 1991

CONTENTS

LIST OF FIGURES

LIST OF TABLES

# 1. PRODUCT OVERVIEW

The ATT2100 is a next generation general purpose microprocessor designed for high performance and low power dissipation.

## 1.1 Product Summary

The following sections summarize the major features of this product.

### *1.1.1 ATT2100 Functional Description*

The ATT2100 microprocessor, which is functionally depicted in Figure 1-1, has a full 32-bit byte addressable architecture. The instruction set uses a simple memory to memory addressing scheme and there are no programmer visible general registers. It has a small but "sufficient" number of user level instructions and addressing modes suitable for programming in a high level language on a highly pipelined machine. With pipelining, most instructions can be executed in a single cycle. Major implementation features of the ATT2100 include:

1. A 3K-byte three-way set associative physical Prefetch Buffer Cache.
2. A 32-entry direct mapped physical/virtual Decoded Instruction Cache.
3. A 256-byte (64-word) direct Stack Cache which maintains the top of stack on chip.
4. Two 32-entry Translation Look-aside Buffers for address translation of text and data references.
5. A simple and efficient subroutine linkage mechanism.
6. Static branch prediction and the folding of branches into other instructions.
7. Data byte encoding selectable for both user and kernel modes of operation.
8. A one cycle minimum access synchronous I/O protocol and a block mode access.
9. 3.3 Volt operation.
10. Low-power stand-by mode.

The Prefetch Buffer Cache holds copies of text from external memory. The Prefetch Buffer is a three-way set associative cache, with each way being 1K bytes in size, and using physical, rather than virtual addressing.

The Decoded Instruction Cache holds one fully decoded instruction per entry. It is a direct-mapped cache with no translation performed on addresses to be looked up in the cache. Therefore, it caches using virtual addressing when virtual addressing in enabled, and physical addresses when virtual addressing is disabled.

To provide for further functional integration, a set of instruction encodings have been reserved to provide floating-point support. These reserved instructions are trapped as are all unimplemented instructions. See Section 2.11.4 for details.

A complete description of the architectural implementation is given in Section 2. A complete description of the performance of the architecture is given in Section 3.

### *1.1.2 Electrical Interface*

The electrical interface of the ATT2100 microprocessor is a CMOS design, with current leakage characteristics of CMOS. All inputs are CMOS circuits with CMOS voltage levels. All outputs are CMOS levels. The ATT2100 operates from two 20 MHz clocks, one delayed 90 degrees in phase with respect to the other as depicted in Figure 9-1. The ATT2100 requires a 3.3 ±5% volt supply. A complete description of the electrical interface is given is Section 4.

### *1.1.3 Summary of the ATT2100 Pin-Out and Protocol Features*

The ATT2100 microprocessor interface is designed to provide an easily interfaced high performance data transfer mechanism. Salient features of the interface are:

- One clock period synchronous bus cycle.
- Synchronous wait state insertion.
- Double-word/quad-word "Block Transfer" capability.
- Read-modify-write interlocked bus cycle.
- IEEE 1149.1/D5 Test Access Port Compatible.
- Six levels of maskable interrupts and a non-maskable interrupt.
- External Bus arbitration
- Byte marks for sub-word access.
- Low-power stand-by mode.

There are 93 active signal pins.

A complete description of the I/O protocol is given in Section 5.

### *1.1.4 Memory Management*

The ATT2100 microprocessor has an integrated Memory Management Unit which supports paged and unpaged segmented address translation. A complete description of the memory management unit is given in Section 6.

### *1.1.5 Reliability*

The long term reliability objective for the ATT2100 microprocessor is 500 FIT when the nominal junction temperature is at or below 85°C. If the nominal junction is at or below 55°C the long term reliability objective is 250 FIT. One FIT is defined as one device failure in 1,000,000,000 device hours. More details on reliability can be found in Section 7.

### *1.1.6 Environmental Requirements*

The environmental section provides temperature and humidity limits for the device. A means of determining junction temperature is provided. The package's thermal resistance is given as well as a means to determine power dissipation at a given frequency of operation. More details on environmental requirements can be found in Section 7.

### *1.1.7 Physical Design*

The ATT2100 microprocessor is available in both a 125 pin CPGA and a 132 pin PQFP. A complete description of packaging is given in Section 8.

### *1.1.8 Timing Specifications*

The I/O timing specification section identifies the preliminary timing of all signals. A complete description of I/O timing is given in Section 9.

### *1.1.9 Testability*

A IEEE 1149.1/D5 interface is provided which allows access to a boundary scan mechanism for board testing. The chip can be tri-stated from the rest of the system to allow safe in-circuit testing of circuit boards. Also, all on-chip caches can be individually disabled to facilitate testing. Details on testability can be found in Section 10.

### 1.2 Supporting Documentation

This document specifies all requirements to be satisfied by the ATT2100 microprocessor. It does not contain supporting documentation such as technical memoranda relating to the project, data sheets, user manuals, etc.

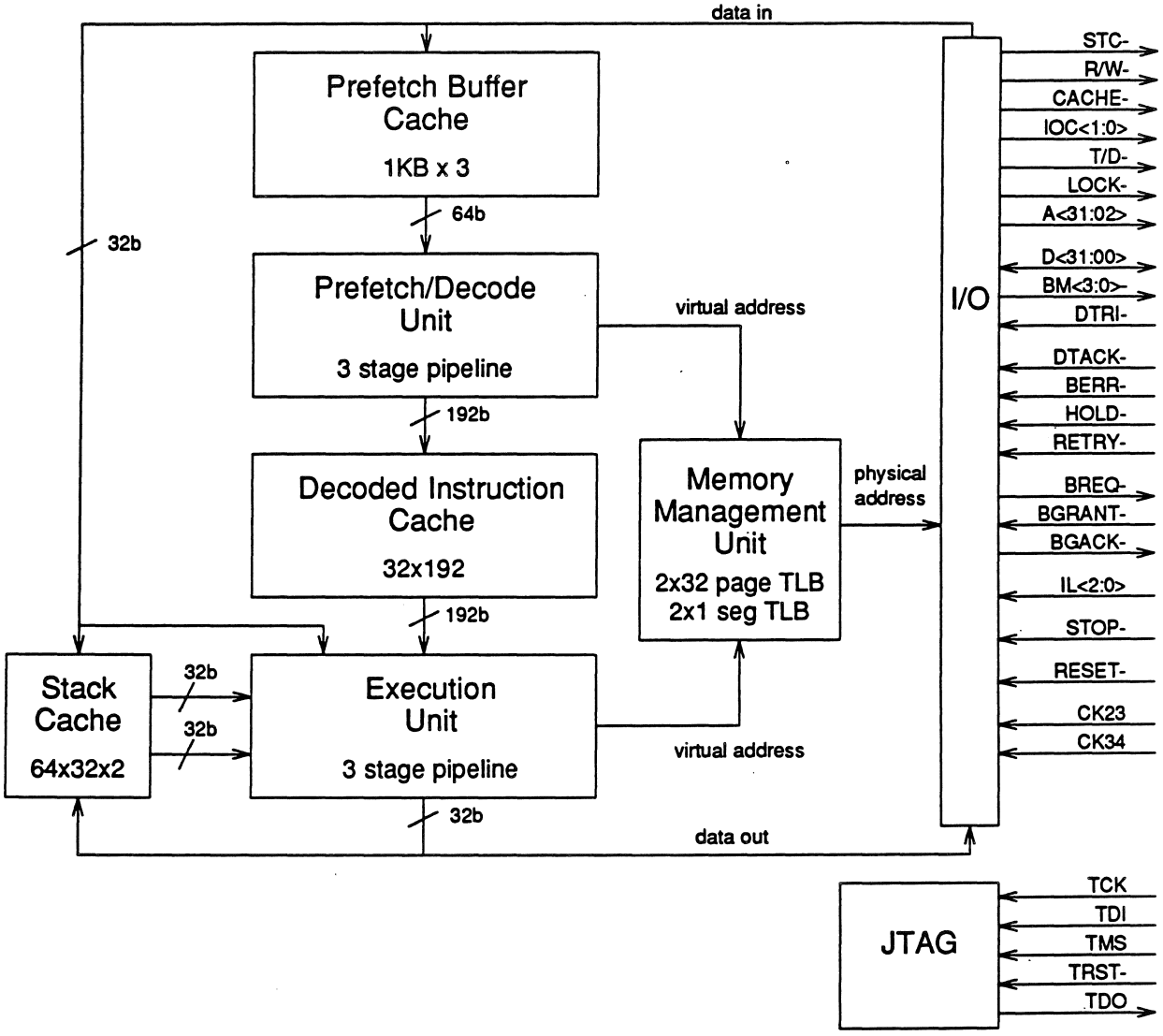



**Figure 1-1.** ATT2100 Functional Block Diagram

## 2. ATT2100 FUNCTIONAL DESCRIPTION

### 2.1 Data Types

Currently, six integer data types are supported: signed and unsigned bytes (8-bits), signed and unsigned half-words (16-bits), and signed and unsigned words (32-bits). Non-word operands are properly aligned and then expanded to 32-bits through sign extension (if signed) or clearing high order bits (if unsigned).

After alignment and expansion, the 32-bit ALU performs the requested function. Carry and overflow are determined relative to the 32-bit result. Section 2.7 gives a full description of integer arithmetic and the determination of carry and overflow.

For destinations less than 32-bits, the least significant bits of the 32-bit ALU result are selected. Changing a value by truncation constitutes neither overflow nor carry. For two-and-a-half-operand instructions, the full 32-bits of the result are placed in the Accumulator regardless of the sizes of the two operands.

To provide for further functional integration, three floating-point data types have been defined. These floating-point data types are: float (32-bits), double (64-bits) and extended (80-bits). Although hardware floating-point support is not provided in this version of the ATT2100, a series of instructions have been defined to provide for software implementation of floating-point operations. Section 2.8 will give a full description of floating-point arithmetic in a future release.

### 2.2 Addressing and Alignment Restrictions

The numbering of bits within bytes and words corresponds to that in the DEC VAX, Intel 80X86 and the Motorola 680X0. The numbering of bytes within data words is selectable for both the User and Kernal modes via the PSW UL-bit and the CONFIG KL-bit, respectively. When the PSW UL-bit and CONFIG KL-bit are set to 0, the numbering of bytes within data words corresponds to that in the IBM 370 and Motorola 680X0 in the user mode and kernal mode, respectively:

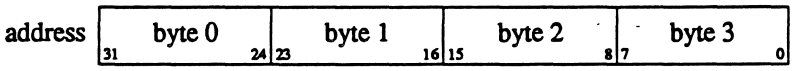



**Figure 2-1.** ATT2100 Little Endian Byte Ordering

When the PSW UL-bit and CONFIG KL-bit are set to 1, the numbering of bytes within data words corresponds to that in the VAX and Intel 80X86 in the user mode and kernal mode, respectively:

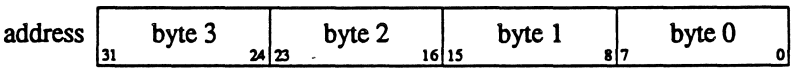



**Figure 2-2.** ATT2100 Little Endian Byte Ordering

Text is always big endian.

The ATT2100 fetches only words; bytes and half-words are accessed by extracting them from the surrounding word. Likewise, all stores are done to word-addresses, with the appropriate write strobes enabled. However, during reads, the byte-strobes indicate which bytes within the word being fetched will ultimately be extracted by the instruction.

All operand addresses should be naturally aligned for the operand type.[1] If an operand fetch or operand

store is to an address which is not properly aligned for the data type, an Alignment Exception is signaled. Instructions can only be fetched on half-word boundaries so instruction addresses should be suitably aligned, although no exception is signaled.

### 2.3 Stack Cache

The goal of the stack cache is to keep the top elements of the stack in high speed registers. The stack cache consists of a bank of 64 registers organized as a circular buffer maintained by two registers, the Maximum Stack Pointer (MSP) and the Stack Pointer (SP). Both the MSP and the SP are 28-bit registers holding quad-word addresses. The MSP contains the address above the highest address of the data which is currently kept in the stack cache registers; the SP delimits the lowest address of data in the stack cache. Therefore only a simple range-check is needed to determine if an address resides within the stack cache. If SP ≤ ADDR < MSP, it falls within the stack cache. Although the stack cache limits are maintained on quad-word boundaries, the stack cache is byte addressable and appears as normal memory. All virtual addresses generated within the ATT2100 to access data may freely reference the stack cache.

#### *2.3.1 Stack Cache Maintenance*

Six instructions maintain the stack cache. They are **CALL**, **CATCH**, **CRET**, **ENTER**, **POPN** and **RETURN**. The **CALL** instruction places the return address on the stack and branches to the target address. The **ENTER** instruction allocates space for the new procedure's stack-frame by subtracting its operand, the size of the new stack-frame, from the SP. The **POPN** instruction deallocates the current stack-frame by adding its argument to the SP. The **RETURN** instruction deallocates the current stack-frame by adding its argument to the SP, then branching to the return address on the stack. The **CATCH** instruction guarantees that the stack cache is filled at least as deep as the number of the bytes specified in its operand. The **CRET** instruction is used by the kernel to load a new SP and MSP and execute the function of **CATCH** to fill the stack cache.

**ENTER** and **CATCH** are also used to handle the cases where the stack cache circular buffer is not large enough to accommodate the entire stack. When a new procedure is entered, the **ENTER** instruction attempts to allocate a new set of registers equal to the size of the new stack-frame. If free register space exists in the circular buffer then all that needs to be done is to modify the SP. If not, then the entries nearest the MSP are flushed back to main memory. Two cases exist:

- If the new stack-frame size is less than 256 bytes, then only the stack-frame size minus the number of free entries must be flushed.
- If the new frame size greater than or equal to 256 bytes, then all valid stack cache entries are flushed and only part of the new stack-frame nearest the SP is kept in the stack cache.

After successful completion of the **ENTER**, the PSW E-bit is set. The PSW E-bit is cleared by the **CLRE** instruction.

After a procedure returns to the caller, it is not known how many of the stack cache entries were flushed since the call, so some entries may need to be restored from off-chip memory. The argument of the **CATCH** instruction specifies the number of stack cache entries that must be valid before execution can continue. The **CATCH** argument is used as a stack offset and a virtual address is generated. If this calculated address resides within the stack cache, (SP ≤ ADDR < MSP) execution continues. However if it lies outside the address range of valid stack cache entries, quad-words pointed to by the MSP are restored from off-chip memory to the stack cache and the MSP incremented until either the **CATCH** instruction is

1. Byte-by-byte, half-words-on-half-words, words-on-words.

satisfied or the stack cache is full. The **CATCH** instruction behaves much like an assertion, since usually no entries need to be restored and the **CATCH** takes only one clock cycle.

### *2.3.2 Integer Accumulator*

The Integer Accumulator is not an actual hardware register. It is the word in memory above the word addressed by the Current Stack Pointer (CSP). The CSP is either the Stack Pointer (SP) or the Interrupt Stack Pointer (ISP), as determined by the Program Status Word (PSW) S-bit. Many two operand instructions use the Integer Accumulator as an implicit destination address. The Integer Accumulator normally resides on chip in the stack cache, but may be off chip if the SP == MSP or CSP == ISP.

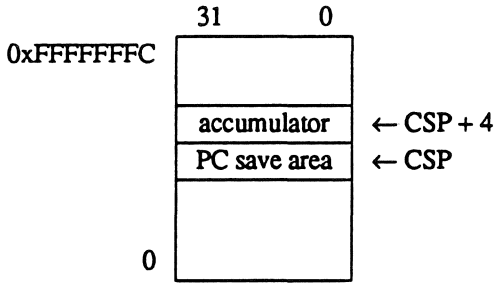



**Figure 2-3.** Integer Accumulator

### *2.3.3 Floating-point Accumulator*

The Floating-point Accumulator is not an actual hardware register. It is the quad-word in memory based at the word addressed by the CSP. The CSP is either the SP or the ISP, as determined by the PSW S-bit. There are no hardware instructions which currently use the Floating-point Accumulator.

The value at R0 is not affected by manipulations of the Floating-point Accumulator due to the alignment of the FPA. R0 is the location pointed to by the CSP and is where the return PC is saved by **CALL** instructions. Section 2.8 will describe the format of the Floating-point Accumulator in a future release.

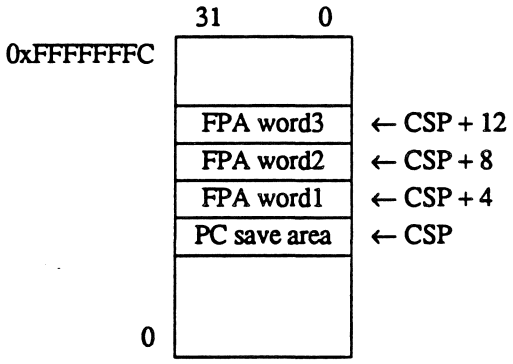



**Figure 2-4.** Floating-point Accumulator

### *2.3.4 Stack Precautions*

As discussed in Section 2.3, the stack cache is conceptually a cache for memory. If an address is generated in any processing stage, for example in doing indirect address calculations, the stack cache is referenced if that address is greater-than the SP and less-than the MSP. However, for implication simplicity, this conceptual model is violated for some interrupt stack references during event processing and for some references to R0 and R4 when executing with CSP = ISP. There are no problems with memory accesses as long as the the user stack, based at the SP, and the interrupt stack, based at the ISP, do not overlap.

For similar reasons, the following addresses must not lie between the SP and MSP:

- The vector table, defined by the Vector Base (VB) described in Section 2.4.14,
- the address translation tables used by the MMU, or
- any text.

## 2.4 Control Registers

### *2.4.1 Configuration Register (CONFIG)*

The Configuration Register (CONFIG), which is set to 0x0 upon reset, contains the following information:

- Reserved (R): Bits 0 through 15 are reserved. They return zeros when read and should be written with zeros on CONFIG writes.
- Kernal Little Endian (KL): A 0 selects data as big endian in kernal mode; a 1 selects data as little endian in kernal mode. Note that text is always big endian.
- PC Extension (PX): A 0 selects zero extension of 16-bit absolute addresses; a 1 selects the extension of 16-bit absolute addresses where bits 29 through 31 are copied from bits 29 through 31 of the PC and bits 16 through 28 set to 0.
- SC Enable (SE): A 0 disables the Stack Cache (SC) from hitting; a 1 enables the SC. The SC is *neither* flushed or altered when this bit is modified.
- IC Enable (IE): A 0 disables the Instruction Cache (IC) from hitting; a 1 enables the IC. The IC is *neither* flushed or altered when this bit is modified.
- PF Enable (PE): a 0 disables the Prefetch Buffer (PFB) from hitting; a 1 enables the PFB. The PFB is *neither* flushed or altered when this bit is modified.
- Prefetch Mode (PM): This bit controls prefetching of instructions. When 0, prefetching off chip is not performed; predecoding from the PFB into the IC is performed. When 1, aggressive prefetching is performed. See Section 2.11 for a full description of the prefetching strategy.
- TIMER1 Configuration (T1): A three-bit field which configures TIMER1:
  - — Bit 0. When 0, TIMER1 counts clock cycles. When 1, TIMER1 counts completed instructions (folded branches do not count).
  - — Bit 1. When 0, TIMER1 is on all the time (with reference to bit 0). When 1, the timer only increments when the PSW X-bit is 0.
  - — Bit 2. When 0, TIMER1 does not generate an interrupt. When 1, TIMER1 generates an interrupt using a TIMER1 vector when an overflow does occur (goes from 0xFFFFFFFF to 0x0). This interrupt is a level one interrupt. If an external level one interrupt occurs at the same time as a TIMER1 interrupt, the external interrupt is serviced first.
- TIMER2 Configuration (T2): A seven-bit field which configures TIMER2:
  - — Bit 0 through 4. A five-bit encoded field which selects the internal event which increments TIMER2.
    - ☞ 0x0 - Count clock cycles.
    - ☞ 0x1 - Count completed instructions (folded branches do not count).
    - ☞ 0x1F - Do not increment the timer; a low power feature.

— Bit 5. When 0, TIMER2 is on all the time (with reference to bits 0 through 4); when 1, the timer only increments when the PSW X-bit is 0.

— Bit 6. When 0, TIMER2 does not generate an interrupt; when 1, TIMER2 generates an interrupt using a TIMER2 vector when an overflow does occur (goes from 0xFFFFFFFF to 0x0). This interrupt is a level one interrupt. If either an external level one interrupt or a TIMER1 interrupt occur at the same time as a TIMER2 interrupt, the other interrupts dominate and are serviced first.

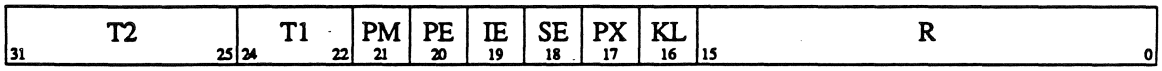



**Figure 2-5.** Configuration Register Format

*2.4.1.1 Assembler Language Syntax*

%CONFIG

**Caution:** Special precautions must be taken when modifying the Configuration Register. The number of **NOP**s which must come after the register write varies according to which bits are being modified and the number of wait states being used by I/O transactions. The prescribed means of modifying CONFIG is to follow the CONFIG write by either a **CRET** or **KRET**.

*2.4.2 Fault Register (FAULT)*

The Fault Register (FAULT) reports the 32b operand aligned virtual address for the processing of exception IDs 0x8 and 0x9. Section 2.13.3 gives more detail on exceptions.

*2.4.2.1 Assembler Language Syntax*

%FAULT

*2.4.3 Floating-point Status Word Register (FPSW)*

The Floating-point Status Word Register (FPSW) is not implemented in this version of the ATT2100, but is defined for software emulation of the unimplemented floating-point instructions.

- Reserved (R): The bits 0 through 2 are reserved. They return zeros when read and should be written with zeros on FPSW writes.
- Remainder Quotient (RQ): The signed low 4-bits from the last **FREM**, in 2's complement.
- Excluded exceptions (XE): A five-bit-field which masks the selected exceptions from LSB to MSB as: invalid, underflow, overflow, division by zero, inexact.
- Exceptions last operation (XL): A five-bit-field which indicates exceptions from the last performed operation defined from LSB to MSB as: invalid, underflow, overflow, division by zero, inexact.
- Exceptions halt enables (XH): A five-bit-field which enables exception halts defined from LSB to MSB as: invalid, underflow, overflow, division by zero, inexact.
- Accumulated exceptions (XA): A five-bit-field which indicates accumulated exceptions defined from LSB to MSB as: invalid, underflow, overflow, division by zero, inexact.
- Rounding precision (RP): A two-bit-field which indicated the rounding precision used as: 00 - to extended, 01 - to double, 10 - to single, 11 - reserved.
- Rounding direction (RD): A two-bit-field which indicates the rounding direction used as: 00 - to nearest, 01 - toward $+\infty$, 10 - toward $-\infty$, 11 - toward 0.

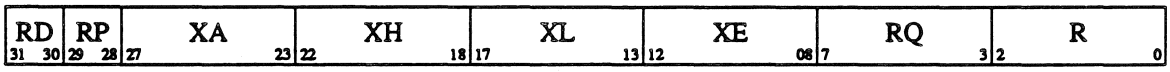



**Figure 2-6.** Floating-point Status Word Register Format

*2.4.3.1 Assembler Language Syntax*

%FPSW

*2.4.4 Identification Register (ID)*

The Identification Register (ID) serves as the JTAG Device Identification Register and is readable by serial shifting through the Test Access Port (TAP) and through normal register access. This register is only readable. No operation is performed if the kernel attempts to write the ID register.

- Manufacturer Code (MC): A 12-bit-field which identifies the manufacturer of the device as AT&T Microelectronics. The encoding is 0x3B from MSB to LSB with LSB closest to the Test Data Output (TDO) pin.
- Part Code (PC): A 16-bit-field which identifies the device. The encoding for the ATT2100 is 0x0.
- Version Code (VC): A 4-bit-field which identifies the version of the device. The encoding for Mask 1 silicon is 0x0 and for Mask 2 silicon is 0x1 with the MSB closest to the Test Data Input (TDI) pin.

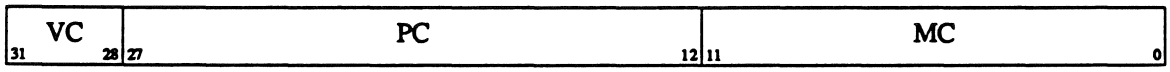



**Figure 2-7.** Identification Register Format

*2.4.4.1 Assembler Language Syntax*

%ID

*2.4.5 Interrupt Stack Pointer (ISP)*

The Interrupt Stack Pointer (ISP) is used to generate addresses (*i.e.*, as the base address in offset modes, to locate the Accumulator, and as the pointer manipulated by the instructions **CALL**, **RETURN** and **ENTER**) whenever the PSW S-bit is 0. The ISP is not associated with the stack cache as detailed in Section 2.3. The instructions **CRET**, **KCALL** and **KRET**, and operating system sequences interrupts and exceptions, use the ISP to maintain a stack of event blocks.

The ISP must be valid at all times. A fault on any ISP based address results in the ATT2100 resetting. See Section 2.13.1 for details.

Address translation is performed if the MMU is enabled by setting the PSW VP-bit to 1.

The ISP is quad-word-aligned. The low-order four bits return zero when read.

*2.4.5.1 Assembler Language Syntax*

%ISP

*2.4.6 Maximum Stack Pointer (MSP)*

The Maximum Stack Pointer (MSP), in conjunction with the SP, is associated with the on-chip stack cache as detailed in Section 2.3. Any address which is greater than or equal to the SP and less than the MSP hits in the stack cache.

SC hit *when* SP ≤ address < MSP

On a memory access which hits in the stack cache, data is fetched or stored in the cache and not in external memory. The MSP must be greater than or equal to the SP and less than or equal to SP + SCSIZE,[2] or the result of stack cache accesses are dependent upon context and therefore are unpredicatable.

Whenever the SP is the direct destination of an instruction, through a CPU-prefixed instruction with the SP as the destination, the MSP is updated with the same value. This defines an empty stack cache (SP = MSP). The MSP is manipulated implicitly by the **CATCH, CRET, ENTER** and **RETURN** instructions.

Address translation is performed if the MMU is enabled by setting the PSW VP-bit to 1.

The MSP is quad-word-aligned. The low-order four bits return zero when read.

*2.4.6.1 Assembler Language Syntax*

%MSP

*2.4.7 Program Counter (PC)*

The Program Counter (PC) addresses the instruction which is currently being executed. Instructions are aligned on parcel (half-word) boundaries. Since parcels are composed of two-bytes, the PC is always a multiple of two and the low-order bit is always 0. The PC cannot be directly manipulated by a general instruction. It can only be read or modified by control-flow instructions **CALL, CRET, JMP, KCALL, KRET,** and **RETURN** and read by the move instruction **LDRAA** .

*2.4.8 Program Status Word (PSW)*

The Program Status Word (PSW), which is set to 0x0 upon reset, contains the following information:

- Reserved (R): The bits 0 through 3 are reserved. They return zeros when read and must be written with zeros on PSW writes, to provide for future expansion compatibility.
- Flag (F): Set/cleared by a **CMP, FCMP, TADD, TESTC, TESTV** and **TSUB** instructions. The F-bit is **not** cleared when the PSW is read.
- Carry (C): When 1, indicates that an operation generated an unsigned overflow; when 0, indicates that an operation did not generate an unsigned overflow. See Section 2.7 for detail.
- oVerflow (V): When 1, indicates that an operation generated a signed overflow; when 0, indicates that an operation did not generate a signed overflow. See Section 2.7 for detail.
- Trace instruction (TI): Controls instruction tracing. When 1, the ATT2100 allows the next instruction, N, to execute normally. The instruction following instruction N, referred to as N+1, is not permitted into the Execution Unit and a "trace instruction" is generated on the fly. This "trace instruction" blocks the pipeline and forces the ATT2100 to take a trace exception using the PC of the N+1 instruction as the exception PC. As branch folding is performed prior to the trace identifier, folded branches are not explicitly traceable. See Section 2.12 for additional detail on tracing. If both TI and TB are set to 1, the function is that of TI.
- Trace basic block (TB): Controls basic block tracing. When 1, the ATT2100 executes instructions until a **CALL, RETURN,** or any jump (folded or not) instruction, referred to as the N instruction, executes. The instruction following instruction N, referred to as N+1, is not permitted into the Execution Unit and a "trace instruction" is generated on the fly. This "trace instruction" blocks the pipeline and forces the ATT2100 to take a trace exception using the PC of the N+1 instruction

2. SCSIZE = stack cache size currently 256 bytes

as the exception PC. As branch folding is performed prior to the trace identifier, folded branches are not explicitly traceable. See Section 2.12 for additional detail on tracing. If both TI and TB are set to 1, the function is that of TI.

- Current Stack Pointer (S): When 1 the SP is used as the Current Stack Pointer (CSP) for address generation; when 0 the ISP is used as the CSP for address generation.
- Execution level (X): When 1 execution at user level is performed, when 0 execution at the kernel (privileged) level is performed.
- Enter guard (E): Set on an uneventful **ENTER**. The E-bit is **not** cleared when the PSW is read.
- Interrupt Priority Level (IPL): Interrupts are accepted when the requesting device level (IL<2:0>) < IPL or if IL<2:0>=0. IPL of 7 enables all interrupts.
- User Little Endian (UL): A 0 selects data as big endian in user mode; a 1 selects data as little endian in user mode. Note that text is always big endian.
- Virtual/Physical (VP): Bit 16, enables virtual-addressing (memory management enabled) when 1, a 0 enables physical-addressing (memory management disabled). When the VP-bit is 0, indicating physical-addressing, the CACHE- pin is de-asserted.
- Unassigned (UA): Bits 17 through 31 are not assigned any CPU function, and may be read at either execution level or written by the kernel.

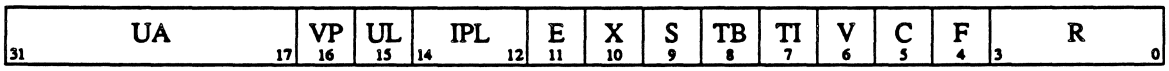



**Figure 2-8.** Program Status Word Format

The exception and interrupt sequences only alter the lower 16 bits of the PSW. To remain restartable the Carry and oVerflow bits are not cleared on reading the PSW until the instruction completes. **Reads of the PSW are not interlocked against flag setting.** If an instruction sets the Flag, the Carry or the oVerflow bits, there must be at least two intervening instructions before the PSW can be read.

**Cautions:**

- Special precautions must be taken when explicitly modifying the VP bit in the PSW. If the VP-bit is explicitly modified, the section of code executing must be mapped physical address = virtual address. The safest means of manipulating the PSW VP-bit is through either **CRET** or **KRET** instructions.
- If the PSW S-bit is modified by a direct write to the PSW, thereby changing the CSP, it is necessary to update SHAD to the value of the new SP. This update is handled automatically by **CRET**, **KCALL** and **KRET**.

  If the S-bit is set to 1, and it was previously 0, the instruction modifying the PSW should be followed by the instruction

  **MOV** %SP,%SHAD

  If the S-bit is set to 0 when it was previously 1, the next instruction should be

  **MOV** %ISP,%SHAD

Due to interrupts and exceptions, it is recommended that the S-bit not be modified by a direct write to the PSW as the above operations can not be guaranteed to be atomic.

*2.4.8.1 Assembler Language Syntax*

%PSW

*2.4.9 Segment Table Base (STB)*

The Segment Table Base (STB) contains a pointer to the start of the Segment Table used in address translation when virtual addressing is turned on by the PSW VP-bit. The base of the Segment Table is always page-size-aligned, where the size of a page in the ATT2100 is 4,096 bytes. The STB is only used during miss processing, which is used in turn to fill entries in the on-chip Translation Look-aside Buffer (TLB) or Segment Registers. The translation process is described in Section 6.

When the STB is written, the TLB's and Segment Registers of the MMU are flushed, invalidating all entries. Neither the physically addressed PFB, the virtually addressed IC or the virtually addressed SC are flushed. Cache coherency is the responsibility of the user.

Bit 11 of the STB is a cacheable bit: it is copied to the cacheable pin whenever a Segment Table access is made during miss processing, indicating whether Segment Table Entries should be cached.

The format of the STB is described in Figure 2-8. The symbol '$' is the cacheable bit. The field marked '0' always returns 0 when read.

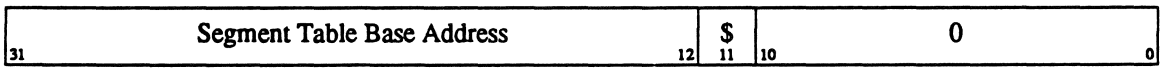



**Figure 2-9.** Segment Table Base Format

*2.4.9.1 Assembler Language Syntax*

%STB

*2.4.10 Shadow (SHAD)*

The Shadow Register (SHAD) is a copy of the CSP. It is maintained by the ATT2100's internal sequences to facilitate restarting of instructions. In the course of **CRET, ENTER, KCALL, KRET** and **RETURN** instructions, or any time the CSP is modified, SHAD is automaticly updated to be consistent with the CSP.

The SHAD is quad-word-aligned. The low-order four bits return zero when read.

**Caution:** As described in Section 2.4.8, if the PSW S-bit is modified by a direct write to the PSW, thereby changing the CSP, it is necessary to update the SHAD to the value of the new SP. This update is handled automatically by **KCALL** and **KRET.**

*2.4.10.1 Assembler Language Syntax*

%SHAD

*2.4.11 Stack Pointer (SP)*

The Stack Pointer (SP) usually addresses the top of stack. The stack grows downwards – towards memory location zero. The SP is used to generate addresses (*i.e.*, as the base address in offset modes, to locate the Accumulator, and as the pointer manipulated by the instructions **CALL, ENTER** and **RETURN**) whenever the PSW CSP-bit is 1.

Address translation is performed if the MMU is enabled by setting the PSW VP-bit to 1.

The SP is quad-word-aligned. The low-order four-bits return zero when read.

*2.4.11.1 Assembler Language Syntax*

%SP

*2.4.12 Timer One (TIMER1)*

Timer One is a 32-bit internal register which can be configured by the three-bit T1 field of CONFIG to count various events. See Section 2.4.1 for a description of the events and how selection is performed.

*2.4.12.1 Assembler Language Syntax*

%TIMER1

*2.4.13 Timer Two (TIMER2)*

Timer Two is a 32-bit internal register which can be configured by the seven-bit T2 field of CONFIG to count various events. See Section 2.4.1 for a description of the events and how selection is performed.

*2.4.13.1 Assembler Language Syntax*

%TIMER2

*2.4.14 Vector Base (VB)*

The Vector Base (VB) is used as the base of a table which contains transfer addresses used by **KCALL**, interrupts, and exceptions. Address translation is performed if the MMU is enabled by setting the PSW VP-bit to 1. The Vector Table, described in Figure 2-9, should always be available. If an access to the Vector Table entry is faulted, the ATT2100 resets. See Section 2.12.1 for details on the reset sequence.

The exception PC handler should be present in memory as a memory fault would cause an infinite loop until the interrupt-stack is exhausted and the ATT2100 resets. Additionally, the niladic trap and unimplemented instruction handlers must be in the user memory space as these handlers can be accessed while in user mode.

The VB is quad-word-aligned. The low-order four bits return zero when read.

| | 31 0 |
|---|---|
| VB + 52 → | FP exception |
| VB + 48 → | timer 2 interrupt |
| VB + 44 → | timer 1 interrupt |
| VB + 40 → | interrupt 6 |
| VB + 36 → | interrupt 5 |
| VB + 32 → | interrupt 4 |
| VB + 28 → | interrupt 3 |
| VB + 24 → | interrupt 2 |
| VB + 20 → | interrupt 1 |
| VB + 16 → | non-maskable interrupt |
| VB + 12 → | unimplemented instruction |
| VB + 8 → | niladic traps |
| VB + 4 → | exception PC |
| VB → | KCALL PC |

**Figure 2-10.** Vector Table

*2.4.14.1 Assembler Language Syntax*

%VB

**2.5 Instruction Format**

Instructions are composed of parcels which are two-bytes long. Instructions are encoded in one-, three- and five-parcel lengths. A canonical instruction is encoded in five-parcels, which allows for the encoding of two complete 32-bit addresses in each instruction. In general, the one- and three-parcel instructions are more compact encodings of five-parcel instructions. Instructions may have at most two operands, for which, in general, any addressing mode may be used. For the dyadic instructions, one source doubles as destination, or the Accumulator is selected to serve as an implicit destination. The instruction formats are:

- One-Parcel Formats
  (for zero-, one-, and two-operand instructions)
- Three-Parcel Formats
  (for one- and two-operand instructions)
- Five-Parcel Format
  (for two-operand instructions)

*2.5.1 One-Parcel Formats*

Many of the most common zero-, one-, and two-operand instruction types may be encoded in one-parcel:

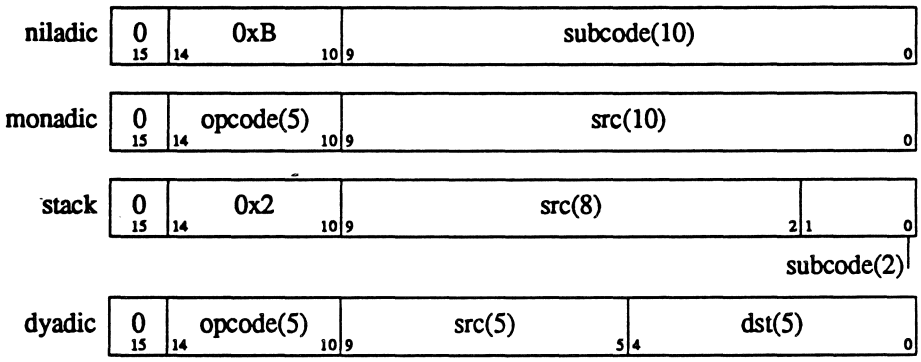



**Figure 2-11.** One-Parcel Instruction Formats

A zero in the most significant bit distinguishes all one-parcel instruction formats. The *subcode* field distinguishes among the different niladic and stack instructions. Five-bit immediate fields are sign-extended, while five-bit stack offset fields are zero-extended. All 10-bit fields are zero-extended except for **CALL** and **JMP** which are sign-extended. The 8-bit fields are zero-extended, except for **ENTER**, which is one-filled. Tables 11-1, 11-2 and 11-3 show how to decode each one-parcel instruction.

Note that operand alignment restrictions allow some address offsets to be scaled, thus extending the effective addressing range. The scaling of certain immediate constants is made possible by the specific operand value restrictions of the corresponding instructions. Five-bit offset values are multiplied by four before they are added to the SP. The 10-bit PC-relative offsets in **JMP** and **CALL** instructions are multiplied by 2 before they are used, the other 10-bit values are multiplied by 4 before they are used.

*2.5.2 Three-Parcel Formats*

Three-parcel instructions are distinguished by a "10" in the two most significant bits. The *subcode* field distinguishes among the different monadic instructions. The notation "operand-lo(16)" refers to the low-

order 16-bits and "operand-hi(16)" refers to the high-order 16-bits. A similar convention applies to the source and destination operands of the dyadic instructions.

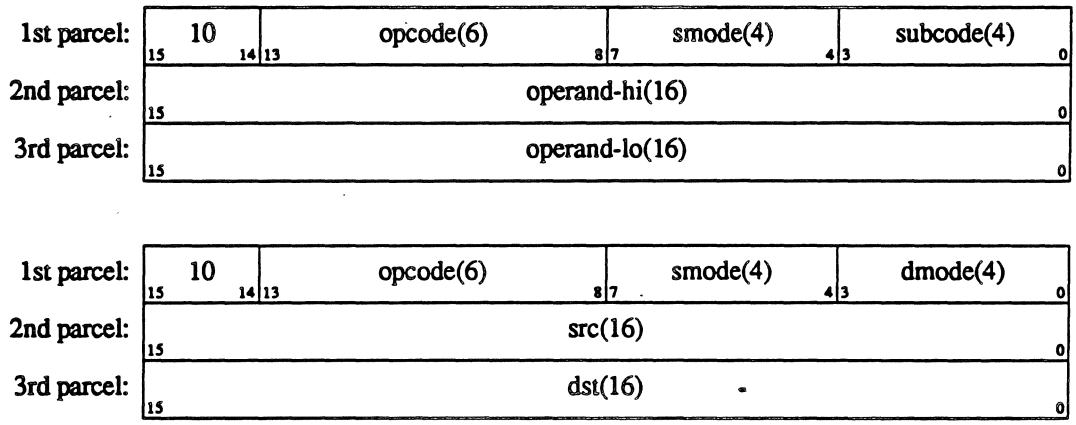



**Figure 2-12.** Three-Parcel Instruction Formats

The 16-bit source and destination fields are sign-extended to 32-bits when they are used in immediate or offset modes. When the 16-bit source and destination fields are used as absolute addresses extension of the upper 16 bits depends upon the setting of the PX bit in CONFIG. If PX is set to 1, bits 16 through 28 are replaced with 0 and bits 29 through 31 (the high-order three bits) are copied from bits 29 through 31 of the PC. If PX is set to 0, the upper 16 bits are set to zero. Tables 11-4 and 11-5 shows how to decode each opcode. The source and destination addressing mode fields are encoded in the same way for both three-parcel and five-parcel instructions. (see Tables 11-7, 11-8, 11-9 and 11-10.)

### *2.5.3 Five-Parcel Format*

Five-parcel instructions are distinguished by a "11" in the two most significant bits. Five-parcel instructions are encoded similarly to three-parcel instructions. See Table 11-6 for instruction encodings.

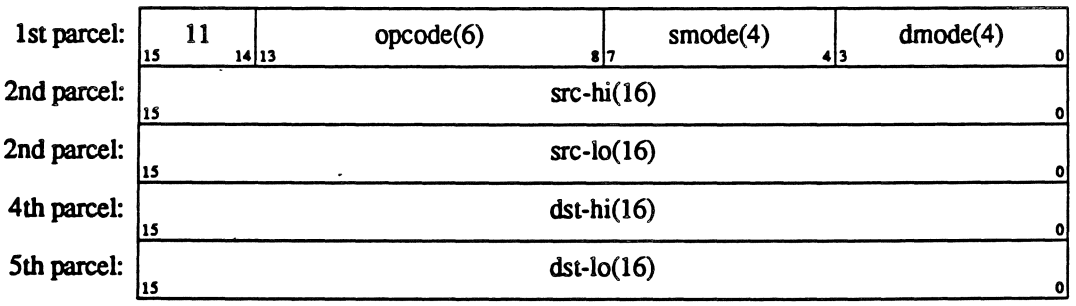



**Figure 2-13.** Five-Parcel Instruction Format

## 2.6 Addressing Modes

There are seven addressing modes:

1. Immediate
2. Absolute

3. Stack Offset
4. Stack Offset Indirect
5. Absolute Indirect
6. Program Counter Relative
7. Register

The ALU operations generally permit any of the first four of these addressing modes to be used with either operand. The valid addressing modes for each instruction are indicated in the detailed instruction descriptions. Any mode which is not explicitly mentioned for a given instruction **should not** be used. This section briefly describes each mode.

The suffixes indicate the size of data operands, a missing suffix implies word operands.

:B signed byte
:UB unsigned byte
:H signed half-word
:UH unsigned half-word
:W word
:F single precision float
:D double precision float
:E extended precision float

### *2.6.1 Immediate*

In the Immediate addressing mode, the operand value is stored in the instruction. Values up to 32-bits in length are permitted. Shorter values are appropriately sign or zero extended before use. An "illegal instruction" exception is executed if any of the uses identified in Section 2.12.3 related to this address mode occur.

#### *2.6.1.1 Assembler Language Syntax*

$data

### *2.6.2 Absolute*

In the Absolute addressing mode, the address of the operand is stored in the instruction.

#### *2.6.2.1 Assembler Language Syntax*

*$addr:B
*$addr:UB
*$addr:H
*$addr:UH
*$addr:W
*$addr:F
*$addr:D
*$addr:E

### *2.6.3 Stack Offset*

In the Stack Offset addressing mode, a signed, two's complement offset stored in the instruction (except for **CATCH** and **ENTER**, see the instruction descriptions for details) is added to the CSP value to obtain the operand address.[3]

*2.6.3.1 Assembler Language Syntax*

R*offset*:B
R*offset*:UB
R*offset*:H
R*offset*:UH
R*offset*:W
R*offset*:F
R*offset*:D
R*offset*:E

*2.6.4 Stack Offset Indirect*

In the Stack Offset Indirect addressing mode, an *offset* is added to the CSP value to obtain the address of the address of the operand. The *offset* must be word aligned[4].

*2.6.4.1 Assembler Language Syntax*

*R*offset*:B
*R*offset*:UB
*R*offset*:H
*R*offset*:UH
*R*offset*:W
*R*offset*:F
*R*offset*:D
*R*offset*:E

*2.6.5 Absolute Indirect*

In the Absolute Indirect addressing mode, the address of the address of the operand is stored in the instruction. This mode is only used for the **JMP** (conditional jump instructions excluded), **CALL**, and **LDRAA** instructions, so that the operand value should be an instruction address which must be parcel aligned.

*2.6.5.1 Assembler Language Syntax*

**$addr

*2.6.6 Program Counter Relative*

In the Program Counter Relative addressing mode, a signed, two's complement offset stored in the instruction is added to the address of the instruction to obtain the operand value. This mode is only used for the **JMP**, **CALL** and **LDRAA** instructions.

*2.6.6.1 Assembler Language Syntax*

label

*2.6.7 Register*

A CPU instruction is never directly executed, but serves to modify the next instruction's addressing modes. This "modified" instruction must use the addressing modes given in Table 11-10.

---

3. For negative offsets, off chip stack accesses are performed and cache coherency is not maintained.
4. An Alignment Fault, 0x4, is executed if the offset is not word aligned.

Mode 0x7 allows access to the internal registers for use as data. The register number is specified in the data portion of the operand. Only bits 0 through 3 are considered for determining the register number[5]. At most one register may be read per instruction. The register encoding presently supported is given in Table 11-11.

If register 0x0 or 0xD through 0xF is specified, an unimplemented register exception sequence, exception ID 0x6, is performed. See Section 2.13.3 for details.

If there is a write of registers 0x1 through 0xC in user mode, a privilege violation exception sequence, exception ID 0x5, is performed.

Register 0x10 is defined as the FPSW allowing emulation via the unimplemented register exception.

*2.6.7.1 Assembler Language Syntax*

%REGISTER

**2.7 Integer Arithmetic**

*2.7.1 The Language of Integer Arithmetic*

The concepts of integer arithmetic as implemented on various processors, while quite straightforward, are usually complicated by sloppy language which obscures the several related but distinct topics. For example, most processors support a so-called "carry bit" to indicate, say, borrow during subtraction, but in fact the setting of this carry bit is typically different from the hardware carry bit associated with the adder logic that supports the subtraction. In this section we carefully distinguish:

a. Signed integer arithmetic as supported by the ATT2100,

b. unsigned integer arithmetic as supported by the ATT2100, and

c. hardware issues which lead to nuances of (a) and (b).

In what follows all values and arithmetic operators are to be interpreted in their true mathematical sense unless otherwise indicated.

*2.7.2 Signed and Unsigned Integer Values*

The ATT2100 uses two common interpretations, denoted functionally, of a bit string $b$, Unsigned($b$) and Signed($b$). If the $n$-bit string $b$ is $b_{n-1}\ b_{n-2} \cdots b_2\ b_1\ b_0$, then:

$$\text{Unsigned}(b) = b_{n-1}*2^{n-1} + b_{n-2}*2^{n-2} + \cdots + b_2*2^2 + b_1*2^1 + b_0*2^0$$

and

$$\begin{aligned}\text{Signed}(b) &= \text{Unsigned}(b) \text{ when } b_{n-1} \text{ is } 0\\ &= -(2^n - \text{Unsigned}(b)) \text{ when } b_{n-1} \text{ is } 1\end{aligned}$$

Bit $b_{n-1}$ is often called the "sign bit." In what follows, we denote specific bit strings in hexadecimal form with a "0x" prefix. To look at a common example in 16–bit arithmetic Unsigned(0xFFFF) is 65535 and Signed(0xFFFF) is −1. Note that the wordlength $n$ is crucial to signed interpretation; in 32-bit arithmetic Signed(0x0000FFFF) = Unsigned(0x0000FFFF) = 65535.

5. Bits 4 through 31 are ignored resulting in modulo 16 addressing. The upper bits should be zero for compatibility with future versions of the ATT2100.

The definition of signed values used in the ATT2100, called "two's complement," is but one of several possible representations, though it enjoys advantages described below. "Ones' complement" representation was popular with some manufactures until the 1960's and "signed magnitude" continues to be popular for floating-point representations. Just for completeness, they are defined for integers:

$$\begin{aligned}\text{OnesComplementSigned}(b) &= \text{Unsigned}(b) \text{ when } b_{n-1} \text{ is } 0\\ &= -(2^n - 1 - \text{Unsigned}(b)) \text{ when } b_{n-1} \text{ is } 1\end{aligned}$$

and

$$\begin{aligned}\text{SignedMagnitudeSigned}(b) &= \text{Unsigned}(b) \text{ when } b_{n-1} \text{ is } 0\\ &= -(\text{Unsigned}(b) - 2^{n-1}) \text{ when } b_{n-1} \text{ is } 1\end{aligned}$$

### *2.7.3 ATT2100 Integer Types*

The ATT2100 supports three widths of integers, 32-bit words, 16-bit half-words and 8-bit bytes. Using the language above, sign interpretation is done with $n = 32$, 16 and 8, respectively.

The ATT2100 extends all operands to word length before an operation and computes a word result, before possibly truncating a result to accommodate a byte or half-word destination. This so-called *sign-extension* requires that byte and half-word operands be given a sign interpretation as part of their addressing mode. Unsigned values are padded on the left with zeros; signed values are padded on the left with a copy of the sign bit.

This definition of sign-extension is obviously correct for positive values, padding on the left with zeros being innocuous. To see how it works for negative values, consider a negative byte $b = b_7 b_6 \cdots b_1 b_0$, that is with $b_7 = 1$. When extended to a word, $b$ would yield $B = 111...11 b_7 b_6 \cdots b_1 b_0$ $= 0xFFFFFF00 + b = 2^{32} - 2^8 + b$, from which we see:

$$\begin{aligned}\text{Signed}(B) &= -(2^{32} - (2^{32} - 2^8 + b))\\ &= -(2^8 - b)\\ &= \text{Signed}(b)\end{aligned}$$

On the other hand, because truncation on the left may remove sign information, narrow results are prone to misinterpretation. For example, consider the byte product of bytes 0x05 and 0x33, which have the values 5 and 51 regardless of sign interpretation. The result is 0x000000FF, unambiguously 255. But this is 0xFF when trimmed to a byte, which may be mistaken for −1 later.

### *2.7.4 Two's Complement Arithmetic*

Before looking at the ATT2100 arithmetic in particular, let's see what makes two's complement arithmetic so desirable. In the last section we saw that the ATT2100 sign-extends all operands to word length. But what is the sign interpretation of that word operand (recall that the word addressing modes are mute about sign)? The answer is, "It usually doesn't matter," because of the following:

**Theorem:** *When applied to word operands, the operations addition, subtraction and multiplication produce word results which are* **independent** *of the sign interpretation of the operands.*

That is, two's complement arithmetic dovetails so nicely with unsigned arithmetic that one need implement only one version of addition, subtraction and multiplication to serve both needs. This simplifies the hardware and shrinks the instruction set. To amplify the theorem, the only distinction between +,− and * applied to signed or unsigned is in the side-effect of (signed and unsigned) overflow of the word result, hence the C- and V-bits below.

Note that the theorem applies to multiplication only when the result is the same width as the operands. Machines which produce the true double width result must distinguish between signed and unsigned

multiplication. The operations division and remainder always require variants for signed and unsigned interpretation of the operands. Hence **DIV** vs. **UDIV** and **REM** vs. **UREM** in the ATT2100.

*2.7.5 ATT2100 Integer Arithmetic Operations*

The ATT2100 integer arithmetic is performed in the following four straightforward steps:

1) The source operand(s) are extended to word length;

2) The mathematically correct result of the operation is computed;

3) Rounding of any nonintegral result to the nearest integer toward zero is performed, and any signed and unsigned overflow are recorded in the V- and C-bits, respectively;

4) The result, truncated on the left to fit a byte or half-word destination if required, is delivered to the destination.

The extension in (1) is as discussed in Section 2.7.3, with sign interpretation given by the addressing mode. The "mathematically correct" result in (2) depends on the sign interpretation of the operands, *but this is independent of the kind of extension in (1);* in the case of **[U]DIV** and **[U]REM** the interpretation is determined by the opcode, and in the case of **ADD**, **MUL** and **SUB** results are computed for *both* interpretations which, in light of the theorem, differ only in overflow conditions. A nonintegral result in (3) is only possible in the case of **DIV** or **UDIV**. C- and V-bits in (3) are discussed below. Truncation in (4) may cause loss of information as demonstrated in Section 2.7.3.

The ATT2100 offers seven arithmetic instructions, with two-and-a-half-address variants for all but **UDIV** and **UREM**:

```
ADD     a,b     ; add a into b
DIV     a,b     ; divide b by a, signed
MUL     a,b     ; multiply a into b
REM     a,b     ; calculate the remainder of signed division of b by a
SUB     a,b     ; subtract a from b
UDIV    a,b     ; divide b by a, unsigned
UREM    a,b     ; calculate the remainder of unsigned division of b by a
```

**REM** and **UREM** are defined in terms of **DIV** and **UDIV**, respectively, and are described more fully in Section 2.7.9. As shown in Section 2.6, operands *a* and *b* may be referenced using a variety of addressing modes, with sign interpretation given for byte and half-word arguments.

*2.7.6 The Carry Bit C and Unsigned Overflow*

On the ATT2100, the Carry bit C indicates the occurrence of a borrow during subtraction, or of unsigned overflow during addition or multiplication. Unsigned overflow arises when a result exceeds Unsigned(0xFFFFFFFF). In terms of the operations above, the PSW C-bit is set precisely when unsigned borrow on a subtract:

$$\text{Unsigned}(b) - \text{Unsigned}(a) < 0$$

or unsigned overflow on an addition or multiplication:

$$\text{Unsigned}(b) \{ + \text{ or } * \} \text{Unsigned}(a) > \text{Unsigned}(\text{0xFFFFFFFF})$$

Unsigned overflow does not apply to the signed operations **DIV** and **REM** and cannot occur in **UDIV** and **UREM**.

### *2.7.7 Hardware versus ATT2100 Arithmetic versus Mathematics*

Careful inspection of the operations of addition and subtraction gives some insight into the interplay between the various disciplines at hand. Internal to the ATT2100 is an *adder* circuit which is capable of computing the mathematical sum of two unsigned word length numbers, with a thirty-third bit on the left to catch the possible *carry-out*.

In the **ADD** operation, the adder computes the sum of $a$ and $b$; the word result is delivered and, if carry-out occurs, the C-bit is set. This is all quite intuitive. However, in the **SUB** operation, the two's complement of $a$ (that is, $2^{32}$–Unsigned($a$)) is added to $b$, except in this case the C bit is set only if *no* carry-out occurs.

This subtlety in the definition of the PSW C-bit, which is quite useful in practice, means that, contrary to ones' expectations since elementary school, adding $-x$ and subtracting $x$ are not identical on the ATT2100.

### *2.7.8 The oVerflow Bit V and Signed oVerflow*

Analogous to the C-bit, the oVerflow bit V signals the occurrence of signed overflow of the word result of an arithmetic operation, this is a result outside the interval:

[Signed(0x80000000) , Signed(0x7FFFFFFF)]

In terms of the operations above, the PSW V-bit is set **unless**

Signed(0x80000000) ≤ ( Signed($b$) {+,– or *} Signed($a$) ) ≤ Signed(0x7FFFFFFF)

Signed overflow does not apply to the unsigned operations **UDIV** and **UREM** and cannot occur in **REM**. Signed overflow does arise in **DIV** in precisely the case of 0x80000000 divided by -1 (i.e. 0xFFFFFFFF).

### *2.7.9 A Note About Division and Remainder*

**UDIV** never suffers unsigned overflow because its dividend is at most Unsigned(0xFFFFFFFF) and its divisor is no less than 1 (except for a zero divisor, which triggers a divide by zero exception), so its result is no greater than its dividend. A similar argument applies to **DIV**, except for the sole case of overflow.

Like **UDIV**, **UREM** never suffers unsigned overflow. To see why, consider the word results *UD* and *UR* of the operations **UDIV** and **UREM** applied to operands $a$ and $b$. **UDIV** and **UREM** are related by the formula:

$$b = (UD * a) + UR, \text{ where } 0 \le UR < a$$

with all values unsigned. It's easy to see that *UR* is no greater than $a$ and therefore no greater than Unsigned(0xFFFFFFFF), hence, overflow cannot occur. A similar argument applies to **REM**.

## 2.8 Tagged Integer Arithmetic

This section will be provided in Release 1.7.

## 2.9 ATT2100 Floating-Point Arithmetic

This section will be provided in Release 1.7.

## 2.10 A Fast Calling Sequence

The ATT2100 microprocessor provides an efficient procedure calling sequence. Procedure call overhead includes copying the outgoing arguments to an argument area, saving the return address, transferring control to the called procedure, and allocating a new stack-frame for the new procedure's local variables, temporaries and outgoing arguments. The ATT2100 calling sequence accomplishes these goals with as few operations as possible. A typical stack-frame is depicted in Figure 2-13.

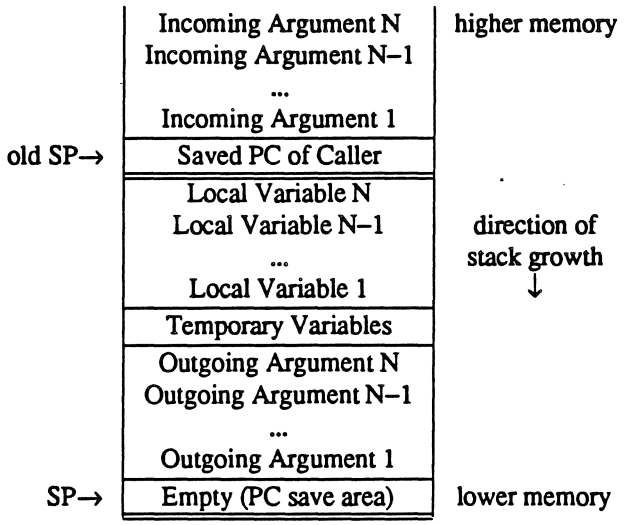



**Figure 2-14.** Typical Stack-frame

The stack grows downward in memory with the Stack Pointer (SP) always pointing to a free memory location. This free slot is where the Program Counter (PC) is stored on a procedure call (or unimplemented instruction exception). This avoids having to adjust the SP to save or restore the PC. The PC is the only machine register implicitly saved during a procedure call. Above the saved PC slot in the stack-frame is an area large enough to store outgoing arguments for any call from the current procedure. Above the outgoing arguments are stored temporary values and local variables. Thus outgoing arguments may be calculated in place with stack offset addressing modes. This statically allocated stack-frame allows the SP to be updated only on procedure entry or procedure return. Traditional *push* or *pop*[6] instructions which automatically adjust the SP are intentionally avoided. Therefore, side-effects to the SP are nearly eliminated and operand address generation for subsequent instructions may smoothly proceed in a pipelined implementation.

The steps required for a procedure call are straightforward. Outgoing arguments are *moved* (or even better: *calculated*) onto the stack-frame. In the event of word arguments, the first argument is stored at SP+4, the second at SP+8, etc. The **CALL** instruction performs an atomic move and jump operation, saving the return point at the CSP and loading the PC with the address of the first instruction of the called procedure. This first instruction of the called procedure, **ENTER**, adjusts the SP to allocate its new stack-frame. The last instruction of the called procedure, **RETURN**, re-adjusts the SP to deallocate its stack-frame and then branches to the address pointed to by the SP. Customarily, a **CATCH** follows the **RETURN**.

This procedure call overhead: call, allocate, deallocate, and return, can be as little as four clock cycles!

### 2.11 Prefetching Strategy

The ATT2100 provides two types of instruction fetching selectable through the Configuration Register PM-bit: prefetching and demand-fetching. When prefetching is enable (CONFIG PM-bit set to 1) the Prefetch Unit on the ATT2100 fetches text, which has not been previously fetched and stored in the prefetch buffer memory, in quad-word pieces consisting of two double-word I/O requests. Text is prefetched sequentially until a branch (predicted jump, unconditional jump, **CALL**, **CRET**, **KCALL**, **KRET** or **RETURN**) is decoded. If the target of the branch is encoded in the instruction (non-indirect),

6. **POPN** is provided to deallocate from the stack-frame and is useful in tail recursion.

prefetching then continues from the target (if it is not already in the prefetch buffer); if the target is indirect, prefetching stops and waits for a demand fetch request from the Execution Unit. A demand fetch is requested if the Execution Unit takes a miss-predicted or indirect branch and the target has not been previously decoded. If at any time while the Prefetch Unit is prefetching sequential code and following taken branches a demand fetch is requested, any I/O requested by the unit will complete and prefetching begins anew from the Execution Unit requested target.

If demand fetching is enabled (CONFIG PM-bit set to 0), the Prefetch Unit only issues an I/O request for text when it is requested by the Execution Unit and is not stored in the prefetch buffer. The I/O request is made for a double-word and all instructions contained in the double-word are decoded, but prefetching ceases until another demand fetch is requested by the Execution Unit. Demand fetching is the default mode on reset of the chip.

### 2.12 Tracing Instructions

Instruction tracing is supported by setting of the PSW TB- and TI-bits. These bits control when tracing is enabled as discussed in Section 2.4.8.

If an instruction is tracable, a trace exception is taken after the instruction completes execution. The PC saved on the interrupt stack is the PC of the next instruction.

Instructions before folded branches cannot be traced ( i.e., if a jump is folded into the previous instruction, the trace will occur after the jump.) To circumvent this from occurring, all jumps must be encoded as three-parcel and hence, will not be folded.

Event sequences are non-tracable. This includes exceptions and interrupts. The unimplemented instruction sequence is tracable a the trace bits are not altered.

**CRET, KCALL** and **KRET** are always non-traceable.

### 2.13 Event Processing

There are several sequences which can be triggered in the ATT2100 that are not usually invoked by the regular instruction set. These events include, in order of priority:

- reset
- interrupt
- exception

The sequences executed by the ATT2100 for each of these events are listed in the following sections. In all cases, interrupts are inhibited while an event processing sequence is in progress.

As described in the following sections, the processing of exceptions, interrupts and unimplemented instructions includes the saving of the PC on the interrupt stack. There are some subtleties to note respecting the definition of the "current PC" being saved; these subtleties are described in the notes portion of the instruction descriptions given in Section 2.14 for the CPU and flow control instructions which save a PC value.

#### *2.13.1 Reset*

The ATT2100 enters the reset sequence when:

1. the external reset pin is asserted.
2. a memory fault, which is signaled either externally or by the MMU,
   - occurs when attempting to read or write the Interrupt Stack during any event processing sequence.

- occurs when attempting to read from the Vector Table during any event processing sequence.

The reset sequence is:

```
disable interrupts
invalidate the PFB and IC
if (reset pin)
        SHAD = 0x0
else
        SHAD = PSW
PSW = 0x0
CONFIG = 0x0
PC = 0x0
enable interrupts
```

As indicated above, after a reset, SHAD is set to either 0x0 or the current PSW depending upon which type of reset occurred. Independent of the type of reset, the PFB and IC are flushed and PSW, CONFIG and the PC initialized to 0x0. 0x0 in the PSW Register sets the execution level to kernel, with physical addressing enabled, tracing disabled, interrupts inhibited and the ISP as the CSP. 0x0 in the CONFIG Register disables all on chip caches, disables timer interrupts and selects demand prefetching. 0x0 in the PC Register starts executing instructions at physical address 0x0.

**Caution:** If the reset sequence was initiated by the external reset pin, the SP and the MSP are undefined. The caches should not be enabled until these registers are assigned values since the range check circuitry will not know whether an address should access the on-chip stack cache or off-chip to memory.

*2.13.2 Interrupt*

An interrupt is signaled when an external device requests service on the interrupt request input lines IL<2:0> or either TIMER1 or TIMER2 overflows with interrupts enabled. The three input lines associated with external interrupts and the timer interrupts, which are asserted at level one, are compared with the PSW Interrupt Priority Level (IPL) field. If the interrupt request is logically less than the IPL field, the interrupt can be serviced. An IPL field of 7 allows interrupts at levels 0 through 6. An IPL field of 0 inhibits interrupts 1 through 6 and allows only interrupts at level 0, which is referred to as a "non-maskable interrupt".

The interrupt request input lines IL<2:0> must be asserted with the same value for at least two cycles before an interrupt is recognized by the ATT2100. If the level of the interrupt request is less-than or equal-to the IPL field of the PSW, the interrupt is accepted and the interrupt request enters at the top of the execution-unit pipeline. Once the interrupt enters the top of the execution-unit pipeline, all further interrupts are disabled until completion of the interrupt sequence outlined in Section 2.13.2.1. No indication is given by the ATT2100 as to when the interrupt is being serviced other than the I/O caused by the interrupt sequence and the subsequent handler.

A non-maskable interrupt can be generated by setting IL<2:0> to 0x0. An interrupt at level 0 is "edge sensitive" in that once asserted, it must be de-asserted for at least two cycles before another interrupt at any level is recognized. Once any interrupt enters the execution pipeline all interrupts are disabled, including NMI. After the interrupt sequence completes, if the NMI is still asserted, it will be serviced.

Most instructions complete execution before the interrupt request enters the top of the execution-unit pipeline. **CATCH, ENTER, MUL[3], DIV[3], REM[3], UDIV** and **UREM** are interruptible. The "CATCH" portion of **CRET** is interruptible. The PC stored on the interrupt stack is the proper value for transparently resuming execution. **CATCH, ENTER** and the "CATCH" portion of **CRET** continue as opposed to restarting.

*2.13.2.1 Interrupt Sequence*

When the interrupt is serviced, the sequence is:

```
disable interrupts
if (CSP == ISP) ISP = SHAD
else  SP = SHAD
*(ISP – 8) = PC of interrupted instruction        /* Becomes R8 wrt new ISP */
*(ISP – 4) = PSW                                  /* Becomes R12 wrt new ISP */
ISP –= 16
SHAD = ISP
PC = *(VB + 16 + (4*interrupt level))
PSW<UL,IPL[0:3],E,X,S,TB,TI,V,C,F,R[0:3]> = <0,0x0,0,0,0,0,0,0,0,0,0x0>
enable interrupts
```

Where "interrupt level" is the value of the IL<2:0> lines producing the interrupt. The ATT2100 state is the same as immediately after a reset, except that the PSW VP-bit and UA-bits and CONFIG do not change. Note that the interrupt sequence is almost the same as the **KCALL** sequence. In particular, the event frame left on the Interrupt Stack is the same, so a **KRET** instruction is sufficient for returning from an interrupt as well as, interrupts are disables during this processing.

*2.13.3 Exceptions*

Exceptions signal an error in a program. Exceptions, preceded by their respective ID, can occur in several ways:

0x1 An integer division by zero.

0x2 A trace operation, single or block.

0x3 An illegal instruction is executed. Illegal instructions include:

- An instruction with an immediate as a destination (other than **CMP** and two-and-one-half-operand instructions).
- A CPU instruction followed by an instruction other than a **MOV** which uses registers as both source and destination.
- A CPU instruction followed by an instruction which uses an addressing mode other than those listed in Table 11-10.
- A CPU instruction followed by a **MOVA** instruction with a source addressing mode of register.
- A CPU instruction followed by a single-parcel instruction.
- A CPU instruction followed by a monadic instruction.
- A CPU instruction followed by an **ADDI, ANDI** or **ORI.**
- A CPU instruction followed by a **DQM.**
- A CPU instruction followed by a **DIV, DIV3, MUL, MUL3, REM, REM3, UDIV** or **UREM.**
- A CPU instruction followed by a **TADD** or **TSUB.**
- A CPU instruction followed by a three-parcel monadic or the corresponding five-parcel slot.

— A **CPU** instruction followed by the last three-parcel or five-parcel instruction slot, opcode all ones.

— A conditional three-parcel branch instruction which uses indirect addressing.

— A **MOVA** instruction with an immediate source operand.

— A six byte monadic instruction with a byte or half-word addressing mode (i.e., ≤0xC); this includes the unimplemented monadics.

— **DQM** with source non-word addressing modes or destination addressing modes other than 0x0, 0x4, 0x8, 0xC, 0xE or 0xF.

— An **ENTER** with a negative Stack Offset Addressing mode.

— A six byte **RETURN** instruction with a source operand mode other than word stack offset (0xD) and negative value stack offsets.

— a **POPN** instruction with an immediate source operand.

0x4 Alignment faults. Alignment faults include:

— Data accesses without natural alignment.

— **DQM** with miss-aligned source addresses.

0x5 Privilege violations. Privilege violations include:

— An instruction which attempts to write a register while the ATT2100 is not in kernel execution level.

— Execution of a **CRET** or **KRET** in user mode. Note that these two instructions are the the only privileged instructions.

0x6 Accesses to unimplemented registers.

0xB MMU table walk access terminated by assertion of the I/O bus error input.

0x7 Instruction fetches terminated by a fault. Such faults are signaled by:

— Violation of the User/Kernel access bits in a PTE or non-paged segment STE.

— Invalid STE or PTE.

— Failure of the bounds test on a non-page segment.

0x8 Data read terminated by a fault. Such faults are signaled by:

— Violation of the User/Kernel access bits in a PTE or non-paged segment STE.

— Invalid STE or PTE.

— Failure of the bounds test on a non-page segment.

0x9 Data write terminated by MMU fault. Such faults are signaled by:

— Violation of the User/Kernel access bits in a PTE or non-paged segment STE.

— Invalid STE or PTE.

— Failure of the bounds test on a non-page segment.

0xA Memory access terminated by assertion of the I/O bus error input.

The exception handler must always be present.

*2.13.3.1 Exception Sequence*

```
disable interrupts
if (CSP == ISP) ISP = SHAD
else  SP = SHAD
*(ISP – 12) = "exception identifier"          /* Becomes R4 wrt new ISP */
*(ISP – 8) = PC of faulted instruction        /* Becomes R8 wrt new ISP */
*(ISP – 4) = PSW                              /* Becomes R12 wrt new ISP */
ISP –= 16
SHAD = ISP
PC = *(VB + 4)
PSW<UL,IPL[2:0],E,X,S,TB,TI,V,C,F,R[3:0]> = <0,0x0,0,0,0,0,0,0,0,0,0,0x0>
enable interrupts
```

Again, the sequence is almost the same as that of **KCALL**. See Table 11-12 for the "exception identifier" codes.

If the target address of a **CALL**, **CRET**, **JMP**, **KCALL**, **KRET** or **RETURN** instruction, or of an interrupt, causes a memory fault, the PC saved on the Interrupt Stack is the target PC, *not* the address of the current instruction.

In the case of exception IDs 0x8 and 0x9, the 32b operand aligned virtual address of faulted access is saved in the Fault Register.

For exception ID 0xA, the PC placed on the interrupt stack **is not** the PC of the instruction associated with the faulted access. Due to the "unhinged" nature of stores in the ATT2100, it is the PC of the instruction which was at the bottom of the execution pipeline when the fault occurred.

*2.13.4 Unimplemented Instruction*

An attempt to execute an unimplemented opcode results in an Unimplemented Instruction sequence. This sequence is faster than the exception sequence facilitating software emulation of extended instructions. As an unimplemented instruction can occur in either execution mode, the unimplemented instruction handler should be in the user address space.

If an unimplemented instruction has an addressing mode which is illegal for that instruction class, it is considered an illegal instruction (exception ID 0x3). Specifically:

1. An unimplemented monadic instruction is considered illegal if it has a non-word addressing mode ( < 0xC).
2. An unimplemented instruction is considered illegal if it follow a **CPU** instruction and contains an addressing mode, or combination of modes, which Section 2.13.3 lists as illegal for an instruction following a **CPU** instruction.
3. **RETURN** with a negative operand.

There are no tests performed upon the addressing modes of unimplemented dyadic instructions which do not follow **CPU** instructions; this includes unimplemented dyadics reserved for floating-point.

*2.13.4.1 Unimplemented Instruction Sequence*

```
*(CSP) = PC of unimplemented opcode
PC = *(VB + 12)
```

where CSP is either SP or ISP, depending upon the state of the PSW S-bit.

*2.13.5 Trapped Niladics*

An attempt to execute a one-parcel niladic with an opcode in the range 0x200 through 0x3FF results in a variant of the previously described Unimplemented Instruction sequence called the Trapped Niladic exception. This sequence is the same as the Unimplemented Instruction sequence except VB + 8 is used for the vector.

The trapped niladic handler should be in the user address space.

*2.13.5.1 Trapped Niladic Sequence*

*(CSP) = PC of unimplemented opcode
PC = *(VB + 8)

where CSP is either SP or ISP, depending upon the state of the PSW S-bit.

*2.13.6 Event Processing Priority*

Given that several event requests can be generated simultaneously, an event processing priority must be established. The priorities assigned to each event type are:

1. reset,
2. interrupt,
3. timer interrupts,
4. trace,
5. instruction fetch faults,
6. illegal instructions,
7. unimplemented instructions,
8. unimplemented registers,
9. alignment faults,
10. operand faults,
11. privilege violation,
12. divide by zero.

Events 4 through 12 are associated with a particular instruction, while the higher priority events (reset, interrupt, and timer) can occur independent of what instruction is being executed. During some internal sequences interrupts are disabled. Many events given in the list above are mutually exclusive of each other and can not occur at the same time or within the same instruction.

**2.14 Instructions**

The format for instructions has already been described. The instructions themselves may be divided into nine categories. The following special notation has been used: [] and (|). **ADD[3]** for example, indicates that both **ADD** and **ADD3** instructions exist. **JMP(F|T)(Y|N)** indicates that **JMPFY, JMPFN, JMPTY,** and **JMPTN** instructions exist.

- Arithmetic
  - **ADD[3]** – add
  - **ADDI** – add interlocked
  - **DIV[3]** – divide

- **MUL[3]** – multiply
- **REM[3]** – remainder
- **SUB[3]** – subtract
- **UDIV** – unsigned divide
- **UREM** – unsigned remainder

- Logical
  - **AND[3]** – bitwise logical and
  - **ANDI** – bitwise logical and interlocked
  - **OR[3]** – bitwise logical or
  - **ORI** – bitwise logical or interlocked
  - **XOR[3]** – bitwise logical exclusive or
- Shift
  - **SHL[3]** – left shift
  - **SHR[3]** – arithmetic right shift
  - **USHR[3]** – logical right shift
- Compare
  - **CMPEQ** – equality comparison
  - **CMPGT** – signed greater than comparison
  - **CMPHI** – high comparison (unsigned greater than)
- Move
  - **DQM** – double or quad word move
  - **LDRAA** – load PC-relative address into accumulator
  - **MOV** – move
  - **MOVA** – move address
- Program Control
  - **CALL** – call subroutine
  - **CATCH** – fill stack cache
  - **CRET** – return from kernel with context
  - **ENTER** – enter subroutine and allocate procedure frame
  - **JMP** – unconditional jump
  - **JMP(F|T)(Y|N)** – conditional jump
  - **KCALL** – kernel call
  - **KRET** – return from kernel
  - **POPN** – free N entries from procedure frame
  - **RETURN** – free procedure frame and return from subroutine
- Tagged
  - **TADD** – tagged addition
  - **TSUB** – tagged subtraction
- Floating-point (unimplemented)
  - **FADD[3]** – floating-point addition
  - **FCLASS** – floating-point classify
  - **FCMP** – floating-point compare
  - **FDIV[3]** – floating-point division
  - **FLOGB** – floating-point exponent extraction

- **FMOV** –floating-point move
- **FMUL[3]** – floating-point multiply
- **FNEXT** – floating-point next-after
- **FREM** – floating-point remainder
- **FSCALB** – floating-point scaling by a power of radix, 2
- **FSQRT** – floating-point square root
- **FSUB[3]** – floating-point subtraction

- Other
  - **CLRE** – clear PSW E-bit
  - **CPU** – register mode escape
  - **FLUSHD** – flush the Data Cache (unimplemented)
  - **FLUSHDCE** – flush an entry in the Data Cache (unimplemented)
  - **FLUSHI** – flush the Decoded Instruction Cache
  - **FLUSHP** – flush the Prefetch Buffer Cache
  - **FLUSHPBE** – flush an entry in Prefetch Buffer Cache
  - **FLUSHPTE** – flush a page-table-entry in the Translation Look-aside Buffers or Segment Registers
  - **NOP** – no operation
  - **TESTC** – copy PSW carry-bit to PSW flag-bit and clear carry-bit
  - **TESTV** – copy PSW overflow-bit to PSW flag-bit and clear overflow-bit

The next section contains detailed descriptions of the instruction set. The following abbreviations are used:

abs32 A 32-bit value with any of the two word operand addressing modes:

- PC-relative or
- absolute.

fgen[*n*] Any of the following modes with a value that can fit in *n*-bits:

- absolute,
- immediate,
- stack offset, or
- stack offset indirect.

flow32 A 32-bit value with any of the four word operand addressing modes (i.e., modes ≥ 0xC):

- absolute,
- absolute indirect,
- PC-relative, or
- stack offset indirect mode.

gen[*n*][7] Any of the following modes with a value that can fit in *n*-bits:

- absolute,
- immediate,
- stack offset, or
- stack offset indirect.

imm[*n*] A two's complement constant in the range $-2^{n-1}$ through $2^{n-1}-1$.

istk5 An Indirect Stack Offset mode of type word with the offset a number divisible by four in the range 0 through 124.

pcrel10 A pc-offset mode in which the offset is a number divisible by 2 in the range −1024 through 1022.

stk5 A Stack Offset mode with the offset a number in the range 0 through 124 and is divisible by four, which is the operand size of word.

stk8 A Stack Offset mode which is operand size of word.

stk32 A Stack Offset mode with the offset any 32-bit number.

uimm[*n*] An unsigned constant in the range 0 through $2^n-1$.

wai[*n*] An unsigned constant in the range 0 through $2^n-1$ which is multiplied by 4 (word aligned).

word32 A 32-bit value with any of the four word operand addressing modes (i.e., modes >= 0xC):

7. The CPU prefix instruction modifies the meaning of these addressing modes.

- absolute,
- immediate mode,
- stack offset, or
- stack offset indirect.

Name: ADD – ADDition

Format: **ADD[3]** src, dst

Operation:

**ADD:**

dst += src
"unsigned overflow" ? PSW.C = 1 : PSW.C = 0
"signed overflow" ? PSW.V = 1 : PSW.V = 0

**ADD3:**

Acc = dst + src
"unsigned overflow" ? PSW.C = 1 : PSW.C = 0
"signed overflow" ? PSW.V = 1 : PSW.V = 0

Description: The source operand is added to the destination operand and the sum is placed in either the destination (**ADD**) or the Accumulator (**ADD3**).

The PSW C-bit is set to 1 on unsigned overflow and the PSW V-bit is set to 1 on signed overflow, otherwise the PSW C- and V-bits are set to 0. See Section 2.7 for a full description of ATT2100 integer arithmetic.

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 2 | 0x0D | **ADD3** | wai5, | stk5 |
| 2 | 0x14 | **ADD** | imm5, | stk5 |
| 2 | 0x15 | **ADD3** | imm5, | stk5 |
| 2 | 0x16 | **ADD** | stk5, | stk5 |
| 2 | 0x17 | **ADD3** | stk5, | stk5 |
| 6 | 0x23 | **ADD** | gen16, | gen16 |
| 6 | 0x33 | **ADD3** | gen16, | gen16 |
| 10 | 0x23 | **ADD** | gen32, | gen32 |
| 10 | 0x33 | **ADD3** | gen32, | gen32 |

Name: ADDI – ADDition Interlocked

Format: **ADDI** src, dst

Operation:
hidden = dst
dst += src
Acc = hidden

Description: The source operand is added to the destination operand and the sum is placed in the destination. The lock pin is asserted during the fetch of *dst*, if *dst* is in memory and not in the stack cache. The lock pin is de-asserted at the completion of the final store to *dst*. No other accesses are done between the fetch and store of *dst*. The original value of *dst* (obtained during the fetch) is placed in the Accumulator. If the Accumulator is not in the stack cache, a store is made after the interlocked I/O completes.

The PSW C- and V-bits are not affected by an **ADDI** instruction.

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 6 | 0x03 | **ADDI** | gen16, | gen16 |
| 10 | 0x03 | **ADDI** | gen32, | gen32 |

Notes: Pipeline bypass hazards associated with semaphore operations are avoided in the ATT2100 by clearing the pipeline before an interlocked instruction enters the first pipeline stage. No other instruction is allowed into the pipeline until the executing interlocked instruction completes.

If R4 is the destination, after the interlocked instruction completes, R4 is the previous value of R4, hence no operation is performed.

If the accumulator is not in the SC, CSP == MSP, an I/O access is made to update the accumulator after the interlocked accesses complete. The access to the accumulator **must** not fault in any manner for the **ADDI** is not restartable from this point of the operation.

Name: AND – bitwise logical AND

Format: **AND[3]** src, dst

Operation: **AND:** dst &= src
**AND3:** Acc = dst & src

Description: A bitwise logical AND operation is performed on the source and destination operands. The result is placed in either the destination (**AND**) or the Accumulator (**AND3**).

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 2 | 0x0E | **AND3** | imm5, | stk5 |
| 2 | 0x0F | **AND** | stk5, | stk5 |
| 6 | 0x22 | **AND** | gen16, | gen16 |
| 6 | 0x32 | **AND3** | gen16, | gen16 |
| 10 | 0x22 | **AND** | gen32, | gen32 |
| 10 | 0x32 | **AND3** | gen32, | gen32 |

Name: ANDI – bitwise logical AND Interlocked

Format: **ANDI** src, dst

Operation:
hidden = dst
dst &= src
Acc = hidden

Description: A bitwise logical AND operation is performed on the source and destination operands and the result is placed in the destination. The lock pin is asserted during the fetch of *dst*, if *dst* is in memory and not in the stack cache. The lock pin is de-asserted at the completion of the final store to *dst*. No other accesses are done between the fetch and store of *dst*. The original value of *dst* (obtained during the fetch) is placed in the Accumulator. If the Accumulator is not in the stack cache, a store is made after the interlocked I/O completes.

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 6 | 0x02 | **ANDI** | gen16, | gen16 |
| 10 | 0x02 | **ANDI** | gen32, | gen32 |

Notes: Pipeline bypass hazards associated with semaphore operations are avoided in the ATT2100 by clearing the pipeline before an interlocked instruction enters the first pipeline stage. No other instruction is allowed into the pipeline until the executing interlocked instruction completes.

If R4 is the destination, after the interlocked instruction completes, R4 is the previous value of R4, hence no operation is performed.

If the accumulator is not in the SC, CSP = MSP, an I/O access is made to update the accumulator after the interlocked accesses complete. The access to the accumulator **must** not fault in any manner for the **ANDI** is not restartable from this point of the operation.

Name: CALL – subroutine CALL

Format: **CALL** src

Operation:

```
*(CSP) = next PC                    /* save return PC in R0 */
PC = src
```

Description: The next Program Counter value (return address) is stored at the location indicated by the SP, or the ISP, whichever is the CSP. The source operand (subroutine entry point) becomes the new Program Counter value.

Encodings

| length | opcode | subcode | instruction | src |
|---|---|---|---|---|
| 2 | 0x01 | - | **CALL** | pcrel10 |
| 6 | 0x00 | 0x1 | **CALL** | flow32 |

Note: If the location pointed to by the **CALL** instruction can not be referenced, a fetch-fault results. In this case, the PC stored on the interrupt stack is the target PC, not the PC of the original **CALL**. The address of the original **CALL** instruction is not saved. In the event of an indirect **CALL**, if the ATT2100 can not reference the indirection word, a read-fault results and the PC stored on the interrupt stack is that of the indirect **CALL**. In either case, fetch-fault or read-fault, the correct return PC is saved in R0.

Name: CATCH – fill stack cache

Format: **CATCH** src

Operation:

```
if ( CSP = = SP) {
        while( ( MSP < (CSP + src) ) && ( (MSP – SP) < SCSIZE) )
        {
                stack_cache[ MSP ] = memory[ MSP ]
                stack_cache[ MSP+4 ] = memory[ MSP+4 ]
                stack_cache[ MSP+8 ] = memory[ MSP+8 ]
                stack_cache[ MSP+12 ] = memory[ MSP+12 ]
                MSP += 16
        }
}
```

Description: If the CSP is SP, the stack cache is filled to the extent indicated by the source operand. The semantics of **CATCH** are somewhat different depending upon the address mode of *src*.

1. If the source operand is defined with a Stack Offset mode (Roffset), the address is formed by adding the offset to the SP to determine the target value for the MSP (MSP = SP + offset).
2. If the source operand is defined with an Immediate mode ($data), the immediate value is used as the target for the MSP (MSP = data).
3. If the source operand is defined with a Stack Offset Indirect mode (*Roffset), the target value for the MSP is fetched from memory (or the Stack Cache) at the address formed by adding the offset to SP (MSP = *(offset + SP)).
4. If the source operand is defined with an Absolute mode (*$addr), the target value for the MSP is fetched from memory (or the Stack Cache) at the address specified in the absolute address (MSP = *(addr)).

In no case will the MSP be incremented beyond the size of the on-chip stack cache. If the CSP is the ISP, **CATCH** is a no-op.

ENCODINGS

| length | opcode | subcode | instruction | src |
|---|---|---|---|---|
| 2 | 0x02 | 0x1 | **CATCH** | stk8* |
| 6 | 0x00 | 0x8 | **CATCH** | word32 |

Notes: The MSP *must* be greater than or equal to the SP when **CATCH** executes, otherwise instruction operation depends upon context and is therefore unpredicatable.

* The 8-bit stack offset is zero-extended and multiplied by 16 giving it an effective range of 0 through 4080 in quad-aligned increments.

If virtual addressing is enabled, and the MSP is updated, the new value is checked to verify that stores are valid at the current execution level. If the address is not valid, either a Read Fault, exception ID 0x8, or a MMU Table Walk Fault, exception ID 0xB, is flagged for the **CATCH** instruction.

Since the lower four bits of the SP do not exist, cache filling is done in 16 byte blocks. If the source operand to **CATCH** is not divisible by 16, the cache is filled to the next multiple of 16.

Name: CLRE – clear PSW E-bit

Format: **CLRE**

Description: The **CLRE** instruction clears the PSW E-bit. The PSW E-bit is set by an **ENTER** instruction which has successfully completed execution.

Encodings

| length | opcode | subcode | instruction |
|---|---|---|---|
| 2 | 0x0B | 0xA | **CLRE** |

Name: CMP – CoMPare

Format: **CMPrel** src1, src2

Operation: src1 *rel* src2 ? PSW.F = 1 : PSW.F = 0

Description: The PSW F-bit is set to 1 if the comparison between the two source operands is true. If the comparison is false the PSW F-bit is set to 0. *Rel* is one of the following:

**EQ** equal to

**GT** signed greater than

**HI** higher (unsigned greater than)

Encodings

| length | opcode | instruction | src1 | src2 |
|---|---|---|---|---|
| 2 | 0x10 | CMPEQ | imm5, | stk5 |
| 2 | 0x11 | CMPGT | stk5, | stk5 |
| 2 | 0x12 | CMPGT | imm5, | stk5 |
| 2 | 0x13 | CMPEQ | stk5, | stk5 |
| 6 | 0x1D | CMPGT | gen16, | gen16 |
| 6 | 0x1E | CMPHI | gen16, | gen16 |
| 6 | 0x1F | CMPEQ | gen16, | gen16 |
| 10 | 0x1D | CMPGT | gen32, | gen32 |
| 10 | 0x1E | CMPHI | gen32, | gen32 |
| 10 | 0x1F | CMPEQ | gen32, | gen32 |

Notes: *src1* is specified in the source operand field. *src2* is specified in the destination operand field.

**CMPEQ** may be used to test either = or ≠, **CMPGT** may be used to test signed >, ≥, <, ≤, and **CMPHI** may be used to test unsigned >, ≥, <, ≤. In the latter cases, it is simply a matter of ordering the operands properly and testing the correct sense of the PSW F-bit.

Name: CPU – register access escape

Format: **CPU**

Description: The **CPU** instruction is a prefix which changes the meaning of the instruction which follows it. Specifically, it changes the definition of address modes to enable access to the internal registers. All word-sized address modes remain the same, while mode 0x7 becomes the register addressing mode, (see Section 2.6.7). The register number is stored in the operand field. The low four bits of the operand are used as the register number; The high-order bits are ignored, but should be zero. Accessing the undefined register 0 results in an Unimplemented Instruction exception.

Encodings

| length | opcode | subcode | instruction |
|---|---|---|---|
| 2 | 0x0B | 0x0 | **CPU** |

Notes:

The instruction following the **CPU** is considered part of the **CPU** instruction: if an exception or interrupt occurs the PC saved on the interrupt stack is the PC of the **CPU** instruction. In the Prefetch and Decode section of the ATT2100, the Program Counter is incremented by four- or six-parcels depending on whether the instruction following the **CPU** instruction is three- or five-parcels.

See Section 2.13.3 for legal and illegal use of the **CPU** instruction.

For future support of floating-point instructions, a **CPU** instruction followed by a reserved floating-point instruction should not be used.

Hazards:

The **CPU** is an interlocked instruction in that no other instruction is started until the **CPU** reaches the RR pipeline stage. It is still possible to cause a hazard between instructions which modify the PSW C-, E-, F- or V-bits. If the either of the two instructions proceeding the **CPU** instruction modified the PSW C-, E-, F- or V-bits, any access of the PSW should be padded by two **NOP** instructions.

See Section 2.4.8 for additional hazards associated with storing into the PSW.

Name: CRET – Context RETurn from kernel

Format: **CRET**

Operation:

```
disable interrupts
SP = *(ISP + 0)                 /* R0 wrt ISP */
fetch *(ISP + 4)
enable interrupts
CATCH(MSP – SP)
disable interrupts
MSP = *(ISP + 4)                /* R4 wrt ISP */
PC = *(ISP + 8)                 /* R8 wrt ISP */
PSW = *(ISP + 12)               /* R12 wrt ISP */
ISP += 16
if (CSP = = ISP)
        SHAD = ISP
else
        SHAD = SP
enable interrupts
```

Description: A new SP is loaded from the Interrupt Stack. The current contents of the stack cache are discarded and an unconditional **CATCH** is performed filling the stack cache to the MSP. The Program Status Word and Program Counter values are restored by "popping" the Interrupt Stack.

Encodings

| length | opcode | subcode | instruction |
|---|---|---|---|
| 2 | 0x0B | 0x5 | **CRET** |

Notes: The target MSP is fetched prior to the "CATCH" portion executes, but the MSP is not updated until the "CATCH" portion completes.

Interrupts are disabled as indicated in the Operation description during a portion of the **CRET**. Interrupts are enabled as indicated during the "CATCH" portion of **CRET** at the level of the restored PSW. The "CATCH" portion of of **CRET** is performed consistent with the restored PSW VP- and S-bits.

If a memory fault occurs while reading from the Interrupt Stack the ATT2100 resets.

The **CRET** instruction is privileged. If a **CRET** is initiated at the user level, a privilege exception is executed.

The **CRET** instruction can not be traced.

If the location pointed to by the new PC value can not be referenced, a fetch-fault results. In this case, the PC stored on the interrupt stack is the new PC value, not the address of the **CRET** instruction.

Name: DIV – DIVide

Format: **DIV**[3] src, dst

Operation: **DIV:** dst /= src
**DIV3:** Acc = dst / src

Description: The destination operand is divided by the source operand and the quotient is placed in either the destination (**DIV**) or the Accumulator (**DIV3**). Two's complement division is performed. See Section 2.7 for a full description of ATT2100 integer arithmetic.

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 6 | 0x27 | **DIV** | gen16, | gen16 |
| 6 | 0x37 | **DIV3** | gen16, | gen16 |
| 10 | 0x27 | **DIV** | gen32, | gen32 |
| 10 | 0x37 | **DIV3** | gen32, | gen32 |

Notes: Division by zero results in a zero divide exception. Division of 0x80000000 by 0xFFFFFFFF sets the PSW V-bit and returns the result 0x80000000. The V-bit is cleared in all the other cases. The C-bit is unchanged in all the cases.

Name: DQM – Double-word or Quad-word Move

Format: **DQM src, dst**

Operation: dst = src

Description: Double- or Quad-word Move moves either two or four contiguous words from the source to the destination. The size of the transfer is determined by the destination address mode field.

Double-word data size is encoded in the destination mode field as 0x0, x4 or 0x8. Quad-word data size is encoded in the destination mode field as 0xC, 0xD or 0xE. If the source mode is 0xF, the constant is replicated either two or four times depending upon the destination mode. If the destination mode is 0xF, an illegal instruction exception is taken. All other addressing modes result in an alignment fault.

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 6 | 0x07 | **DQM** | gen16*, | gen16* |
| 10 | 0x07 | **DQM** | gen32*, | gen32* |

Note: Source and destination addresses of quad-word operands must be divisible by 16 (quad-aligned) and addresses of double-word operands must be divisible by 8 (double-aligned). Otherwise an alignment exception occurs.

Only word addressing modes are permitted for the source and the special modes for the destination. Other modes trigger an illegal instruction sequence.

* The limitations given in the description and note apply.

Name: ENTER – ENTER subroutine

Format: **ENTER** src

Operation:

```
if( CSP = = ISP )
{
        SHAD = ISP = target*
}
if( (CSP = = SP) && (src address mode != Stack Offset) )
{
        /* flush stack cache unconditionally */
        while ( MSP > SHAD )
        {
                memory[ MSP–16 ] = stack_cache[ MSP–16 ]
                memory[ MSP–12 ] = stack_cache[ MSP–12 ]
                memory[ MSP–8 ] = stack_cache[ MSP–8 ]
                memory[ MSP–4 ] = stack_cache[ MSP–4 ]
                MSP –= 16
        }
        /* force stack cache to be empty */
        SHAD = MSP = SP = target
}
if( (CSP = = SP) && (src address mode = = Stack Offset) )
{
        /* flush only as much of the stack cache as is necessary */
        if ( MSP – target > SCSIZE)
        {
                while (( MSP ≥ SHAD) && (MSP – target > SCSIZE))
                {
                        memory[ MSP–16 ] = stack_cache[ MSP–16 ]
                        memory[ MSP–12 ] = stack_cache[ MSP–12 ]
                        memory[ MSP–8 ] = stack_cache[ MSP–8 ]
                        memory[ MSP–4 ] = stack_cache[ MSP–4 ]
                        MSP –= 16
                }
                if( MSP > (target + SCSIZE) )
                        MSP = target + SCSIZE
        }
        SHAD = SP = target
}
PSW.E = 1
```

*see Description for determination of target value

Description: The CSP is altered either by adding the source operand (Stack Offset addressing mode), or replacing it with a new value (all other addressing modes). If SP is not the current stack pointer no data traffic between the Stack Cache and memory is performed, and the MSP is not updated. If SP is the CSP, the contents of the stack cache are written to memory (if necessary) in quad word transfers until no more than *SCSIZE* bytes are held in the cache. The semantics of **ENTER** are somewhat different depending upon the address mode of *src*.

1. If the source operand is defined with a Stack Offset mode (Roffset), the address formed by adding the offset to the CSP is used to determine the target value (MSP = CSP + offset). The bounds of the stack cache are set to encompass the full amount of the **ENTER** instruction, within the limits of *SCSIZE*.
2. If the source operand is defined with an Immediate mode ($data), the immediate value is used as the target value (MSP = data) and the stack cache is set empty at the completion of the **ENTER** instruction.
3. If the source operand is defined with a Stack Offset Indirect mode (*Roffset), the target value is fetched from memory (or the Stack Cache) using the address formed by adding the offset to the CSP (MSP = *(offset + CSP)) and the stack cache is set empty at the completion of the **ENTER** instruction.
4. If the source operand is defined with an Absolute mode (*$addr), the target value for the CSP is fetched from memory (or the Stack Cache) using the address specified in the absolute address (MSP = *(addr)) and the stack cache is set empty at the completion of the **ENTER** instruction.

Upon successful completion of the **ENTER**, the PSW E-bit is set. The PSW E-bit is cleared with the **CLRE** instruction.

Encodings

| length | opcode | subcode | instruction | src |
|---|---|---|---|---|
| 2 | 0x02 | 0x0 | **ENTER** | stk8† |
| 6 | 0x00 | 0x9 | **ENTER** | word32 |

Notes:

If the 6 byte form of **ENTER** is used with a Stack Offset mode for *src*, the magnitude of the offset must be greater than *SCSIZE*, and the offset **must** be less than or equal to 0, or unpredictable results may occur. The MSP **must** be greater than or equal to the SP when an **ENTER** begins, otherwise instruction operation depends upon context and therefore is unpredictable.

For the Stack Offset Addressing model, only negative stack offsets are legal, positive stack offsets trigger an illegal instruction sequence. This includes **ENTER** R0.

If virtual addressing is enabled, the target address and the new MSP, if the MSP is updated, are checked to verify that stores are valid at the current execution level. If the addresses are not valid, a Read Fault exception, exception type 8, or MMU a MMU Table Walk Fault, exception ID 0xB, is flagged for the **ENTER** instruction. The exception is processed after any stack flushing is completed.

Since the lower four bits of the SP do not exist, the lower four bits of the source operand are ignored.

† The 8-bit stack offset is left padded with ones and multiplied by 16 giving it an effective range of -16 through -4096 in quad-aligned decrements.

Name: FADD – Floating-point ADDition

Format: **FADD[3]** src, dst

Operation: **FADD:** dst += src
**FADD3:** Acc = dst + src

Description: The source operand is added to the destination operand and the sum is placed in either the destination (**FADD**) or the floating-point Accumulator (**FADD3**).

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 6 | 0x2B | **FADD** | fgen16, | fgen16 |
| 6 | 0x3B | **FADD3** | fgen16, | fgen16 |
| 10 | 0x2B | **FADD** | fgen32, | fgen32 |
| 10 | 0x3B | **FADD3** | fgen32, | fgen32 |

Notes: May result in overflow, underflow, an inexact or an invalid operation. See Section 2.9 for a complete description of floating-point arithmetic.

The **FADD[3]** instruction is not implemented in hardware. An unimplemented instruction sequence is taken.

Name: FCLASS – Floating-point CLASSify

Format: **FCLASS** src,dst

Operation:

```
if (src = = "signaling NaN")
    dst = 1
else if (src = = "quiet NaN")
    dst = 2
else if (src = = "∞")
    dst = 3
else if (src = = "0")
    dst = 4
else if (src = = "normalized")
    dst = 5
else if (src = = "subnormalized")
    dst = 6
```

Description: The destination is set to the nonzero integral value whose sign is that of the source operand and whose magnitude is determined by the source.

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 6 | 0x13 | **FCLASS** | fgen16, | gen16 |
| 10 | 0x13 | **FCLASS** | fgen32, | gen32 |

Notes: The **FCLASS** operation never results in an exception. See Section 2.9 for a complete description of floating-point arithmetic.

The **FCLASS** instruction is not implemented in hardware. An unimplemented instruction sequence is taken.

Name: FCMP – Floating-point CoMParison

Format: **FCMPrel** src1,src2

Operation: src1 *rel* src2 ? PSW.F = 1 : PSW.F = 0

Description: The PSW F-bit is set to 1 if the comparison between the two source operands is true. If the comparison is false PSW F-bit is set to 0. *Rel* is one of the following comparisons:

EQ equal to

EQN equal to or unordered with

GT greater than

GE greater than or equal to

N unordered

Encodings

| length | opcode | instruction | src1 | src2 |
|---|---|---|---|---|
| 6 | 0x18 | **FCMPGE** | fgen16, | fgen16 |
| 6 | 0x19 | **FCMPGT** | fgen16, | fgen16 |
| 6 | 0x1A | **FCMPEQ** | fgen16, | fgen16 |
| 6 | 0x1B | **FCMPEQN** | fgen16, | fgen16 |
| 6 | 0x1C | **FCMPN** | fgen16, | fgen16 |
| 10 | 0x18 | **FCMPGE** | fgen32, | fgen32 |
| 10 | 0x19 | **FCMPGT** | fgen32, | fgen32 |
| 10 | 0x1A | **FCMPEQ** | fgen32, | fgen32 |
| 10 | 0x1B | **FCMPEQN** | fgen32, | fgen32 |
| 10 | 0x1C | **FCMPN** | fgen32, | fgen32 |

Notes: **FCMPGT** and **FCMPGE** signal invalid operation if the operands are unordered. See Section 2.9 for a complete description of floating-point arithmetic.

The **FCMP** instructions are not implemented in hardware. An unimplemented instruction sequence is taken.

Name: FDIV – Floating-point DIVision

Format: **FDIV[3]** src, dst

Operation: **FDIV:** dst /= src
**FDIV3:** Acc = dst / src

Description: The destination operand is divided by the source operand and the quotient is placed in either the destination (**FDIV**) or the floating-point Accumulator (**FDIV3**).

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 6 | 0x2A | **FDIV** | fgen16, | fgen16 |
| 6 | 0x3A | **FDIV3** | fgen16, | fgen16 |
| 10 | 0x2A | **FDIV** | fgen32, | fgen32 |
| 10 | 0x3A | **FDIV3** | fgen32, | fgen32 |

Notes: May result in overflow, underflow, division by zero, an inexact or an invalid operation. See Section 2.9 for a complete description of floating-point arithmetic.

The **FDIV[3]** instruction is not implemented in hardware. An unimplemented instruction sequence is taken.

Name: FLOGB – Floating-point exponent extraction

Format: **FLOGB** src, dst

Operation:

```
if (src = = "±0")
    dst = −∞
else if (src = = "±∞")
    dst = +∞
else if (src = = "NaN")
    dst = src
else
    dst = exponent of src
```

Description:

The destination receives the exponent of the source operand, in floating-point format, with the special cases.

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 6 | 0x12 | **FLOGB** | fgen16, | fgen16 |
| 10 | 0x12 | **FLOGB** | fgen32, | fgen32 |

Notes: Never results in an exception. See Section 2.9 for a complete description of floating-point arithmetic.

The **FLOGB** instruction is not implemented in hardware. An unimplemented instruction sequence is taken.

Name: FLUSHD – FLUSH Data cache

Format: **FLUSHD**

Description: The Data Cache is flushed: all entries are marked invalid.

Encodings

| length | opcode | subcode | instruction |
|---|---|---|---|
| 2 | 0x0B | 0x6 | **FLUSHD** |

Note: The **FLUSHD** instruction is not implemented in hardware as there is no data cache. An unimplemented instruction sequence is taken.

Name: FLUSHDCE – FLUSH a Data Cache Entry

Format: **FLUSHDCE** src

Description: The quad-word at *src*, is flushed from the DC.

Encodings

| length | opcode | subcode | instruction | src |
|---|---|---|---|---|
| 6 | 0x00 | 0xD | **FLUSHDCE** | word32 |

Note: The **FLUSHDCE** instruction is not implemented in hardware as there is no data cache. An unimplemented instruction sequence is taken.

Name: FLUSHI – FLUSH decoded Instruction cache

Format: **FLUSHI**

Description: The Decoded Instruction Cache is flushed: all entries are marked invalid.

Encodings

| length | opcode | subcode | instruction |
|---|---|---|---|
| 2 | 0x0B | 0x3 | **FLUSHI** |

Name: FLUSHP – FLUSH Prefetch buffer cache

Format: **FLUSHP**

Description: The Prefetch Buffer Cache is flushed: all entries are marked invalid.

Encodings

| length | opcode | subcode | instruction |
|---|---|---|---|
| 2 | 0x0B | 0x4 | **FLUSHP** |

Name: FLUSHPBE – FLUSH a PreFetch Buffer Entry

Format: **FLUSHPBE** src

Description: The quad-word at *src*, is marked invalid in the PFB. None of the other caches are affected.

Encodings

| length | opcode | subcode | instruction | src |
|---|---|---|---|---|
| 6 | 0x00 | 0xC | **FLUSHPBE** | word32 |

Name: FLUSHPTE – FLUSH a Page Table Entry from the tlb's

Format: **FLUSHPTE** src

Description: If there is a Page Table Entry for the address defined by *src*, in either the Text or Data Translation Look-aside Buffers, the entry is marked invalid. Both the Text and Data Non-paged Segment Registers are invalidated.

Encodings

| length | opcode | subcode | instruction | src |
|---|---|---|---|---|
| 6 | 0x00 | 0xB | **FLUSHPTE** | word32 |

Note: For the **FLUSHPTE** instruction the *src* operand is an address. Normally, the address would be moved into the Stack Cache and the Stack Offset Indirect addressing mode would be used for *src*.

Name: FMOV – Floating-point MOVe

Format: **FMOV** src, dst

Operation: dst = src

Description: The source operand is moved into the destination with any required conversions performed.

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 6 | 0x11 | **FMOV** | fgen16, | fgen16 |
| 10 | 0x11 | **FMOV** | fgen32, | fgen32 |

Notes: May result in an overflow, underflow, an inexact or an invalid operation. See Section 2.9 for a complete description of floating-point arithmetic.

The **FMOV** instruction is not implemented in hardware. An unimplemented instruction sequence is taken.

Name: FMUL – Floating-point MULtiplication

Format: **FMUL[3]** src, dst

Operation: **FMUL:** dst *= src
**FMUL3:** Acc = dst * src

Description: The product of the source and the destination operands is placed in the destination (**FMUL**) or the floating-point Accumulator (**FMUL3**).

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 6 | 0x29 | **FMUL** | fgen16, | fgen16 |
| 6 | 0x39 | **FMUL3** | fgen16, | fgen16 |
| 10 | 0x29 | **FMUL** | fgen32, | fgen32 |
| 10 | 0x39 | **FMUL3** | fgen32, | fgen32 |

Notes: May result in an overflow, underflow, an inexact or an invalid operation. See Section 2.9 for a complete description of floating-point arithmetic.

The **FMUL[3]** instruction is not implemented in hardware. An unimplemented instruction sequence is taken.

Name: FNEXT – Floating-point NEXT-after

Format: **FNEXT** src, dst

Operation:
```
if ( (src | dst) = = "NaN")
    dst = "NaN"
else
    dst = value adjacent to src in the direction of dst
```

Description: The destination receives the value "adjacent to" the source operand in the direction of the destination operand. "Adjacency" is meant in terms of the format of the source operand.

If either operand is NaN, the result is NaN. Otherwise, for the purposes of **FNEXT**, the floating-point values in a particular format may be taken to be *lexicographically* ordered:

$\{-\infty\} < \{\text{negative numbers}\} < \{-0\} < \{+0\} < \{\text{positive numbers}\} < \{+\infty\}$

Lexicographic order differs from numeric order only in that –0 and +0 are taken to be distinguished neighbors, even though they are equal when compared with any of the variants of **FCMP**.

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 6 | 0x08 | **FNEXT** | fgen16, | fgen16 |
| 10 | 0x08 | **FNEXT** | fgen32, | fgen32 |

Notes: Never results in an exception. See Section 2.9 for a complete description of floating-point arithmetic.

The **FNEXT** instruction is not implemented in hardware. An unimplemented instruction sequence is taken.

Name: FREM – Floating-point REMainder

Format: **FREM** src, dst

Operation:

Given R in the format of dst, and unbounded integers IQ and QQ satisfying:

$$dst = (src * IQ) + R, \text{ with } |R| \leq |src + 2|$$
$$QQ = IQ / 16, \text{ rounded toward 0,}$$

then:

$$dst = R$$
$$FPSW.RQ = IQ - (16 * QQ)$$

Description: The destination operand is given its floating-point remainder modulo the source. The low four bits of the integer quotient, with the quotient's sign, are delivered as a 5-bit two's complement number to the RQ field of the FPSW.

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 6 | 0x0B | **FREM** | fgen16, | fgen16 |
| 10 | 0x0B | **FREM** | fgen32, | fgen32 |

Notes: **FREM** may be thought of as the dividion of *dst* by *src* to produce all (possibly thousands) of the integer quotient bits, which is called IQ in the operation description. To insure that R is no bigger than half of *src*, IQ may be adjusted upward by 1. The resulting remainder, R, is exact. The FPSW RQ-field is the low four bits of the magnitude of IQ, with the sign of IQ, as a two's compliment value. In doing argument reduction for operations such as trigonometric functions, the FPSW RQ-field is of great value when the *src* might be $\pi/4$; the FPSW RQ-field would indicate in which octant the argument, *dst*, lies.

May result in an invalid operation. See Section 2.9 for a complete description of floating-point arithmetic.

The **FREM** instruction is not implemented in hardware. An unimplemented instruction sequence is taken.

Name: FSCALB – Floating-point SCALing by a power of the radix, 2

Format: **FSCALB** src, dst

Operation:

dst *= $2^{src}$

Description: The destination is scaled by 2 raised to the source power. The source, which is an integral, not a floating-point, operand, is truncated to 16-bits before scaling is performed.

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 6 | 0x09 | **FSCALB** | gen16, | fgen16 |
| 10 | 0x09 | **FSCALB** | gen32, | fgen32 |

Notes: May result in an overflow, underflow, an invalid or an inexact operation. See Section 2.9 for a complete description of floating-point arithmetic.

The **FSCALB** instruction is not implemented in hardware. An unimplemented instruction sequence is taken.

Name: FSQRT – Floating-point SQuare RooT

Format: **FSQRT** src, dst

Operation:

$dst = \sqrt{src}$

Description: The destination receives the square root of the source.

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 6 | 0x10 | **FSQRT** | fgen16, | fgen16 |
| 10 | 0x10 | **FSQRT** | fgen32, | fgen32 |

Notes: May result in an inexact or invalid operation. See Section 2.9 for a complete description of floating-point arithmetic.

The **FSQRT** instruction is not implemented in hardware. An unimplemented instruction sequence is taken.

Name: FSUB – Floating-point SUBtraction

Format: **FSUB[3]** src, dst

Operation: **FSUB**: dst –= src
**FSUB3**:Acc = dst – src

Description: The source operand is subtracted from the destination operand and the difference is placed in either the destination (**FSUB**) or the floating-point Accumulator (**FSUB3**).

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 6 | 0x28 | **FSUB** | fgen16, | fgen16 |
| 6 | 0x38 | **FSUB3** | fgen16, | fgen16 |
| 10 | 0x28 | **FSUB** | fgen32, | fgen32 |
| 10 | 0x38 | **FSUB3** | fgen32, | fgen32 |

Notes: May result in overflow, underflow, inexact or an invalid operation. See Section 2.9 for a complete description of floating-point arithmetic.

The **FSUB[3]** instruction is not implemented in hardware. An unimplemented instruction sequence is taken.

Name: JMP – JuMP

Format:
**JMP** dst
**JMPT(Y|N)** dst
**JMPF(Y|N)** dst

Operation:

**JMP:**
PC = &dst
**JMPT:**
if (PSW.F) PC = &dst
**JMPF:**
if (!PSW.F) PC = &dst

Description: The jump instructions cause the address of the destination operand to become the new Program Counter value unconditionally (**JMP**), if the PSW F-bit is one (**JMPT**), or if the PSW F-bit is zero (**JMPF**). A branch prediction bit is available for the conditional jumps to indicate that the jump more likely will (**Y**), or will not (**N**) be taken. Conditional jumps cannot use the indirect addressing modes.

Encodings

| length | opcode | subcode | instruction | src(dst) |
|---|---|---|---|---|
| 2 | 0x03 | - | **JMP** | pcrel10 |
| 2 | 0x04 | - | **JMPFN** | pcrel10 |
| 2 | 0x05 | - | **JMPFY** | pcrel10 |
| 2 | 0x06 | - | **JMPTN** | pcrel10 |
| 2 | 0x07 | - | **JMPTY** | pcrel10 |
| 6 | 0x00 | 0x3 | **JMP** | flow32 |
| 6 | 0x00 | 0x4 | **JMPFN** | abs32 |
| 6 | 0x00 | 0x5 | **JMPFY** | abs32 |
| 6 | 0x00 | 0x6 | **JMPTN** | abs32 |
| 6 | 0x00 | 0x7 | **JMPTY** | abs32 |

Note: If the location pointed to by the jump instruction can not be referenced, a fetch-fault results. In this case, the PC stored on the interrupt stack is the target PC, not the PC of the original jump. The address of the original jump instruction is not saved. In the event of an indirect jump, if the ATT2100 can not reference the indirection word, a read-fault results and the PC stored on the interrupt stack is that of the indirect jump.

Name: KCALL – Kernel CALL

Format: **KCALL** src

Operation:

```
disable interrupts
*(ISP – 12) = src                          /* R4 wrt new ISP */
*(ISP – 8) = PC of next instruction        /* R8 wrt new ISP */
*(ISP – 4) = PSW                           /* R12 wrt new ISP */
ISP –= 16
SHAD = ISP
PC = *(VB + 0)
PSW = PSW & 0xFFFF0000
enable interrupts
```

Description: The Program Status Word, the Program Counter (return point) and the *src* operand values are saved on the Interrupt Stack as a quad-word. The new Program Counter value is read from the memory location pointed to by the Vector Base Register. The low-order 16-bits of the PSW are set to 0, which selects kernel execution level, the ISP as the CSP, disables tracing and inhibits interrupts. The PSW VP- and UA-bits do not change.

Encodings

| length | opcode | subcode | instruction | src |
|---|---|---|---|---|
| 2 | 0x00 | - | **KCALL** | imm10* |
| 6 | 0x00 | 0x0 | **KCALL** | word32 |

Notes: Interrupts are disabled while **KCALL** is processing. If a memory fault occurs while writing to the Interrupt Stack or reading from the table pointed to by the Vector Base, the ATT2100 resets.

If the location pointed to by the **KCALL** PC entry in the vector table can not be referenced, a fetch-fault results. In this case, the PC stored on the interrupt stack is the target PC (the value in the location pointed to by the VB), not the PC of the original **KCALL** instruction.

* The 10-bit immediate value is zero-extended and multiplied by four giving it an effective range of 0 through 4092 in increments of 4.

Name: KRET – Kernel RETurn

Format: **KRET**

Operation:

```
disable interrupts
PC = *(ISP + 8)                 /* R8 wrt ISP */
PSW = *(ISP + 12)               /* R12 wrt ISP */
ISP += 16
if (CSP = = ISP)
        SHAD = ISP
else
        SHAD = SP
enable interrupts
```

Description: The Program Status Word and Program Counter values are restored from the Interrupt Stack.

Encodings

| length | opcode | subcode | instruction |
|---|---|---|---|
| 2 | 0x0B | 0x1 | **KRET** |

Notes: Interrupts are disabled while **KRET** is processing. If a memory fault occurs while reading from the Interrupt Stack the ATT2100 resets.

The **KRET** instruction is privileged. If a **KRET** is executed at the user level, a privilege exception is executed.

The **KRET** instruction can not be traced.

If the location pointed to by the new PC value can not be referenced, a fetch-fault results. In this case, the PC stored on the interrupt stack is the new PC value, not the address of the **KRET** instruction.

Name: LDRAA – LoaD Relative Address into Accumulator

Format: **LDRAA** dst

Operation: ACC = &dst

Description: The destination address is calculated as if a **JMP** instruction were being executed and stored in the Accumulator.

Encodings

| length | opcode | subcode | instruction | src |
|---|---|---|---|---|
| 6 | 0x00 | 0xA | **LDRAA** | flow32 |

Name: MOV – MOVe

Format: **MOV** src, dst

Operation: dst = src

Description: The value of the source operand is stored in the destination.

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 2 | 0x0A | **MOV** | wai5, | stk5 |
| 2 | 0x18 | **MOV** | stk5, | stk5 |
| 2 | 0x19 | **MOV** | istk5, | stk5 |
| 2 | 0x1A | **MOV** | stk5, | istk5 |
| 2 | 0x1B | **MOV** | istk5, | istk5 |
| 2 | 0x1C | **MOV** | imm5, | stk5 |
| 6 | 0x06 | **MOV** | gen16, | gen16 |
| 10 | 0x06 | **MOV** | gen32, | gen32 |

Name: MOVA – MOVe Address

Format: MOVA src, dst

Operation: dst = &src

Description: The address of the source operand is calculated and stored in the destination.

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 2 | 0x1D | MOVA | stk5, | stk5 |
| 6 | 0x04 | MOVA | gen16*, | gen16 |
| 10 | 0x04 | MOVA | gen32*, | gen32 |

Note: If the size of the destination is byte or half-word, the calculated address is truncated (or sign-extended) to 8- or 16-bits. An immediate source operand as well as a register source or destination causes an illegal instruction exception.

* The source operand must use a word addressing mode (i.e., modes ≥ 0xC) except for immediate as already noted. Any other mode causes an illegal instruction exception.

Name: MUL – MULtiply

Format: **MUL**[3] src, dst

Operation:

**MUL:**

dst *= src
"unsigned overflow" ? PSW.C = 1 : PSW.C = 0
"signed overflow" ? PSW.V = 1 : PSW.V = 0

**MUL3:**

Acc = dst * src
"unsigned overflow" ? PSW.C = 1 : PSW.C = 0
"signed overflow" ? PSW.V = 1 : PSW.V = 0

Description: The source operand is multiplied by the destination operand and the product is placed in either the destination (**MUL**) or the Accumulator (**MUL3**). The PSW C-bit is set to 1 if the product of the operands as unsigned values overflows the destination (or Accumulator); similarly, the PSW V-bit is set to 1 if the product of the operands as signed values overflows the destination (or Accumulator); otherwise, the PSW C- and V-bits are set to 0. See Section 2.7 for a full description of ATT2100 integer arithmetic.

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 6 | 0x26 | MUL | gen16, | gen16 |
| 6 | 0x36 | MUL3 | gen16, | gen16 |
| 10 | 0x26 | MUL | gen32, | gen32 |
| 10 | 0x36 | MUL3 | gen32, | gen32 |

Name: NOP – No OPeration

Format: **NOP**

Description: No operation is performed.

Encodings

| length | opcode | subcode | instruction |
|---|---|---|---|
| 2 | 0x0B | 0x2 | **NOP** |

Name: OR – bitwise logical OR

Format: **OR[3]** src, dst

Operation: **OR:** dst |= src
**OR3:** Acc = dst | src

Description: A bitwise logical OR is performed on the source and destination operands and the result is placed in either the destination (**OR**) or the Accumulator (**OR3**).

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 6 | 0x21 | **OR** | gen16, | gen16 |
| 6 | 0x31 | **OR3** | gen16, | gen16 |
| 10 | 0x21 | **OR** | gen32, | gen32 |
| 10 | 0x31 | **OR3** | gen32, | gen32 |

Name: ORI – bitwise logical OR Interlocked

Format: **ORI** src, dst

Operation:
hidden = dst
dst |= src
Acc = hidden

Description: A bitwise logical OR operation is performed on the source and destination operands and the result is placed in the destination. The lock pin is asserted during the fetch of *dst*, if *dst* is in memory and not in the stack cache. The lock pin is de-asserted at the completion of the store to *dst*. No other accesses are done between the fetch and store of *dst*. The original value of *dst* (obtained during the fetch) is placed in the Accumulator. If the Accumulator is not in the stack cache, a store is made after the interlocked I/O completes.

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 6 | 0x01 | **ORI** | gen16, | gen16 |
| 10 | 0x01 | **ORI** | gen32, | gen32 |

Notes: Pipeline bypass hazards associated with semaphore operations are avoided in the ATT2100 by clearing the pipeline before an interlocked instruction enters the first pipeline stage. No other instruction is allowed into the pipeline until the executing interlocked instruction completes.

If R4 is the destination, after the interlocked instruction completes, R4 is the previous value of R4, hence no operation is performed.

If the accumulator is not in the SC, CSP == MSP, an I/O access is made to update the accumulator after the interlocked accesses complete. The access to the accumulator **must** not fault in any manner for the **ORI** is not restartable from this point of the operation.

Name: POPN – Pop N entries from stack cache

Format: **POPN** src

Operation:

```
disable interrupts
SHAD = CSP = CSP + src
if ( (CSP = = SP) && (CSP > MSP) )
        MSP = SP
enable interrupts
```

Description: The src operand is fetched, added to the CSP and SHAD. If the CSP is SP, and the new SP value exceeds the MSP, the MSP is also updated to the new value. If the CSP is ISP, the MSP is not updated.

Encodings

| length | opcode | subcode | instruction | src |
|---|---|---|---|---|
| 2 | 0x02 | 0x3 | **POPN** | stk8* |
| 6 | 0x00 | 0xF | **POPN** | stk32 |

Notes: Only the Stack Offset Addressing mode is legal; any other mode results in an illegal instruction exception sequence. Negative stack offsets are illegal.

* The 8-bit stack offset is zero extended and multiplied by 16 giving it an effective range of 0 through 4080 in quad-aligned increments.

Name: REM – REMainder

Format: **REM[3]** src, dst

Operation: **REM:** dst %= src
**REM3:** Acc = dst % src

Description: The destination operand is divided by the source operand and the remainder is placed in either the destination (**REM**) or the Accumulator (**REM3**). Two's complement division is performed. See Section 2.7 for a full description of ATT2100 integer arithmetic.

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 6 | 0x25 | **REM** | gen16, | gen16 |
| 6 | 0x35 | **REM3** | gen16, | gen16 |
| 10 | 0x25 | **REM** | gen32, | gen32 |
| 10 | 0x35 | **REM3** | gen32, | gen32 |

Notes: Division by zero results in a zero divide exception.

The PSW V-bit is always cleared in the **REM** operation. The C-bit is unchanged in all the cases.

Name: RETURN – RETURN from subroutine

Format: **RETURN** src

Operation:

```
disable interrupts
PC = *(CSP + src)
SHAD = CSP = CSP + src
if( (CSP = = SP) && (CSP > MSP) )
        MSP = SP
enable interrupts
```

Description: The *src* operand is fetched and used as the new Program Counter value. If the CSP is SP, and the new SP value exceeds the MSP, the MSP is also updated to the new value. If the CSP is ISP, the MSP is not updated.

Encodings

| length | opcode | subcode | instruction | src |
|---|---|---|---|---|
| 2 | 0x02 | 0x2 | **RETURN** | stk8* |
| 6 | 0x00 | 0x2 | **RETURN** | stk32 |

Notes: Only the Stack Offset Addressing mode is legal; any other mode results in an illegal instruction exception sequence. Even though the lower four bits of the SP do not exist, **RETURN** can obtain a new PC for a word aligned register offset which is not a multiple of 16, but when adjusting the SP, the lower four bits of the offset is ignored. For example:

**RETURN** R4

obtains the new PC from R4, but the SP does not change. Similarly,

**RETURN** R20

obtains the new PC from R20, but the SP only increments 16.

Only positive offsets are legal. Negative offsets result in an illegal instruction exception sequence.

If the location pointed to by the new PC value can not be referenced, a fetch-fault results. In this case, the PC stored on the interrupt stack is the new PC values, not the address of the **RETURN** instruction.

* The 8-bit stack offset is zero extended and multiplied by 16 giving it an effective range of 0 through 4080 in quad-aligned increments.

Name: SHL – SHift Left

Format: **SHL[3]** src, dst

Operation: **SHL**: dst <<= Unsigned(src)
**SHL3**: Acc = dst << Unsigned(src)

Description: The destination operand is shifted left by the number of bits indicated by the source operand. Zeroes replace the bits shifted out of the least-significant-bit of *dst*. Only the low-order five bits of *src* are used for the shift amount. The upper-bits are ignored.

For **SHL3**, the result is placed in the Accumulator and the destination is left unchanged.

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 2 | 0x1E | **SHL3** | uimm5, | stk5 |
| 6 | 0x2E | **SHL** | gen16, | gen16 |
| 6 | 0x3E | **SHL3** | gen16, | gen16 |
| 10 | 0x2E | **SHL** | gen32, | gen32 |
| 10 | 0x3E | **SHL3** | gen32, | gen32 |

Name: SHR – arithmetic SHift Right

Format: **SHR**[3] src, dst

Operation: **SHR:** dst >>= src
**SHR3:** Acc = dst >> src

Description: The destination operand is shifted right by the number of bits indicated by the source operand. The sign-bit of the destination is copied as bits are shifted rightward. Only the low-order five bits of *src* are used for the shift amount. The upper-bits are ignored.

For **SHR3**, the result is placed in the Accumulator and the destination is left unchanged.

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 2 | 0x1F | **SHR3** | uimm5, | stk5 |
| 6 | 0x2C | **SHR** | gen16, | gen16 |
| 6 | 0x3C | **SHR3** | gen16, | gen16 |
| 10 | 0x2C | **SHR** | gen32, | gen32 |
| 10 | 0x3C | **SHR3** | gen32, | gen32 |

Name: SUB – SUBtract

Format: **SUB[3]** src, dst

Operation:

**SUB:**

dst –= src
"unsigned borrow" ? PSW.C = 1 : PSW.C = 0
"signed borrow" ? PSW.V = 1 : PSW.V = 0

**SUB3:**

Acc = dst – src
"unsigned borrow" ? PSW.C = 1 : PSW.C = 0
"signed borrow" ? PSW.V = 1 : PSW.V = 0

Description: The source operand is subtracted from the destination operand and the difference is placed in either the destination (**SUB**), or the Accumulator (**SUB3**). The PSW C-bit is set on unsigned overflow and the PSW V-bit is set on signed overflow, otherwise the PSW C- and V-bits are set to 0. See Section 2.7 for a full description of ATT2100 integer arithmetic.

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 6 | 0x20 | **SUB** | gen16, | gen16 |
| 6 | 0x30 | **SUB3** | gen16, | gen16 |
| 10 | 0x20 | **SUB** | gen32, | gen32 |
| 10 | 0x30 | **SUB3** | gen32, | gen32 |

Name: TADD – Tagged ADDition

Format: **TADD** src, dst

Operation:

```
if ((src[1:0] != 0x0) || (dst[1:0] != 0x0)
        PSW.F = 1
else {
        dst + src
        "unsigned overflow" ? PSW.C = 1 : PSW.C = 0
        "signed overflow" ? PSW.V = 1 : PSW.V = 0
        PSW.F = PSW.V
}
if (PSW.F == 0)
        dst = dst + src
```

Description: The source operand is added to the destination operand and the sum is placed in the destination if the PSW F-bit is set to 0. The PSW F-bit is set to 1 if the low two bits of either the source and destination operands are non-zero or the PSW V-bit was set to 1. The PSW C-bit is set to 1 on unsigned overflow and the PSW V-bit is set to 1 on signed overflow, otherwise the PSW C- and V-bits are set to 0. See Section 2.7 for a full description of ATT2100 integer arithmetic.

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 6 | 0x0C | **TADD** | gen16, | gen16 |
| 10 | 0x0C | **TADD** | gen32, | gen32 |

Name: TESTC – TEST psw Carry

Format: **TESTC**

Operation: PSW.F = PSW.C
PSW.C = 0

Description: The PSW C-bit is copied into the PSW F-bit and the PSW C-bit is set to 0.

Encodings

| length | opcode | subcode | instruction |
|---|---|---|---|
| 2 | 0x0B | 0x9 | **TESTC** |

Name: TESTV – TEST psw oVerflow

Format: **TESTV**

Operation: PSW.F = PSW.V
PSW.V = 0

Description: The PSW V-bit is copied into the PSW F-bit and the PSW V-bit is set to 0.

Encodings

| length | opcode | subcode | instruction |
|---|---|---|---|
| 2 | 0x0B | 0x8 | **TESTV** |

Name: TSUB – Tagged SUBtraction

Format: **TSUB** src, dst

Operation:

```
if ((src[1:0] != 0x0) || (dst[1:0] != 0x0)
        PSW.F = 1
else {
        dst – src
        "unsigned borrow" ? PSW.C = 1 : PSW.C = 0
        "signed borrow" ? PSW.V = 1 : PSW.V = 0
        PSW.F = PSW.V
}
if (PSW.F == 0)
        dst = dst – src
```

Description: The source operand is subtracted from the destination operand and the difference is placed in the destination if the PSW F-bit is set to 0. The PSW F-bit is set to 1 if the low two bits of either the source and destination operands are non-zero or the V-bit was set to 1. The PSW C-bit is set to 1 on unsigned overflow and the PSW V-bit is set to 1 on signed overflow, otherwise the PSW C- and V-bits are set to 0. See Section 2.7 for a full description of ATT2100 integer arithmetic.

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 6 | 0x0D | **TSUB** | gen16, | gen16 |
| 10 | 0x0D | **TSUB** | gen32, | gen32 |

Name: UDIV – Unsigned DIVide

Format: **UDIV** src, dst

Operation: dst += src

Description: The destination operand is divided by the source operand and the quotient is placed in the destination. Unsigned division is performed. See Section 2.7 for a full description of ATT2100 integer arithmetic.

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 6 | 0x2F | **UDIV** | gen16, | gen16 |
| 10 | 0x2F | **UDIV** | gen32, | gen32 |

Notes: Division by zero results in a zero divide exception.

The PSW C-bit is always cleared in the **UDIV** operation. The V-bit is unchanged in all the cases.

Name: UREM – Unsigned REMainder

Format: **UREM** src, dst

Operation: dst %= src

Description: The destination operand is divided by the source operand and the remainder is placed in the destination. Unsigned division is performed. See Section 2.7 for a full description of ATT2100 integer arithmetic.

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 6 | 0x05 | **UREM** | gen16, | gen16 |
| 10 | 0x05 | **UREM** | gen32, | gen32 |

Note: Division by zero results in a zero divide exception.

The PSW C-bit is always cleared in the **UDIV** operation. The V-bit is unchanged in all the cases.

Name: USHR – Unsigned SHift Right

Format: **USHR**[3] src, dst

Operation: **USHR:** dst >>= Unsigned(src)
**USHR3:** Acc = dst >> Unsigned(src)

Description: The destination operand is shifted right by the number of bits indicated by the source operand. Zeroes replace the bits shifted out of the most-significant-bit of destination operand. Only the low five bits of the source operand are used for the shift amount. The upper-bits are ignored.

For **USHR3**, the result is placed in the Accumulator and the destination is left unchanged.

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 6 | 0x2D | **USHR** | gen16, | gen16 |
| 6 | 0x3D | **USHR3** | gen16, | gen16 |
| 10 | 0x2D | **USHR** | gen32, | gen32 |
| 10 | 0x3D | **USHR3** | gen32, | gen32 |

Name: XOR – bitwise logical eXclusive OR

Format: **XOR[3]** src, dst

Operation: **XOR:** dst ^= src
**XOR3:** Acc = dst ^ src

Description: A bitwise logical exclusive OR operation is performed on the source and destination operands and the result is placed in either the destination (**XOR**) or the Accumulator (**XOR3**).

Encodings

| length | opcode | instruction | src | dst |
|---|---|---|---|---|
| 6 | 0x24 | **XOR** | gen16, | gen16 |
| 6 | 0x34 | **XOR3** | gen16, | gen16 |
| 10 | 0x24 | **XOR** | gen32, | gen32 |
| 10 | 0x34 | **XOR3** | gen32, | gen32 |

### 2.15 Pipeline Considerations

Because of the pipelining within the ATT2100 microprocessor, certain combinations of instructions may have unexpected results. Some of these cases were noted in the previous section. These cases include:

1. There must be at least two instructions between instructions that might set the Carry and oVerflow bits (such as **ADD** or **MUL**) and an instruction that explicitly reads the PSW, using the **CPU** prefix. The intervening instructions are not necessary if the Carry and Overflow bits are queried with the **TESTC** or **TESTV** instructions.
2. An **ENTER** cannot immediately follow the invalidation of the page into which it enters. There should be two instructions between the invalidation of the page and the **ENTER** to allow memory table to be updated.
3. If a **ADD**, **SHL** or **MUL** instruction with a destination size of byte or half-word results in a number which overflows the destination size, but can fit in a 32-bit word, a subsequent instruction may use the 32-bit version of the result, rather than a truncated 8- or 16-bit result. The non-truncated result may affect the computation if the **MUL**, **USHR** or **ADD** overflows its byte or half-word destination and
   a. The following instruction is a divide or a right shift and it uses the destination of the first instruction as one of its operands, or
   b. The destination of the second instruction is larger than the destination of the first instruction.

   The use of the truncated version of the result can be forced by interposing two instructions between the **MUL**, **SHL** or **ADD** and the following instruction.

   Examples:

```
MUL   $0x7F,R4:B      MUL   $0x7F,R4:B
USHR  $4,R4:B →       instr
                      instr
                      USHR  $4,R4:B

MUL   $0x7F,R4:B      MUL   $0x7F,R4:B
MOV   R4:B,R8:L →     instr
                      instr
                      MOV   R4:B,R8:L
```

4. An instruction which reads the SHAD register cannot be executed immediately after an **ENTER** or **RETURN** instruction. Two NOPs should be placed between such instructions to permit the writing of the SHAD.

Examples:

```
ENTER   R-16                ENTER   R-16
MOV     $new,%SHAD →        NOP
                            NOP
                            MOV     $new,%SHAD

CALL    routine             CALL    routine
ADD     $16,%SHAD →         NOP
                            NOP
                            ADD     $16,%SHAD
```

## 3. PERFORMANCE

This section contains performance data on an instruction basis.

### 3.1 Instruction Execution Times

Because of the highly pipelined nature of the ATT2100 microprocessor, it is difficult to determine how long it takes to execute any single instruction. Many instructions can be executed at the rate of one per cycle, because pipelining allows the execution of instructions to be overlapped. In order to describe the time it takes to execute an instruction, execution times will be specified assuming the following conditions exist:

- instruction fetches must hit in the instruction cache,
- only the stack offset, immediate, register, absolute (for jumps and calls only), or program counter relative addressing modes are used,
- all stack offset accesses are captured in the stack cache,
- and no data hazards occur between instructions.

In addition to the instruction execution time, there are a number of pipeline delays that cause instructions to take longer to execute. These delays are listed in supplementary tables and should be added to the base execution time if they exist.

The following abbreviations will be used in this section:

**TABLE 3-1.** Performance Abbreviations

| Abbreviation | Meaning |
|---|---|
| IC | decoded instruction cache |
| PDU | prefetch and decode unit |
| A | memory access time for a single word access |
| D | memory access time for a double word access |
| Q | memory access time for a quad word access |
| M | memory |
| N | the number of valid entries in the stack cache |
| SC | stack cache |
| E | **ENTER** size |

Instructions are grouped into five types for the purposes of estimating their execution time. These types are presented in the subsequent tables. Delays are also grouped into four types which are presented at the end of this section.

Delays for simple instructions are given in Table 3-2. For such instructions, instruction fetch and operand access delays, given in Tables 3-7 and 3-8, are possible.

**TABLE 3-2.** Simple Instruction Execution Times

| Instruction | Min | Max | Instruction | Min | Max |
|---|---|---|---|---|---|
| **ADD** | 1 | 1 | **OR** | 1 | 1 |
| **ADD3** | 1 | 1 | **OR3** | 1 | 1 |
| **ADDI** | 2 | 2 | **ORI** | 2 | 2 |
| **AND** | 1 | 1 | **RETURN*** | 2 | 2 |
| **AND3** | 1 | 1 | **POPN** | 2 | 2 |
| **ANDI** | 2 | 2 | **SHL** | 1 | 1 |
| **CALL** | 1 | 1 | **SHL3** | 1 | 1 |
| **CLRE** | 1 | 1 | **SHR** | 1 | 1 |
| **CMP** | 1 | 1 | **SHR3** | 1 | 1 |
| **CPU** | 0 | 0 | **SUB** | 1 | 1 |
| **FLUSHI** | 1 | 1 | **SUB3** | 1 | 1 |
| **FLUSHP** | 1 | 1 | **TADD** | 1 | 1 |
| **FLUSHPBE** | 1 | 1 | **TESTC** | 1 | 1 |
| **FLUSHPTE** | 1 | 1 | **TESTV** | 1 | 1 |
| **JMP**** | 0 | 1 | **TSUB** | 1 | 1 |
| **LDRAA** | 1 | 1 | **USHR** | 1 | 1 |
| **MOV** | 1 | 1 | **USHR3** | 1 | 1 |
| **MOVA** | 1 | 1 | **XOR** | 1 | 1 |
| **NOP** | 1 | 1 | **XOR3** | 1 | 1 |

* Do not add delay for indirection since RETURNs are always indirect.

** If an unconditional jump is folded into the previous instruction, it takes no time to execute, otherwise it takes one cycle.

The execution times for multi-cycle arithmetic instructions given in Table 3-3 are data dependent. For these instructions, instruction fetch, operand access, and data type delays,[1] given in Tables 3-7, 3-8, and 3-9, are possible.

1. Data type delays only occur for signed multi-cycle instructions.

**TABLE 3-3.** Multi-Cycle Arithmetic Instruction Execution Times

| Instruction | Min | Max |
|---|---|---|
| **DIV** | 38 | 38 |
| **DIV3** | 38 | 38 |
| **MUL** | 3 | 20 |
| **MUL3** | 3 | 20 |
| **REM** | 38 | 38 |
| **REM3** | 38 | 38 |
| **UDIV** | 38 | 38 |
| **UREM** | 38 | 38 |

The execution times for DQM are given in Table 3-4. For DQM, instruction fetch delays, given in Tables 3-7, are possible.

**TABLE 3-4.** DQM Instruction Execution Times

| Type of DQM | Cycles |
|---|---|
| Constant to SC double-word | 3 |
| Constant to SC quad-word | 5 |
| Constant to M double-word | 1 + D |
| Constant to M quad-word | 1 + Q |
| SC double-word to SC double-word | 4 |
| SC quad-word to SC quad-word | 8 |
| SC/M double-word to M/SC double-word | 2 + D |
| SC/M quad-word to M/SC quad-word | 4 + Q |
| M double-word to M double-word | 2 * D |
| M quad-word to M quad-word | 8 * S |

The delays for the remaining miscellaneous instructions are given in Table 3-5. For these instructions, instruction fetch and miscellaneous delays, given in Tables 3-7 and 3-10, are possible.

**TABLE 3-5.** Miscellaneous Instruction Execution Times

| Instruction | Cycles |
|---|---|
| **CATCH** | 1 |
| **CRET** | 11 + Q |
| **ENTER** | 1 + (Q * E) |
| **KCALL** | 8 + D + A |
| **KRET** | 10 + D |
| unimplemented opcode | 7 + A |
| exception | 9 + D + A |

The delays associated with conditional jump instructions are given in Table 3-6. For this class of instruction the fetch delays given in Table 3-7 are possible if the branch is not folded.

**TABLE 3-6.** Conditional Jump Instruction Execution Times

| Instruction | Cycles |
|---|---|
| correct prediction, folded | 0 |
| correct prediction, unfolded | 1 |
| incorrect prediction, jump after compare | 3 |
| incorrect prediction, jump 2 instructions after compare | 2 |
| incorrect prediction, jump 3 instructions after compare | 1 |
| incorrect prediction, unfolded, 4 or more instructions after compare | 1 |
| incorrect prediction, folded, 4 or more instructions after compare | 0 |

Instruction fetch delays occur when the instruction is not immediately available for execution by the EU. The instruction misses the IC and the EU resets the PDU to fetch the desired instruction. These delays are best case. If the EU is using I/O, the PDU is doing an unrelated memory access at the time of reset, or the PDU is handling a previously received fault, the delays will be longer.

Operand accesses may also take longer than can be predicted using these tables because of the possibility of a data hazard or internal contention for I/O. Data hazards occur when a previous instructions tries to write to a memory location which overlaps the location being read by a subsequent instruction. I/O contention occurs when the EU wants to make an external memory access while the PDU is in the middle of an access. These delays also cannot be accurately predicted.

**TABLE 3-7.** Instruction Fetch and Empty Pipeline Delays

| Condition | Penalty |
|---|---|
| IC miss, instruction contained in prefetch buffer | 3 |
| IC miss, instruction contained in single double word in memory | 5 + D |
| IC miss, instruction contained in single quad word in memory | 5 + 2D |
| IC miss, instruction contained in 2 quad words in memory | 8 + 2D |
| IC miss and instruction is a CPU-prefix operation | 1 |
| EU pipeline empty | 2 |

**TABLE 3-8.** Operand Access Delays

| Condition | Penalty |
|---|---|
| one operand in memory | 1 + A |
| two operands in memory | 1 + 2A |
| one or two operands indirect, both pointers in stack cache | 1 |
| one or two operands indirect, one pointer in memory | 1 + A |
| two operands indirect, both pointers in memory | 1 + 2A |
| destination in memory | A |

**TABLE 3-9.** Data Type Delays

| Condition | Penalty |
|---|---|
| One operand not word type | 1 |
| Two operands not word type | 2 |

**TABLE 3-10.** Miscellaneous Delays

| Condition | Penalty |
|---|---|
| one or two operands indirect, both pointers in stack cache | 1 |
| one or two operands indirect, one pointer in memory | 1 + A |
| two operands indirect, both pointers in memory | 1 + 2A |

### 3.2 Branch Folding

The ATT2100 provides a next address field with each decoded instruction. When the PDU detects a non-branching operation followed by a branch, it "folds" the two instructions to form a single instruction/branch operation. As a result, branches are rarely explicitly executed since they are folded and executed along with other instructions. A one-parcel branch will be folded into a previous one- or three-parcel instruction and executed together except when the previous instruction is one of the following:

1. another jump of any kind,
2. any one-parcel instruction with an opcode (five-bit) in the range 00000→ 00111, or
3. any three-parcel monadic instruction, i.e., opcode equals 000000.

## 4. ELECTRICAL INTERFACE

This section specifies input, clock, and output voltage and current operating levels. Unless otherwise stated, all voltage level specifications are referenced to $V_{SS}$ (ground input).

### 4.1 Input Protection

Specification of the input protection capability and testing technique will be given in Release 2.0.

### 4.2 Pin Electrical Specifications

All inputs, clocks, outputs and input/outputs from the ATT2100 are CMOS compatible, except for the test inputs which have pull-up or pull-down devices for termination when left unconnected. Table 4.1 contains the specifications of voltage levels, drive current, and leakage current.

**TABLE 4-1.** Pin Electrical Specifications

| Parameter | Symbol | Min | Nom | Max | Unit |
|---|---|---|---|---|---|
| Supply Voltage 3.3V Operation | $V_{DD}$ | 3.135 | 3.3 | 3.465 | V (DC) |
| Supply Voltage 5.0V Operation | $V_{DD}$ | 4.75 | 5.0 | 5.25 | V (DC) |
| Input High Voltage | $V_{IH}$ | $V_{DD}$–0.5 | $V_{DD}$ | $V_{DD}$+0.5 | V (DC) |
| Input Low Voltage | $V_{IL}$ | -0.5 | 0 | +0.5 | V (DC) |
| TDI Input Low Current | $I_{TDI}$ | - | - | –(0.36+($V_{DD}$–3)0.32) | mA |
| TMS Input Low Current | $I_{TMS}$ | - | - | –(0.16+($V_{DD}$–3)0.16) | mA |
| TCK Input Low Current | $I_{TCK}$ | - | - | –(0.18+($V_{DD}$–3)0.16) | mA |
| TRST- Input High Current | $I_{TRST}$ | - | - | –(0.32+($V_{DD}$–3)0.22) | mA |
| Input Leakage Current | $I_I$ | -1 | - | 1 | µA |
| Output High Voltage | $V_{OH}$ | $V_{DD}$–0.25 | $V_{DD}$ | - | V (DC) |
| Output Low Voltage | $V_{OL}$ | - | 0 | 0.25 | V (DC) |
| Output High Current | $I_{OH}$ | -1 | - | - | mA |
| Output Low Current | $I_{OL}$ | - | - | 1 | mA |
| Tri-stated Output Leakage Current | $I_{OTI}$ | -1 | - | 1 | µA |
| 20 MHz Supply Current at 3.465V | $I_{DD}$ | - | 150 | 175 | mA |
| Standby Current | $I_{SB}$ | 0 | - | 40 | µA |

### 4.3 Absolute Maximum Rating

The Table 4.2 gives the absolute maximum ratings for the ATT2100.

**TABLE 4-2.** Absolute Maximum Ratings

| Type | Symbol | Min | Max | Unit |
|---|---|---|---|---|
| Supply Voltage | $V_{DD}$–$V_{SS}$ | | ≤ 7 | V (DC) |
| Ambient Operating Temperature Range | $T_A$ | 0 | +70 | °C |
| Junction Operating Temperature | $T_J$ | 0 | +125 | °C |
| Storage Temperature Range | $T_{STG}$ | -40 | +125 | °C |

# 5. PIN-OUT AND PROTOCOL

## 5.1 Summary of Pin-Out and Protocol Features

The ATT2100 interface is designed to provide a high performance data transfer mechanism. Salient features of the interface are as follows:

- One clock period synchronous bus transactions.
- Synchronous wait state insertion.
- Double-word/Quad-word "Block Transfer" capability.
- Read-modify-write interlocked bus transactions.
- Six levels of maskable interrupts and one non-maskable interrupt.
- Fast external bus arbitration.
- Byte marks for sub-word access.
- "High Priority" bus arbitration through retry.
- IEEE 1149.1/D5 Test Access Port Compatible.
- Low-power stand-by mode.

There are 93 active signal pins plus 19 power and 20 ground signals (see Table 5-1) accounting for a total of 132 pins.[1]

In Section 5 the following notation has been used to describe signal pins:

— Any signal name suffixed with a minus sign is designated active low, while signals that are not suffixed with a minus sign are designated active high.

— Multi-bit field signals are described by the notation, NAME<msb:lsb>. For example, A<31:02> defines 30 signals whose designations are A02 through A31.

## 5.2 Pin-Out

The 93 active signal pins on the ATT2100 can be divided into several logical groups. Each of these have been individually described here.

### *5.2.1 Clock Group*

Two 1X clocks in quadrature are required by the ATT2100. The internal clocks are decoded from these inputs. The internal clocks can be stopped in phase 1 by a synchronous input allowing for burst-mode and single-stepping operation. See Figure 9-1 for clock timing requirements.

| | |
|---|---|
| CK23 | Phase 23 Clock. Input. The primary clock high during phases 2 and 3. |
| CK34 | Phase 34 Clock. Input. The primary clock high during phases 3 and 4. |
| STOP- | Stop Clocks. Input. STOP- is used to stop the ATT2100 master clock decoder in phase 1. This input is asserted a setup time prior to phase 1 to halt the ATT2100. |

1. TAB packaging is being investigated. As well, a higher pin count package may be required to satisfy switching noise requirements. The additional pins will be allocated to power and ground.

**TABLE 5-1.** ATT2100 Pin Designations

| Function | Type | Name | Description | Pins |
|---|---|---|---|---|
| Start Cycle | O | STC-* | Start of Cycle Strobe | 1 |
| Write/Read- | O | W/R-* | Write or Read Transfer Strobe | 1 |
| Not Cache | O | NCACHE-* | Address may not be Cached | 1 |
| IO Count | O | IOCOUNT<1:0>* | Block Transfer I/O Remaining | 2 |
| Data/Text- | O | D/T-* | Data or Text Bus Transaction | 1 |
| Bus Lock | O | LOCK-* | Multiple Transfer Bus Lock | 1 |
| Address Bus | O | A<31:02>* | Address Bus | 30 |
| Byte Marks | O | BM<3:0>-* | Bytes Active During Trans. | 4 |
| Data Bus | I/O | D<31:00>* | Bi-directional Data Bus | 32 |
| Data Ack. | I | DTACK-** | Data Transfer Acknowledge | 1 |
| Bus Error | I | BERR-** | Transaction Error Signal | 1 |
| Hold | I | HOLD-* | Bus Transaction Hold Input | 1 |
| Retry | I | RETRY-* | Bus Transaction Retry Input | 1 |
| Bus Grant | I | BGRANT- | Bus Grant Input from Bus Arbiter | 1 |
| Bus Request | O | BREQ- | ATT2100 Bus Request to Bus Arbiter | 1 |
| Bus Grant Ack. | O | BGACK- | Bus Grant Acknowledge | 1 |
| Interrupt Req. | I | IL<2:0> | Interrupt Level Inputs | 3 |
| Clock Stop | I | STOP- | ATT2100 Stop Input | 1 |
| Data Tri-state | I | DTRI- | Data Tri-state | 1 |
| Reset Signal | I | RESET- | Reset Input to ATT2100 | 1 |
| Clock Input | I | CK23 | Phase 23 Clock Input | 1 |
| Clock Input | I | CK34 | Phase 34 Clock Input | 1 |
| Test Clock | I | TCK | Test Clock Input | 1 |
| Test Input | I | TDI | Test Data Input | 1 |
| Test Mode | I | TMS | Test Mode Select Input | 1 |
| Test Reset | I | TRST- | Test Reset Input | 1 |
| Test Out | O | TDO | Test Data Output | 1 |
| Power | P | $V_{DD}$<18:00> | Power Pins | 19 |
| Ground | G | $V_{SS}$<19:00> | Ground Pins | 20 |

* Tri-stated when RESET- asserted or bus not owned after de-assertion of BGRANT-.

** Masked when RESET- asserted, bus not owned after de-assertion of BGRANT- or RETRY- is asserted.

### *5.2.2 Bus Arbitration Group*

To facilitate multiple bus masters, the bus arbitration protocol does not make the ATT2100 microprocessor default master. A centralized arbiter selects the current bus master.

BGRANT- is used to grant exclusive use of the bus. In a multiple bus master system, only one BGRANT- is to be asserted at any time to avoid bus contention. See Section 5.4.4 for a complete description of the bus arbitration protocol.

The following signals are used for granting and releasing the bus.

BREQ- Bus Request. Output. BREQ- is asserted when the ATT2100 has a valid I/O transaction pending. BREQ- is de-asserted when the ATT2100 has no pending I/O transactions. When BREQ- is de-asserted, I/O may be in progress on the pins, but there are no transactions following the bus transaction in progress.

BGRANT- Bus Grant. Input. The bus arbiter asserts BGRANT- to the ATT2100 indicating it is bus master for the next bus transaction. While BGRANT- is asserted, the ATT2100 remains bus master.

BGRANT- is asserted or de-asserted by the arbiter a set-up time prior to the rising edge of CK23.

While BGRANT- is asserted the ATT2100 remains in control of the bus. The arbiter de-asserts BGRANT- to tell the ATT2100 to get off the bus after it completes the current bus transaction.

BGACK- Bus Grant Acknowledge. Output. BGACK- is asserted by the ATT2100 to indicate ownership of the bus and de-asserted to indicate that the ATT2100 no longer owns the bus. BGACK- is provided for systems which can not follow bus transactions to determine when the ATT2100 is finished with the current transaction.

*5.2.3 Exception Handling Group*

The exception handling group of signals provides a means by which external devices can inform the ATT2100 of an unusual condition which requires the ATT2100 to deviate from its normal execution.

See Section 2.13.3 for a description of exception processing.

RESET- Reset Signal. Asynchronous Input. The ATT2100 can be reset by asserting this signal for at least two consecutive clock cycles. RESET- is internally double sampled.

For multiple masters, RESET- should be synchronous to insure proper initialization. In this mode, de-assertion of RESET- should be a set-up time prior to the rising edge of CK34.

BERR- Bus Error. Input. The assertion of BERR- indicates an errant a bus transaction. BERR- is used to signal a transaction error for any type of bus transaction. An internal I/O fault is generate when BERR- is asserted and a DTACK- is received.

When BERR- is asserted and DTACK- received, the exception taken depends upon the type of bus transaction being terminated. See Section 2.13.3 for details.

BERR- is asserted and de-asserted by the slave device a set-up time prior to the rising edge of CK34.

HOLD- Hold IO State. Input. HOLD- is asserted to hold up any further I/O transactions by the ATT2100.

HOLD- is asserted or de-asserted a set-up time prior to the rising edge of CK23.

Once HOLD- is de-asserted, bus transactions are allowed to start once the ATT2100 obtains ownership of the bus as HOLD- is orthogonal to bus arbitration.

There are several applications in which the hold feature is necessary. For example, in systems with slow tri-stating devices assertion of HOLD- may be necessary to allow the device time to get off the bus after DTACK-, described in Section 5.2.4.1, has been returned.

RETRY- Retry Bus Transaction. Input. RETRY- is asserted to retry the current bus transaction.

RETRY- is asserted or de-asserted a set-up time prior to the rising edge of CK23.

When RETRY- is asserted during a valid bus transaction, the ATT2100 aborts the current bus transfer and masks the DTACK- input.

Once RETRY- is de-asserted, the bus transaction is rerun after the ATT2100 obtains ownership of the bus as RETRY- is orthogonal to bus arbitration.

There are several applications in which the retry feature is necessary. For example, in systems with gateways through which two busses communicate with each other the retry feature is required to break deadlock conditions when the two busses have simultaneous requests for their respective counterpart bus.

*5.2.3.1 Priorities Of Exception Handling Pins*

A bus transfer may be ended in the normal case by DTACK- or in case of an exception by RESET-.

If two or more of these occur simultaneously the ATT2100 uses the priority scheme shown in Table 5-2. If RETRY- is asserted, DTACK- is masked, hence RETRY- has a higher priority than DTACK- even though it does not end the bus transfer, but does abort the transaction which will be rerun after de-assertion of RETRY-.

**TABLE 5-2.** Priorities of Bus Transaction Termination Signals

| Signal | Priority Level |
|---|---|
| RESET- | HIGHEST |
| RETRY- | |
| DTACK- | LOWEST |

*5.2.4 Transfer Group*

This group of signals is used for addressing devices and transferring text and data. These signals can be further divided into three sub-groups.

*5.2.4.1 Hand-shake Signals*

The hand-shake signals control bus transaction hand-shaking.

STC- Start Cycle. Output. Start cycle strobe is asserted by the master having BGRANT- to indicate start of a bus transaction. STC- is asserted for only one clock cycle at the beginning of the bus transaction.

DTACK- Data Transfer Acknowledge. Input. DTACK- is used to handshake between the ATT2100 and the slave devices. During a normal bus transfer this signal is used to terminate the transaction [data latched during read transaction, withdrawn during write transaction]

DTACK- is asserted and de-asserted by the slave device a set-up time prior to the rising edge of CK34.

*5.2.4.2 Address and Data Signals*

This group of signals indicate the address and data.

A<31:02> Address lines. Output. This 30-bit address bus indicates word-aligned physical addresses. The byte mark signals, BM<3:0>- described in Section 5.2.4.3, are used for sub-word accesses.

D<31:00> Data lines. Bi-directional. This 32-bit data bus conveys data to and from the ATT2100. On byte writes, the active byte is indicated by the BM<3:0>- signals with that byte replicated on the other inactive bytes. On half-word writes, the active half-word is indicated by the BM<3:0>- signals with that half-word replicated on the inactive half-word.

Looping-back of the data bus is supported. After completion of a read transaction, if the current bus master retains ownership of the bus and there are no other transactions pending, the data just read by the ATT2100 is looped-back onto the data bus.

*5.2.4.3 Transfer Qualifier Signals*

This group of signals identify the type of transfer.

LOCK- Bus Lock. Output. This signal is asserted to identify interlocked operations.

The instruction set allows the ATT2100 to run interlocked operations for communication and message passing in multiprocessor system. As well, the MMU asserts LOCK- during miss-processing (see Section 6.5). Interlocked transfers in the ATT2100 are of the read-modify-write (RMW) type although MMU miss-processing may abort the interlocked operation before the write starts. LOCK- remains asserted through the write access.

Once the ATT2100 begins an interlocked operation, loss of bus ownership must not occur until LOCK- is de-asserted. Effectively, a dead cycle is inserted after a RMW operation.

Interlocked transfers are not interruptible.

Interlocked transfers may not be retryed after the read completes. It is up to the system to enforce this restriction. If RETRY- is asserted any time during an interlocked transfer, the retry is honored. It is illegal to assert RETRY- after the read portion of the transfer since it causes the ATT2100 to abort the operation and become susceptible to bus arbitration, thus breaking the lock on the bus.

Bus error can be asserted in either the Read or Write portion of the interlocked transfer. The interlocked operation is faulted with the appropriate exception sequence executed.

If the operands being read by the interlocked instruction are in the stack cache, the lock signal is not asserted. See Section 2.14 for descriptions of the interlocked instructions.

NCACHE- None Cache Transaction. Output. The NCACHE- output is provided for use with external caches to indicate an address may not be cached. When the PSW VP-bit is 1, the MMU uses the NCACHE- output to indicate the status of the $-bit in various entries. See Sections 6.2 and 6.3 for details. When the PSW VP-bit is 0, NCACHE- is asserted.

IOCOUNT<1:0> I/O Transaction Count. Output. These signals indicate the number of words remaining to be transferred.

IOCOUNT<1:0> is used to determine the size of a block transfer being performed by the ATT2100. These block transfers look like a series of bus transfers with STC- asserted for each and the new address provided by incrementing the lower address bits for each word transfer.

Once the ATT2100 begins a block transfer operation, loss of bus ownership must not occur until the block transfer is completed.

The block transfer is not interruptible.

W/R- Write/Read. Output. This signal indicates whether a read or write bus transaction is taking place. It is asserted (high for write, low for read) at the beginning of each bus transfer and is valid for the entire length of the transaction.

D/T- Data/Text. Output. This signal indicates whether data or text is being accessed. It is asserted (high for data, low for text) at the beginning of each bus transfer and is valid for the entire length of the transaction.

BM<3:0>- Byte Marks. Output. These signals indicate which bytes are valid during a data transfer, which may be either a read or a write (See Table 5-3). The bus is capable of doing 8-, 16-, 24- or 32-bit data transfers (although the instruction set uses only 8-, 16- or 32-bit data transfers).

Combinations of the byte mark strobes are used to accomplish the desired word or sub-word transfer.

Either little-endian or big-endian byte encoding may be selected for data via the PSW UL-bit or the CONFIG KL-bit for the user or kernel, respectively. Text is always big-endian encoding.

**TABLE 5-3.** Byte Mark Strobe Encoding

| Pin Name | Bits Active |
|---|---|
| BM0- | D<31:24> |
| BM1- | D<23:16> |
| BM2- | D<15:8> |
| BM3- | D<7:0> |

### *5.2.5 Interrupt Handling Group*

This group of signals controls interrupting of the ATT2100.

IL<2:0> Interrupt Level. Inputs. The ATT2100 recognizes six levels of interrupts encoded onto these lines. These lines are intended to be connected to the outputs of an 8-to-3 priority encoder. The Table 5-4 gives the interrupt level encoding.

When a valid interrupt is recognized, the ATT2100 requests ownership of the bus if it is not bus master. After becoming the bus master, the ATT2100 services the interrupt after aborting or completing the current instruction, depending on the type of instruction being executed.

The internal latching of the interrupt is not predictable; it is necessary that the interrupting device maintain its interrupt assertion until it is serviced, see Sections 2.13.2 and 5.5.3 for more detail.

An external Interrupt Controller is required to resolve conflicts between simultaneously occurring interrupts.

**TABLE 5-4.** Interrupt Levels

| IL<2:0> | Interrupt Level |
|---|---|
| 000 | nmi |
| 001 | Level 1 |
| 010 | Level 2 |
| 011 | Level 3 |
| 100 | Level 4 |
| 101 | Level 5 |
| 110 | Level 6 |
| 111 | No interrupt |

### *5.2.6 Test Pins*

The ATT2100 microrocessor provides boundary scan and extensive built-in-test (BIT) features accessed through an interface specified by the IEEE 1149.1/D5. See Section 10 for details on the implementation.

The signals required to provide this interface are described in this section for completeness.

TCK — Test Clock. Input. An externally gated clock signal with a 50% duty cycle. The changes on the TAP input signals (TMS and TDI) are clocked into the TAP controller, instruction register or selected test data register on the rising edge of TCK. Changes at the TAP output signal (TDO) occur on the falling edge of TCK. This signal **does not** conform to IEEE 1149.1/D5 requirement of TCK being a free-running clock at all times. TCK must be **stopped at 1** when internal BIT features are accessed. The TCK input has **a built in pull-up resistor** to ensure a high signal value is seen on an unterminated input.

TMS — Test Mode Select. Input. TMS is a serial control input which is clocked into the TAP controller on the rising edge of TCK. The TMS input has a built in pull-up resistor to ensure a high signal value is seen on an unterminated input.

TDI — Test Data Input. Input. TDI is clocked into the selected register—data or instruction—on the rising edge of TCK. The TDI input has a built in pull-up resistor to ensure a high signal value is seen on an unterminated input.

TDO — Test Data Output. Output. The contents of the selected register—data or instruction—are shifted out of the TDO on the falling edge of TCK. TDO is tri-stated except when scanning of data is in progress.

TRST- — Test Reset. Input. TRST- is the reset input to the TAP controller. Assertion of this input forces the TAP controller into the reset state. The TRST- input **does not** conform to IEEE 1149.1/D5 as it has a built in **pull-down** resistor to ensure a low signal value is seen on an unterminated input to force the TAP controller into the reset state.

### *5.2.7 Power and Ground Pins*

The current frame design allocates the following number of power and ground pins.

$V_{DD}$<18:00> — +3.3 Volt to +5.0 Volt Power pins.

$V_{SS}$<19:00> — 0 Volt Ground Pins.

## 5.3 Bus Transaction Types

Normal bus transfers begin with the assertion of STC- and end with the assertion of DTACK-. In case of an exception during a bus transfer, the transaction may be ended by the assertion of RESET- or BERR-

with DTACK-. Interlocked bus transfers end with the negation of LOCK- following a DTACK-. Multiple word transfers end when IOCOUNT<1:0> are zero with assertion of DTACK-.

Instruction prefetch is initiated by the prefetch unit according to the chosen prefetch strategy, selectable via the CONFIG register PM-bit. Section 2.11 details the prefetching strategy.

A data read transaction is initiated by the ATT2100 to read data from memory or a peripheral device.

A data write transaction is initiated by the ATT2100 to write data to memory or a peripheral device.

**TABLE 5-5.** Latch and Toggle Points

| Pin Name | I/O | Latch Point | Toggle Point | Drive Point | Tri-state Point |
|---|---|---|---|---|---|
| A<31:02> | O | | ↑34 | ↑34 | ↑34 |
| BGRANT- | I | ↑23 | | | |
| IOCOUNT<1:0> | O | | ↑34 | ↑34 | ↑34 |
| BERR- | I | ↑34 | | | |
| BGACK- | O | | ↓23 | | |
| BREQ- | O | | ↑23 | | |
| BM<3:0> | O | | ↑34 | ↑34 | ↑34 |
| D<31:00> | I/O | ↑34 | ↓23 | ↓23* | |
| D/T- | O | | ↑34 | ↑34 | ↑34 |
| DTACK- | I | ↑34 | | | |
| DTRI- | I | not latched | | | |
| HOLD- | I | ↑23 | | | |
| IL<2:0> | I | ↓23 | | | |
| LOCK- | O | | ↑34 | ↑34 | ↑34 |
| NCACHE- | O | | ↑34 | ↑34 | ↑34 |
| RESET- | I | ↑34 | | | |
| STC- | O | | ↑34 | ↑34 | |
| STOP- | I | not latched | | | |
| RETRY- | I | ↑23 | | | |
| W/R- | O | | ↑34 | ↑34 | ↑34 |

* The DTRI- signal can tri-state the data pins a propagation delay from assertion.

↑34 refers to the rising edge of CK34
↓34 refers to the falling edge of CK34
↑23 refers to the rising edge of CK23
↓23 refers to the falling edge of CK23

### 5.4 Protocol

In this section the various bus transactions are described with respect to the phases of the system clock.

### *5.4.1 ATT2100 and System Clocks*

All other devices using the synchronous bus must derive their clocks from CK23 and CK34.

The ATT2100 bus cycle begins on the rising edge of the CK34 and ends on the next rising edge.

### *5.4.2 Latch and Toggle Points of Signals*

Table 5-5 gives the ATT2100 input latch, output toggle, output drive and output tri-state points.

### *5.4.3 Reset*

The reset sequence is initiated by assertion of RESET- or by internal events. The reset sequence seen on the I/O for an externally requested reset follows:

— The RESET- signal must be held low for at least 2 clock cycles before the ATT2100 is reset. (RESET- is internally double sampled on the rising edge of Phase 3)

— While RESET- is held low all output signals except BREQ- and BGACK- are tri-stated. BREQ- and BGACK- are driven inactive (high).

— After RESET- is de-asserted the ATT2100 internally executes the reset sequence. If BGRANT- is asserted to the ATT2100, all output signals are driven to their inactive levels and once the internal reset sequence is completed, bus transactions begin.

— After the internal reset sequence completes, BREQ- is asserted by the ATT2100.

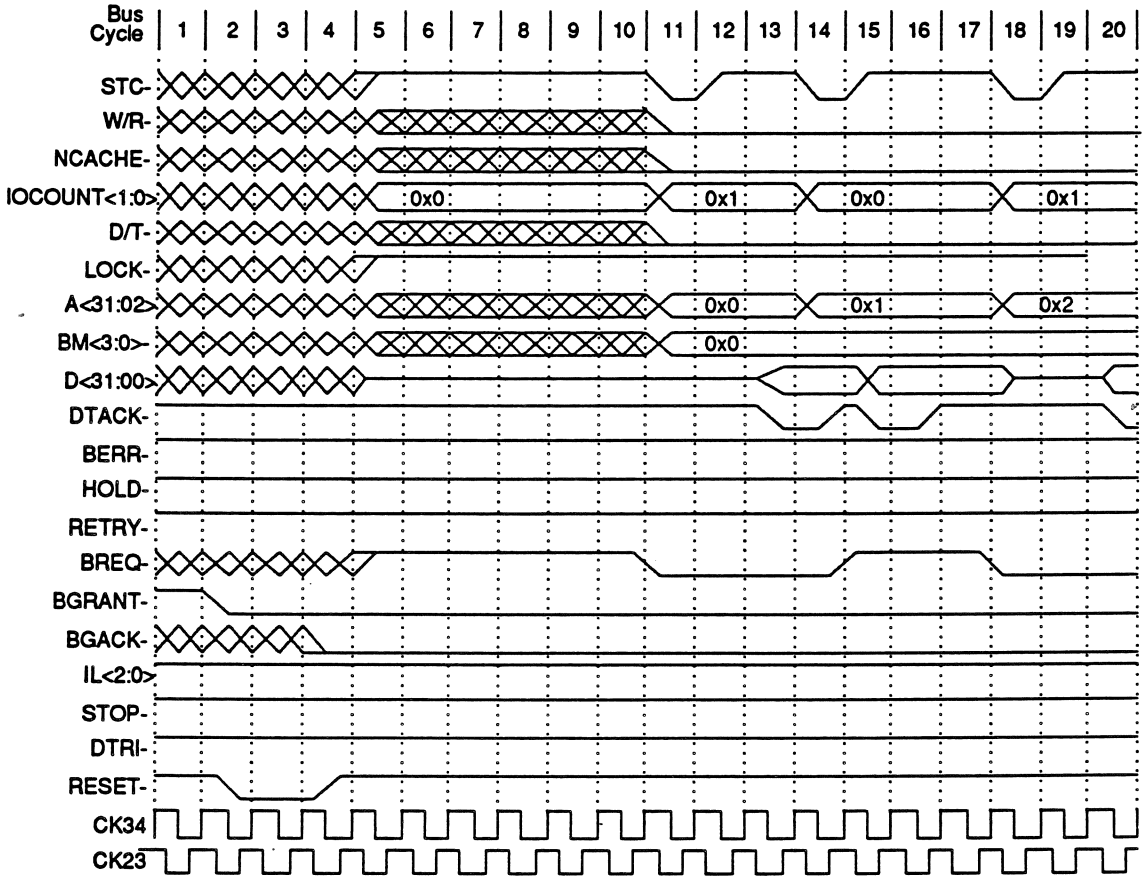



**Figure 5-1.** Reset with Bus Grant Asserted

A reset sequence can be internally generated. If there is a bus transaction owned by the ATT2100 in progress when the internal reset is generated, the transaction is allowed to complete prior to execution of the reset sequence. There is no external indication that an internal reset has occured other than the text fetch which occurs at location 0x0 after the internal reset sequence completes. Section 2.13.1 gives details on the internal reset sequence.

Refer to Figure 5-1 for the following example.

In bus cycle 2, phase 1, the system asserts RESET- and BGRANT-. In bus cycle 3, phase 1, the ATT2100 asserted BGACK-. In bus cycle 4, phase 2, the system de-asserts RESET-. In bus cycle 5, phase 3, the ATT2100 takes ownership of the bus de-asserting STC-, BREQ- and LOCK-, asserting IOCOUNT<0:1> at zero and all other bus signals at unknown except for D<31:00> which is tri-stated.

In bus cycles 6 through 9, the ATT2100 is going through the internal reset sequence. In bus cycle 10, the ATT2100 begins to fetch text at location 0x0.

### *5.4.4 Bus Arbitration*

To facilitate multiple bus masters, the bus arbitration protocol does not make the ATT2100 default master. A centralized arbiter selects the current bus master and controls transactions over the bus.

A synchronous bus protocol is used to exchange ownership of the bus from one master to another. The central bus arbiter must execute this protocol, asserting and negating BGRANT- to the various bus masters in a consistent manner.

The signals involved in this protocol generated by the central bus arbiter are: RESET-, BGRANT-, DTACK- and RETRY-. There are separate BGRANT- per bus master, with the other signals shared among bus masters.

The signals involved in this protocol generated by the bus masters are: BREQ-, STC-, IOCOUNT<0:1> and LOCK-. There is a separate BREQ- per bus master, with the other signals shared among bus masters.

Upon reset of the system, which must be synchronous, the arbiter selects one of the bus masters as current bus master by asserting its BGRANT-. Having received BGRANT-, the master takes ownership of the bus. The bus arbiter monitors the bus, keeping track of the state of the bus.

A ATT2100 asserts BREQ- when an I/O transaction is pending (upon reset all ATT2100 microprocessors want to start execution at address 0x0).

The arbiter selects a new bus master by negating BGRANT- to the current bus master and asserting BGRANT- to the next bus master at the end of any outstanding bus transactions. The current bus master loses ownership of the bus with negation of BGRANT-. If the current bus master loses BGRANT- with an outstanding transaction on the bus, that master remains on the bus until DTACK- is asserted.

The new bus master takes ownership of the bus at the beginning of the next bus cycle after receipt of BGRANT-. The arbiter must assert BGRANT- in a manner which inserts a dead-cycle between the end of the previous bus owner's cycle and the beginning of the next bus owner's cycle.

For systems which need to know when the ATT2100 has relinquished the bus in response to the negation of BGRANT-, BGACK- is provided. BGACK- is asserted when the ATT2100 owns the bus and de-asserted when it no longer owns the bus.

#### *5.4.4.1 Requesting the Bus*

The ATT2100 makes requests for the bus using the BREQ- signal whenever there is a valid request for the bus within the ATT2100.

Refer to Figure 5-2 for the following example.

In bus cycle 1, the ATT2100 does not have ownership of the bus but has a valid request for the bus internally. In bus cycle 2, phase 4, the ATT2100 asserts BREQ-. In bus cycle 3, phase 1, the external arbiter asserts BGRANT-. In bus cycle 4, phase 3, the ATT2100 drives A<31:02> and transfer qualifier signals. I bus cycle 4, phase 4, the ATT2100 asserts BGACK- and drives D<31:00> on write cycles.

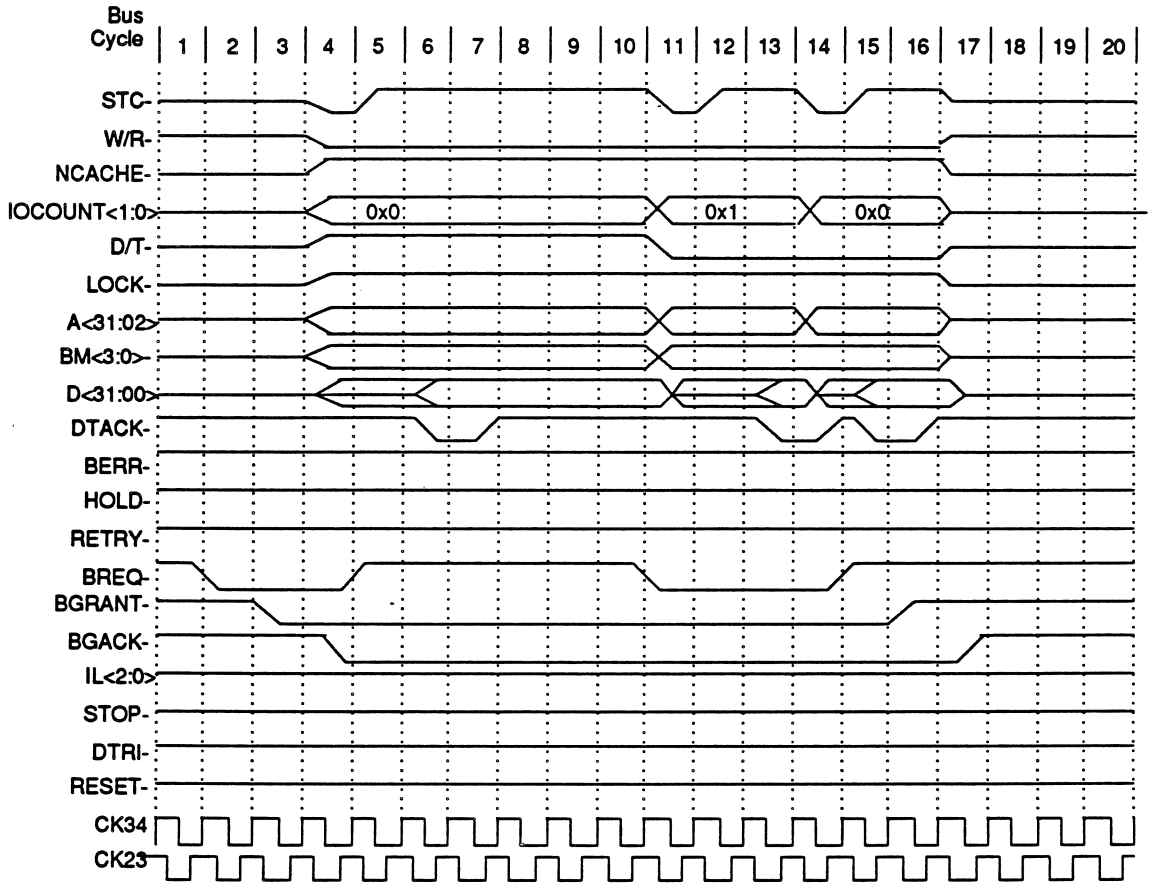



**Figure 5-2.** ATT2100 Read Bus Cycles with Bus Arbitration

*5.4.4.2 Surrendering The Bus*

The arbiter signals the ATT2100 to release the bus by de-asserting BGRANT-. When BGRANT- is de-asserted, the ATT2100 will relinquish ownership of the bus which is signaled by the ATT2100 de-asserting BGACK-. If the ATT2100 is running a bus transaction and BGRANT- is de-asserted, ownership of the bus will be relinquished after receipt of DTACK-. If the ATT2100 is not running a bus transaction and BGRANT- is de-asserted, ownership of the bus will be relinquished at the beginning of the next bus cycle. BGACK- will be de-asserted by the ATT2100 in the bus cycle in which ownership of the bus is being relinquished.

Most arbitration protocols will want to make the current processor remain bus master as long as the processor has an outstanding transaction on the bus. To implement such a protocol the the bus arbiter should de-assert BGRANT-:

— At the end of a single or *n* cycle transfer signaled by DTACK-,

— after the last transfer of a block transfer signaled by DTACK-,

— at the end of an interlocked transfer signaled by DTACK-, or

— during a retryed transaction.

The first three conditions are detected by checking IOCOUNT<1:0> are equal to zero and LOCK- is de-asserted. Detection of the retryed transaction will vary depending upon the design of the system.

Refer to Figure 5-2 for the following example.

In bus cycle 15, phase 4, the ATT2100 is on the bus and de-asserts BREQ-. In bus cycle 16, phase 1, the external arbiter de-asserts BGRANT-. In bus cycle 16, phase 2, the external contoller asserts DTACK-. In bus cycle 17, phase 4, the ATT2100 tri-states A<31:02> and transfer qualifier signals. In bus cycle 17, phase 1, the ATT2100 tri-states D<31:00> and de-asserts BGACK-.

*5.4.5 Read Transactions*

Read transactions occure to fetch text or data. Text reads are double-word transfers. Data reads are either single-, double- or quad-word transfers. After completion of a read transaction, looping-back is performed if the ATT2100 remains owner of the bus and there are no pending bus transactions.

Refer to Figure 5-2 for the following example.

In bus cycle 4, phase 1, the ATT2100 asserts STC-, A<31:02> and the transfer qualifier signals. In bus cycle 5, phase 1, the ATT2100 de-asserts STC- and a wait-state is inserted. In bus cycle 6, phase 2, the slave device asserts D<31:00> and DTACK- to end bus transaction.

In bus cycle 7, phase 3, ownership of the bus is maintained and loop-back cycle performed. The ATT2100 holds all bus signals their previous values. In bus cycle 7, phase 4, the ATT2100 loops-back the data read on the previous cycle. In bus cycles 8 through 10, loop-back cycles are performed.

In bus cycle 11, phase 1, the ATT2100 asserts STC-, A<31:02> and all other data transfer signals. In bus cycle 12, phase 1, the ATT2100 de-asserts STC- and a wait-state cycle is inserted. In bus cycle 13, phase 2, the slave device asserts D<31:00> and DTACK- to end the bus transaction.

In bus cycle 14, phase 1, the ATT2100 asserts STC-, A<31:02> and all other transfer qualifier signals. In bus cycle 15, phase 1, the ATT2100 de-asserts STC- and a wait-state is inserted. In bus cycle 15, phase 2, the slave device asserts D<31:00> and DTACK- to end the bus transaction.

In bus cycle 16, phase 3, ownership of the bus maintained and loop-back cycle performed. The ATT2100 holds all bus signals at their previous values. In bus cycle 16, phase 4, the ATT2100 loops-back the data read on the previous cycle.

NOTE: The bus transaction may be ended by RESET- or BERR- with DTACK- to signal an exception.

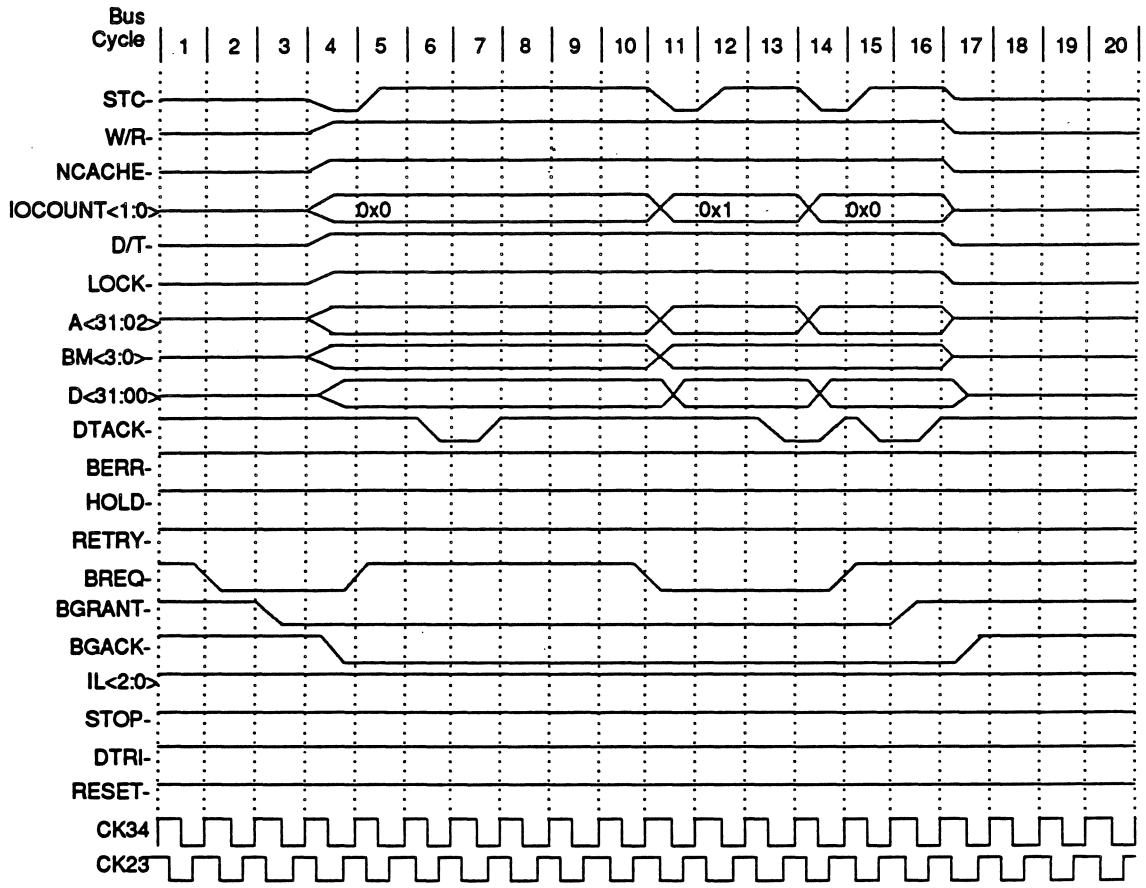



**Figure 5-3.** ATT2100 Write Bus Cycles with Bus Arbitration

*5.4.6 Write Transactions*

Write transactions are either single-, double- or quad-word transfers.

Refer to Figure 5-3 for the following example.

In bus cycle 4, phase 1, the ATT2100 asserts STC-, A<31:02> and the transfer qualifier signals. In bus cycle 5, phase 2, the ATT2100 asserts D<31:00>. In bus cycle 5, phase 1, the ATT2100 de-asserts STC- and a wait-state is inserted. In bus cycle 6, a wait-state is inserted. In bus cycle 7, phase 2, the slave device asserts DTACK- to end bus transaction.

In bus cycle 8, phase 3, ownership of the bus is maintained with the ATT2100 holding all bus signals at their previous values. In bus cycle 9 and 10, the ATT2100 maintains the previous bus signal values.

In bus cycle 11, phase 1, the ATT2100 asserts STC-, A<31:02> and the transfer qualifier signals. In bus cycle 11, phase 2, the ATT2100 asserts D<31:00>. In bus cycle 12, phase 1, the ATT2100 de-asserts STC- and a wait-state is inserted. In bus cycle 13, phase 2, the slave device DTACK- to end the bus transaction.

In bus cycle 14, phase 1, the ATT2100 asserts STC- A<31:02> and the transfer qualifier signals. In bus cycle 14, phase 2, the ATT2100 asserts D<31:00>. In bus cycle 15, phase 1, the ATT2100 de-asserts STC-. In bus cycle 15, phase 2, the slave device asserts DTACK- to end the bus transaction.

In bus cycle 16, phase 3, ownership of the bus is maintained with the ATT2100 maintaining all bus signals at their previous values.

NOTE: The bus transaction may be ended by RESET- or BERR- with DTACK- to signal an exception.

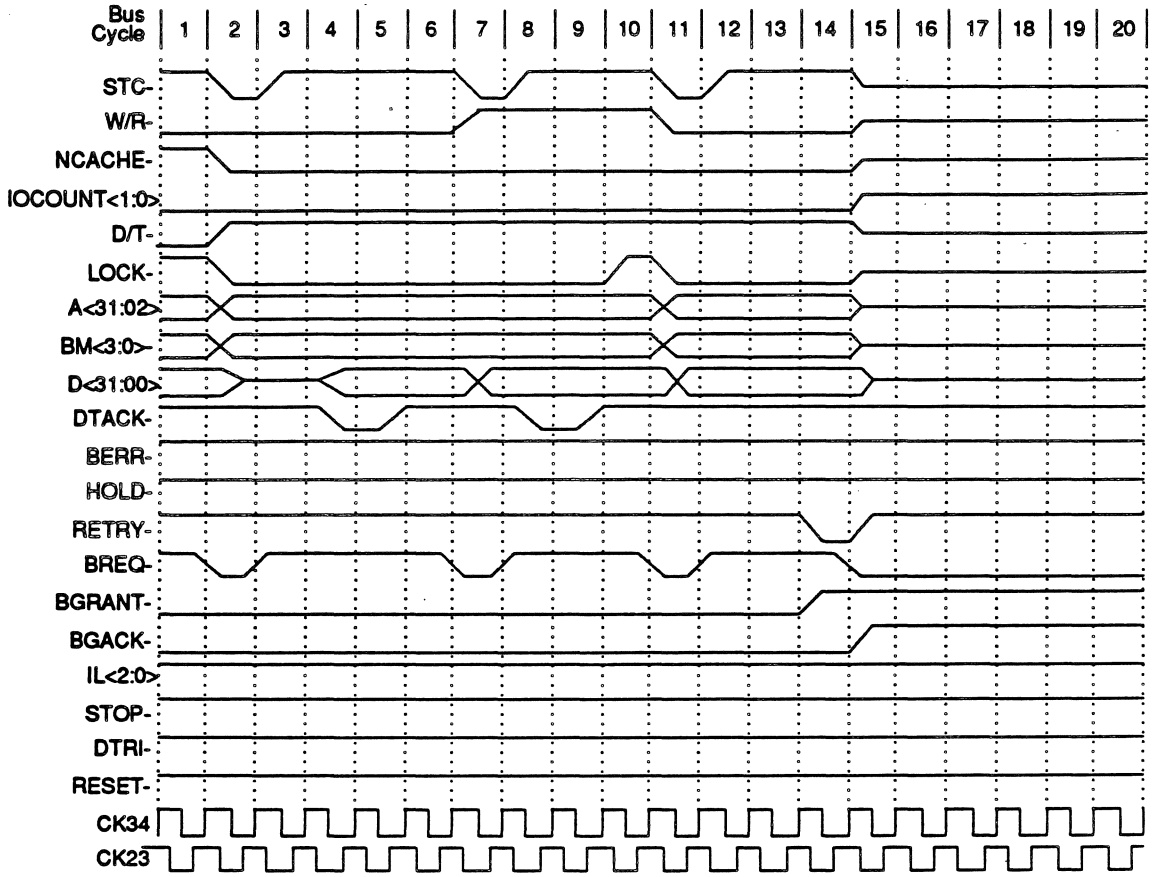



**Figure 5-4.** Interlocked Bus Transfer without and with Retry

*5.4.7 Interlocked Bus Transfer.*

This is a read-modify-write type bus operation. This sequence of operations is non-interruptible. The bus remains locked through the write. If BGRANT- is de-asserted during an interlocked operation, the operation is completed and transfer of bus ownership is delayed a clock cycle.

Refer to Figure 5-4 for the following example.

In bus cycle 2, phase 1, the ATT2100 asserts STC-, A<31:02>, LOCK- and the other transfer qualifier signals indicating the the read portion of a RMW transaction. In bus cycle 2, phase 2, the ATT2100 tri-states D<31:00>. In bus cycle 3, a wait-state is inserted. In bus cycle 4, phase 2, the slave device asserts D<31:00> and DTACK- to end the read portion of the RMW transaction.

In bus cycles 5 and 6, the ATT2100 is performing the internal operation.

In bus cycle 7, phase 1, the ATT2100 asserts STC-, A<31:02>, LOCK- and the other transfer qualifier signals indicating the the write portion of the RMW transaction. In bus cycle 7, phase 2, the ATT2100 asserts D<31:00>. In bus cycle 8, phase 2, the slave device asserts DTACK- to end the RMW transaction.

In bus cycle 9, LOCK- remains asserted by the ATT2100 adding a dead cycle. In bus cycle 10, the next bus cycles begins.

*5.4.8 Block Data Transfer*

The block transfer sizes that are supported are double- and quad-word. The block transfer looks like a series of single word bus transfers with the ATT2100 incrementing address bits A<03:02> and decrementing IOCOUNT<1:0> for each access.

Block transfers are non-interruptible. Block transfers may be retryed with the transfer resuming where it was aborted when RETRY- is de-asserted.

**5.5 Special Purpose Bus Transactions**

These include the exception and interrupt transactions and bus arbitration cycles. They are used by external devices to inform the ATT2100 of an unusual condition.

*5.5.1 Bus Error*

Errors in the system resulting from transactions requested by the ATT2100 are reported by assertion of BERR- with DTACK-.

Upon reception of a valid bus error the ATT2100 takes the following actions:

— The current bus transfer is ended.

— If when the error occurred the the bus transfer was an instruction prefetch, the ATT2100 puts the prefetch unit to sleep and goes on executing.

— If the bus transfer was not an instruction prefetch, the ATT2100 initiates a bus error trap which eventually causes the ATT2100 to branch to the exception handling routine.

— If an access to the interrupt stack, vector base table, segment table, or page table is faulted, the ATT2100 resets.

— Once a bus error is received, there is no guarantee that the next bus transaction is part of the exception routine. It may take several clock cycles until the bus error handling routine gets processed in the execution pipe. In the meantime other pipeline stages may have requests pending that may be honored by the I/O.

*5.5.1.1 Bus Error in an Interlocked or Block Transfer*

The timing for a bus error during an interlocked transfer is similar to that of a read or write transaction except that the ATT2100 de-asserts LOCK- after receiving a bus error and and performs a dead cycle.

A bus error during a block transfer terminates the transaction with transfer of ownership permitted.

*5.5.2 Retry*

RETRY- is needed to signal the ATT2100 that the current transaction should be aborted and then run again.

RETRY- must be asserted a set-up time prior to the rising edge of CK23. If RETRY- is asserted on an access DTACK- is masked. The next transaction initiated by the ATT2100 after the negation of RETRY- is a rerun of the transaction which was aborted by the assertion of RETRY-.

*5.5.2.1 Retry in an Interlocked Transfer*

Retrying the read portion of an interlock access is allowed. However, assertion of RETRY- in the write portion of the locked transfer is illegal as it might lead to the ATT2100 giving up the bus before the write portion is executed.

Refer to Figure 5-4 for the following example.

In bus cycle 11, phase 1, the ATT2100 asserts STC-, A<31:02>, LOCK- and the other transfer qualifier signals indicating the the read portion of a RMW transaction. In bus cycle 11, phase 2, the ATT2100 tri-

states D<31:00>. In bus cycle 12, a wait-state is inserted. In bus cycle 13, a wait-state is inserted. In bus cycle 14, phase 1, the system de-asserts BGRANT- and asserts RETRY- to abort the read portion of the RMW transaction.

In bus cycle 15, phase 4, the ATT2100 tri-states A<31:02> and the transfer qualifier signals. In bus cycle 15, phase 1, the ATT2100 tri-states D<31:00> and asserts BREQ- indicating it needs the bus to rerun the aborted RMW transaction.

It is not shown, but once BGRANT- is asserted the ATT2100 will rerun the aborted RMW transaction.

### *5.5.3 Interrupts*

The ATT2100 has seven possible levels of interrupt inputs which when recognized causes an interrupt trap and jump to a particular interrupt handling routine. This section only describes the method by which devices generate interrupts and how they are acknowledged (for details or the interval sequence of events see Section 2.13.2)

#### *5.5.3.1 Generating Interrupts*

The interrupting device must generate an encoded interrupt on the IL<2:0> lines. These interrupt lines are latched by the ATT2100 in phase 3. The decision of when the interrupts are internally sampled is made on an instruction to instruction basis (see Section 2.13.2). Most of the internal sampling is done after the completion of the current instruction. Once the interrupt has been internally sampled, the interrupt level is compared with the interrupt mask level in the PSW. If the interrupt is valid the appropriate interrupt sequence is executed.

NOTE: An external interrupt controller/encoder is needed to resolve conflicts between simultaneous interrupts and to encode interrupts.

### *5.5.4 TestingSupport*

While RESET- is asserted all outputs of the ATT2100 are disabled and the ATT2100 is put in an inactive state. The ATT2100 clock must be run for four clock cycles while RESET- is asserted to insure that RESET- is properly sampled. The RESET- signal can be used to tri-state all outputs except BREQ-, BGACK-, which are driven to one, and TDO which is only controlled by the IEEE 1149.1/D5 port.

Boundary scan is provided via a IEEE 1149.1/D5 interface. The complete specification of this interface is given in Section 10.

## 6. MEMORY MANAGEMENT

The ATT2100 has an on-chip Memory Management Unit, which can translate virtual addresses, as seen by a programmer, into physical addresses. Two methods for address translation are provided: paged and non-paged segments.

The 32-bit virtual address space is divided into 1,024 segments each representing 4MB of virtual addresses with a 4MB alignment. Paged segment address translation segments are further divided into 1,024-word pages. Non-paged segment address translation provides a variable sized contiguous segment of memory. In paged segment address translation, each page can be mapped anywhere in the 32-bit physical address space.

Address translation is enabled by setting the PSW VP-bit to 1. The ATT2100 has two Translation Look-aside Buffers (TLBs) — one for text addresses, one for data addresses — to speed paged segment address translation. Each TLB has 32 entries and is fully associative. Two registers — one for a text address and one for a data address — speed non-paged segment address translation. The non-paged segment registers (NPSRs) are compared in parallel with their respective TLB.

Additionally, to provide a physical prefetch buffer, a micro-TLB is provided for text references. This micro-TLB contains the last translation used by the prefetch unit and provides zero cycle address translation. If the micro-TLB misses, one cycle is required for update if the address translation hits in the text TLB or text NPSR.

If an address to be translated is not contained in the appropriate TLB or NPSR, an on-chip miss-processor automatically fetches the appropriate descriptor by walking the memory management tables.

### 6.1 Address Mapping

Logically, all addresses in the ATT2100 are translated by walking a series of map tables. All map tables in the ATT2100 memory mapping scheme are 4,096 bytes long — one Page Frame. All addresses contained within a memory management table are physical addresses, so address translation is not recursive.

A Page Frame is a contiguous region of 4,096 bytes, beginning at an address evenly divisible by 4,096 (the low-twelve-bits of the address are all 0). Because all Page Frames begin Page boundaries, additions are not necessary to calculate addresses.

Address mapping checks the validity of virtual addresses as well as translating them into physical addresses. A virtual address is flagged as illegal if:

— There is no physical mapping,

— user execution level code attempts to access kernel execution level addresses, or

— a store is attempted to read-only data.

Any violation is signaled as a memory fault, which generates a Fetch Exception (prefetches to invalid addresses do not generate a Fetch Exception), Read Exception (which may be ignored because of a mispredicted branch) or Write Exception (which may be postponed in favor of an interrupt). Section 2.13.3 gives details on the exceptions. Section 2.13.2 gives details on interrupts.

When using paged segment translation, virtual addresses are divided into three fields:

1. Segment Number
2. Page Number
3. Page Offset

A paged segment translation table entry is shown in Figure 6-1:

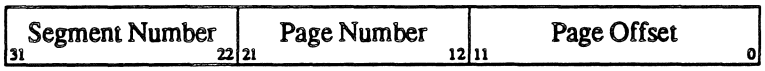



**Figure 6-1.** Page-based Virtual Address Format

When using non-paged segment translation, virtual addresses are divided into two fields:

1. Segment Number
2. Segment Offset

A non-paged segment translation table entry is shown in Figure 6-2:


**Figure 6-2.** Non-paged Virtual Address Format

### 6.2 Segment Mapping

The Segment Number selects from 1,024 entries in the Segment Table, a 4,096 byte table located in one Page Frame in physical memory. Each Segment Table Entry is 4 bytes long and contains the base address of a Page Table. The base address of the Segment Table is contained in the Segment Table Base register.

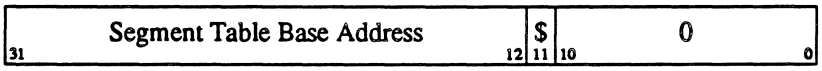



**Figure 6-3.** Segment Table Base Format

The Segment Table defines 1,024 segments each 1,024 pages long (for a total of 4,294,967,296 bytes). Segments are defined as a series of pages, so there may be "holes" in a segment's address space. There is no specific length specification for a segment: the validity of constituent pages defines a segment's extent. Each Segment Table Entry defines a Page Table. The address of a Segment Table Entry is formed by concatenating the upper 20-bits of the Segment Table Base Register with the upper 10-bits of the virtual address: the Base Address field in the STB defines the beginning of a Page Frame in physical memory, and the Segment Number field of the virtual address defines a word within that Page Frame. The Cache-bit ($) in the STB indicates whether the Segment Table Entries may be cached. If $ is set, the NCACHE- output pin is de-asserted when fetching this Segment Table Entry.

There are two possible formats for a Segment Table Entry. If paged segment address translation is used, the Segment Table Entry defines the base of a Page Table:


**Figure 6-4.** Paged Segment Table Entry Format

where the fields mean:

- Page Table Base Address: the Page Frame in physical memory of the Page Table
- $: Cache-bit. If 1, the NCACHE- output pin is de-asserted when fetching Page Table Entries.
- unused: available to software

- S: Segment-bit. A 0, indicates paged segment translation.
- V: Valid-bit. If 1, the entry is valid.

If non-page segment address translation is used, the Segment Table Entry defines the base and bound of a segment:

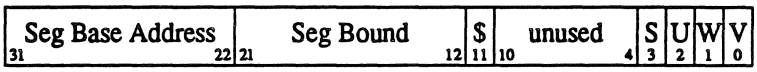



**Figure 6-5.** Non-paged Segment Table Entry Format

where the fields mean:

- Seg Base Address: the base of the segment in physical memory
- Seg Bound: the size of the segment, ranging from 4096 bytes (0x0) to 4M bytes (0x3FF)
- $: Cache-bit. When 1, the NCACHE- output pin is de-asserted when accessing non-paged segment and on chip cacheing is performed. When 0, the NCACHE- output is asserted when accessing non-paged segments. When the $-bit is 0, text fetches will not be cached in the Prefetch Buffer Cache but will be cached in the Decoded Instruction Cache. When the $-bit is 0, data fetches will be cached in the Stack Cache.
- unused: available to software
- S: Segment-bit. A 1, indicates non-paged segment translation.
- U: User-bit. If 1, the segment can be accessed at user execution level
- W: Writable-bit. If 1, the segment can be written (all valid NPSR can be read)
- V: Valid-bit. If 1, the entry is valid.

### 6.3 Page Mapping

The address of a Page Table Entry is formed by concatenating the upper 20-bits of the Segment Table Entry with bits 12-21 of the virtual address. Each Page Table occupies one Page Frame, so the Page Table Base Address is the address of the Page Frame and the Page Number field within the virtual address is a word offset within that Frame.

A Page Table Entry defines the physical address corresponding to the virtual address, along with providing protection information and other data available for paging algorithms. The Reference- and Modified-bits are automatically set by the miss-processing hardware (but must be cleared by software). The format of a Page Table Entry is:


**Figure 6-6.** Page Table Entry Format

where the fields mean:

- Page Frame Address: the physical address of the Page Frame in which the virtual address is mapped
- $: Cache-bit. If 1, the NCACHE output pin is de-asserted when accessing this page and on chip cacheing is performed. When 0, the NCACHE- output is asserted when accessing this page. When the $-bit is 0, text fetches will not be cached in the Prefetch Buffer Cache but will be cached in the Decoded Instruction Cache. When the $-bit is 0, data fetches will be cached in the Stack Cache.

- unused: available to software
- M: Modified-bit. Set to 1 when a page is first written. On subsequent writes to this page, the memory copy of the PTE is not accessed to set the M-bit. If a direct write to the memory copy of the PTE changes the M-bit, the entry should be flushed from the TLB using the **FLUSHPTE** instruction.
- R: Referenced-bit. Set to 1 when a page is first referenced. On subsequent references to this page, the memory copy of the PTE is not accessed to set the R-bit. If a direct write to the memory copy of the PTE changes the R-bit, the entry should be flushed from the TLB using the **FLUSHPTE** instruction.
- U: User-bit. If 1, the page can be accessed at user execution level (all readable pages can be accessed by the kernel)
- W: Writable-bit. If 1, the page can be written (all valid pages can be read)
- V: Valid-bit. If 1, the page is valid.

The Page Offset field of the virtual address defines the byte within the Page Frame in which the virtual address is mapped. The physical address consists of the Page Frame Address from the Page Table Entry concatenated with the Page Offset field of the Virtual Address. If a protection violation is detected, no memory access is made and a memory fault exception is executed. Section 2.13.3 gives details on exceptions.

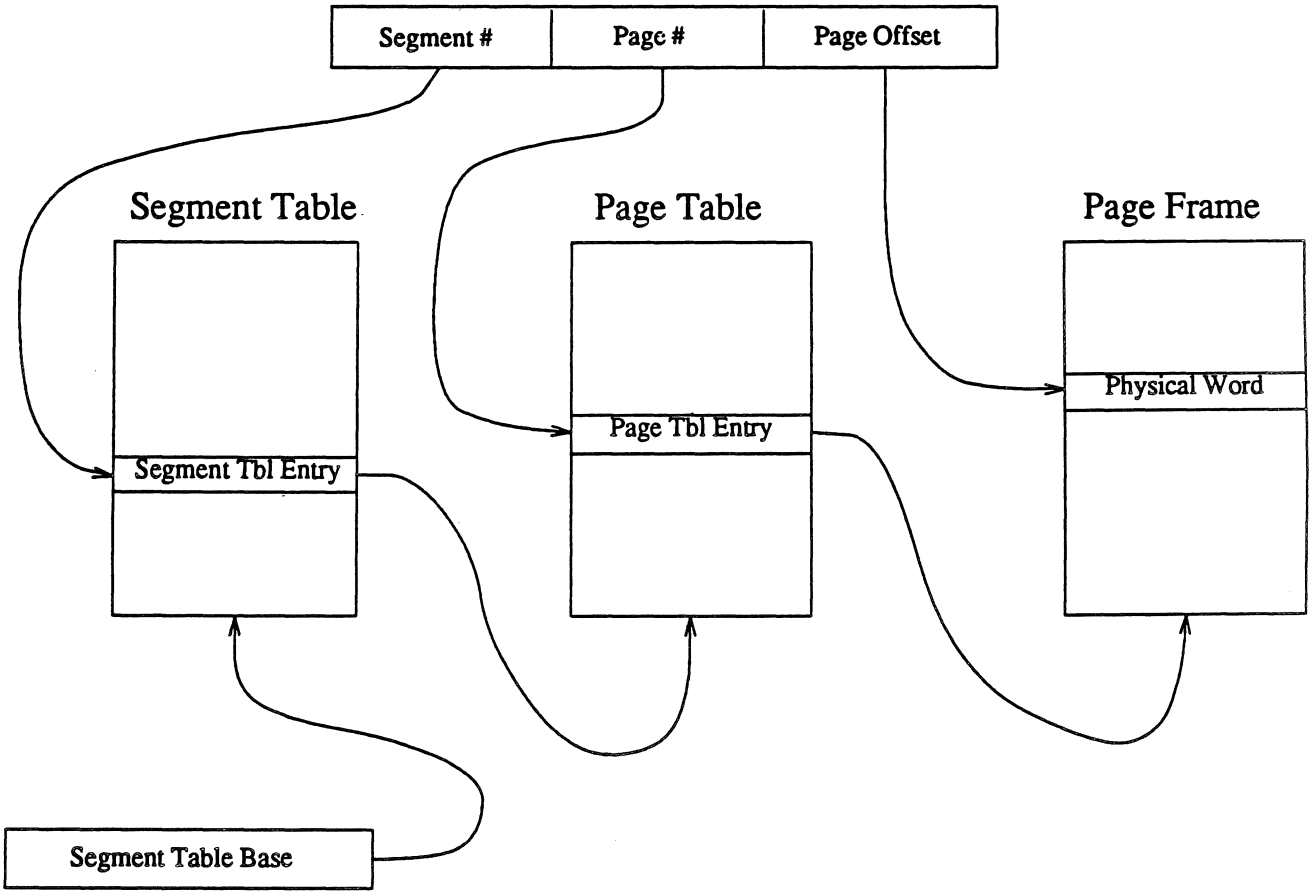



**Figure 6-7.** Paged Segment Address Mapping

### 6.4 Memory Management Summary

Address mapping for paged segments is summarized Figure 6-7. Address mapping for non-paged segments is summarized Figure 6-8.

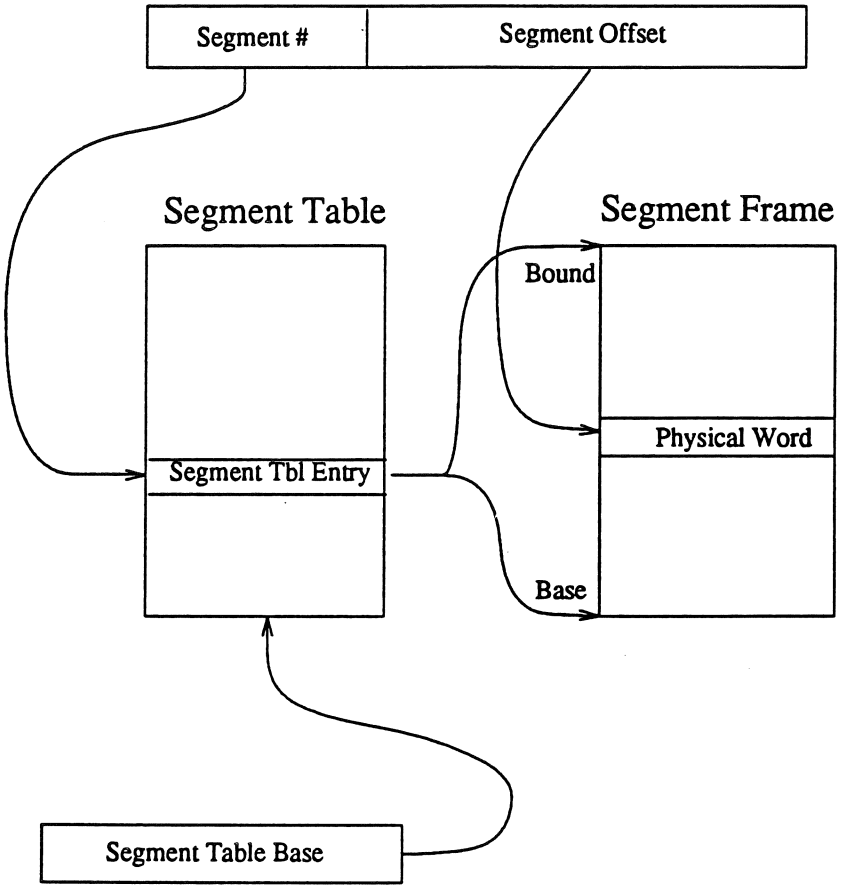



**Figure 6-8.** Non-paged Segment Address Mapping

### 6.5 Memory Management Operations

Both TLBs and NPSRs are completely flushed whenever the ATT2100 is reset (either by asserting the external reset pin, or the detection of an internal event which causes the ATT2100 to reset). The TLBs and NPSRs are also flushed whenever the Segment Table Base register is written.

Individual TLB and NPSR entries may be flushed with the **FLUSHPTE** instruction, described in Section 2.14. If the address created by the **FLUSHPTE** instruction is cached in one or both of the TLBs or NPSRs, the TLB or NPSR entry is marked invalid, so any subsequent access of that virtual address will be translated by the full memory mapping table walk. The **FLUSHPTE** instruction is not privileged, so a user process may flush any or all entries in the on-chip TLBs or NPSRs. Although this may degrade the performance of the process, it does not affect correctness, since the memory management tables in physical memory define the address mapping and the **FLUSHPTE** instruction does not alter the tables in memory.

The LOCK- output pin is asserted when Page Table Entries are fetched. If the R- and M-bits of the entry are current, the LOCK- output pin is cleared. If either R- or M-bits must be updated, the Page Table Entry is written back to memory with the LOCK- output pin still asserted. The LOCK- output pin is de-asserted when the write completes. Section 5.4.7 gives details on locked bus cycles.

If there is an external bus error signaled during the memory management table walk, the ATT2100 will take an exception. Section 2.13.3 gives details on exceptions.

### 6.6 MMU Performance

Table 6-1 gives the performance of address translation. These performance numbers do not include delay due to accessing the actual item being accessed. In Table 6-1, "A" indicates the I/O delay for a single word access.

**TABLE 6-1.** Address Translation Performance

| Condition | Penalty |
|---|---|
| Text reference, micro-TLB miss, TLB/NPSR miss, paged segment walk, R-bit modified | 3A + 3 |
| Text reference, micro-TLB miss, TLB/NPSR miss, paged segment walk, R-bit previously set | 2A + 3 |
| Text reference, micro-TLB miss, TLB/NPSR miss, non-paged segment walk | 1A + 1 |
| Text reference, micro-TLB miss, TLB/NPSR hit | 1 |
| Text reference, micro-TLB hit | 0 |
| Data read, TLB/NPSR miss, paged segment walk, R-bit modified | 3A + 3 |
| Data read, TLB/NPSR miss, paged segment walk, R-bit previously set | 2A + 3 |
| Data read, TLB/NPSR miss, non-paged segment walk | 1A + 3 |
| Data read, TLB/NPSR hit | 0 |
| Data write, TLB/NPSR miss, paged segment walk, R- and/or M-bit modified | 3A + 3 |
| Data write, TLB/NPSR miss, paged segment walk, R- and M-bit previously set | 2A + 3 |
| Data write, TLB/NPSR miss, non-paged segment walk, | 1A + 3 |
| Data write, TLB/NPSR hit | 0 |

## 7. ENVIRONMENTAL REQUIREMENTS

### 7.1 RELIABILITY

The reliability objectives for the ATT2100 microprocessor are:

5000 FITS[1] at the end of the 1st month
1700 FITS at the end of the 1st half year
500 FITS thereafter (long term reliability)

when the nominal junction temperature is at or below 85°C.

If the nominal junction temperature is at or below 55°C, the long term reliability objective is 250 FIT.

### 7.2 Shipping and Storage

The device (and heat sink if used) will be subjected to temperature cycling due to power cycling, shipping and storage. In consideration of seasonal temperature variations, a temperature range of -40°C to 65°C can be experienced during shipping. A range of -55°C to 125°C is allowable for storage.

### 7.3 POWER

The maximum power dissipation for the device at a case temperature of 85°C is 0.60 W at 20MHz. The allowable operating ambient temperature is 0°C to 70°C. The operating humidity range is 5% to 95%.

The junction temperature rise above local ambient temperature is equal to the thermal resistance and power product. The maximum power dissipated varies directly with the operating frequency over the practical range of operating and may be determined below:

$$P_{max} = 0.03 \times F$$

where Pmax is the maximum power (Watts), and F is the operating frequency (MHz). This does not imply an infinite selection of frequency, it is given only as a means of determining power at a given frequency.

1. FIT (Failures in 1,000,000,000 device-hours).

## 8. PHYSICAL DESIGN

### 8.1 ATT2100 Ceramic PGA Prototype Package

The ceramic prototype housing is a 100 mil spaced 125 CPGA package. The pinout for the 125 CPGA package is given in Figure 8-1 and Table 8-1.

The ATT2100 CPGA has the following attributes:

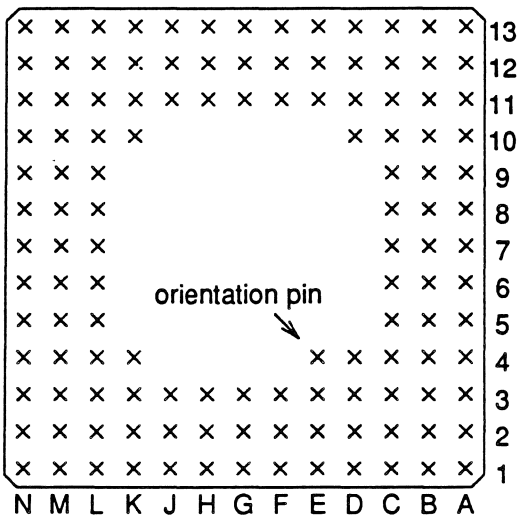
- $V_{DD}$ and $V_{SS}$ planes.
- Optional mounting of four low inductance 0.033 µF AVX surface mounted capacitors. Order 05085C333MAT050R from AVX.

Two versions of CPGA were designed: one package with HOLD tied to the VDD plane; one package with HOLD- connected to pin K3. The pinout in Table 8-1 shows HOLD- connected to K3, which is the preferred package configuration.

### 8.2 ATT2100 Plastic Prototype Package

The plastic prototype housing is a 0.25 mil pitch gull-lead package. The package conforms to the JEDIC standards for 132 pin PQFP. The pinout for the PQFP is given in Table 8-2. The pad number corresponds to the JEDIC pin number for 132 PQFP packages. The buffer type indicates the driver size for output and bi-directional buffers.

**Figure 8-1.** ATT2100 125 CPGA Pin Location




VIEW OF PGA PACKAGE FROM THE PIN SIDE

**TABLE 8-1.** ATT2100 125 PGA Pad Assignments with PGA Pin Assignment

| Pad | Name | Pin | Pad | Name | Pin | Pad | Name | Pin | Pad | Name | Pin |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 017 | D08 | C3 | 116 | A27 | C11 | 083 | D25 | L11 | 050 | TMS | L3 |
| 016 | A23 | D4 | 115 | D17 | D11 | 082 | A31 | K10 | 049 | TDI | K2 |
| 015 | D09 | A2 | 114 | A08 | A13 | 081 | D26 | N12 | 048 | IL0 | N1 |
| 014 | VSS01 | VSS | 113 | VSS06 | VSS | 080 | VSS11 | VSS | 047 | VSS16 | VSS |
| 013 | A24 | A3 | 112 | D18 | B13 | 079 | A13 | N11 | 046 | IL1 | M1 |
| 012 | D10 | B4 | 113 | A28 | E12 | 078 | D27 | L10 | 045 | IL2 | J2 |
| 011 | A04 | A4 | 110 | D19 | C13 | 077 | A02 | N10 | 044 | DTACK- | L1 |
| 010 | VDD01 | VDD | 109 | VDD06 | VDD | 076 | VDD11 | VDD | 043 | HOLD- | K3 |
| 009 | D11 | A5 | 108 | A09 | D13 | 075 | D28 | N9 | 042 | BERR- | K1 |
| 008 | VSS02 | VSS | 107 | VSS07 | VSS | 074 | VSS12 | VSS | 041 | VSS17 | VSS |
| 007 | A16 | C6 | 106 | D05 | E13 | 073 | A14 | M8 | 040 | RETRY- | J1 |
| 006 | D12 | B5 | 105 | A19 | F12 | 072 | D29 | M9 | 039 | BGRANT- | H2 |
| 005 | A05 | A6 | 104 | D06 | F13 | 071 | A03 | N8 | 038 | TDO | H1 |
| 004 | VDD02 | VDD | 103 | VDD07 | VDD | 070 | VDD12 | VDD | 037 | VDD17 | VDD |
| 003 | D00 | A7 | 102 | A10 | G13 | 069 | D03 | N7 | 036 | BREQ- | G1 |
| 002 | VSS03 | VSS | 101 | VSS08 | VSS | 068 | VSS13 | VSS | 035 | VSS18 | VSS |
| 001 | A17 | B6 | 100 | D07 | G11 | 067 | A15 | L7 | 034 | BGACK- | F2 |
| 132 | VDD03 | VDD | 099 | VDD08 | VDD | 066 | VDD13 | VDD | 033 | VDD18 | VDD |
| 131 | D01 | A8 | 098 | A20 | G12 | 065 | D04 | N6 | 032 | STC- | G2 |
| 130 | VSS04 | VSS | 097 | VSS09 | VSS | 064 | VSS14 | VSS | 031 | VSS19 | VSS |
| 129 | A06 | B8 | 096 | D20 | H13 | 063 | A22 | M6 | 030 | LOCK- | F1 |
| 128 | D02 | C9 | 095 | A11 | H12 | 062 | D30 | M5 | 029 | IOCOUNT0 | F3 |
| 127 | A18 | B7 | 094 | D21 | J13 | 061 | D31 | M7 | 028 | IOCOUNT1 | E1 |
| 126 | VDD04 | VDD | 093 | VDD09 | VDD | 060 | VDD14 | VDD | 027 | VDD19 | VDD |
| 125 | D13 | A9 | 092 | A21 | K13 | 059 | DTRI- | N5 | 026 | BM0- | D1 |
| 124 | VSS05 | VSS | 091 | VSS10 | VSS | 058 | VSS15 | VSS | 025 | VSS20 | VSS |
| 123 | A07 | A10 | 090 | D22 | L13 | 057 | CLK23 | N4 | 024 | BM1- | C1 |
| 122 | D14 | C10 | 089 | A12 | J12 | 056 | STOP- | L4 | 023 | BM2- | E3 |
| 121 | A25 | A11 | 088 | D23 | K12 | 055 | CLK34 | N3 | 022 | BM3- | D2 |
| 120 | VDD05 | VDD | 087 | VDD10 | VDD | 054 | VDD15 | VDD | 021 | VDD20 | VDD |
| 119 | D15 | B11 | 086 | A29 | M13 | 053 | RESET- | M3 | 020 | W/R- | B1 |
| 118 | A26 | D10 | 085 | D24 | K11 | 052 | TCK | K4 | 019 | NCACHE- | D3 |
| 117 | D16 | A12 | 084 | A30 | N13 | 051 | TRST- | N2 | 018 | D/T- | A1 |

VSS Pins - B3, B10, C2, C5, C8, C12, E4, E11, G3, H11, J3, L2, L6, L9, L12, M4, M11
VDD Pins - B2, B9, B12, C4, C7, D12, E2, F11, H3, J11, L5, L8, M2, M10, M12

**TABLE 8-2.** ATT2100 132 PQFP Pad Assignments with Buffer Types

| Pad | Name | Buffer | Pad | Name | Buffer | Pad | Name | Buffer | Pad | Name | Buffer |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 017 | D08 | 100pf | 116 | A27 | 100pf | 083 | D25 | 100pf | 050 | TMS | I |
| 016 | A23 | 100pf | 115 | D17 | 100pf | 082 | A31 | 100pf | 049 | TDI | I |
| 015 | D09 | 100pf | 114 | A08 | 150pf | 081 | D26 | 100pf | 048 | IL0 | I |
| 014 | VSS01 | | 113 | VSS06 | | 080 | VSS11 | | 047 | VSS16 | |
| 013 | A24 | 100pf | 112 | D18 | 100pf | 079 | A13 | 150pf | 046 | IL1 | I |
| 012 | D10 | 100pf | 111 | A28 | 100pf | 078 | D27 | 100pf | 045 | IL2 | I |
| 011 | A04 | 150pf | 110 | D19 | 100pf | 077 | A02 | 100pf | 044 | DTACK- | I |
| 010 | VDD01 | | 109 | VDD06 | | 076 | VDD11 | | 043 | HOLD- | I |
| 009 | D11 | 100pf | 108 | A09 | 150pf | 075 | D28 | 100pf | 042 | BERR- | I |
| 008 | VSS02 | | 107 | VSS07 | | 074 | VSS12 | | 041 | VSS17 | |
| 007 | A16 | 150pf | 106 | D05 | 115pf | 073 | A14 | 150pf | 040 | RETRY- | I |
| 006 | D12 | 100pf | 105 | A19 | 150pf | 072 | D29 | 100pf | 039 | BGRANT- | I |
| 005 | A05 | 150pf | 104 | D06 | 115pf | 071 | A03 | 100pf | 038 | TDO | 100pf |
| 004 | VDD02 | | 103 | VDD07 | | 070 | VDD12 | | 037 | VDD17 | |
| 003 | D00 | 115pf | 102 | A10 | 150pf | 069 | D03 | 115pf | 036 | BREQ- | 100pf |
| 002 | VSS03 | | 101 | VSS08 | | 068 | VSS13 | | 035 | VSS18 | |
| 001 | A17 | 150pf | 100 | D07 | 115pf | 067 | A15 | 150pf | 034 | BGACK- | 100pf |
| 132 | VDD03 | | 099 | VDD08 | | 066 | VDD13 | | 033 | VDD18 | |
| 131 | D01 | 115pf | 098 | A20 | 150pf | 065 | D04 | 115pf | 032 | STC- | 100pf |
| 130 | VSS04 | | 097 | VSS09 | | 064 | VSS14 | | 031 | VSS19 | |
| 129 | A06 | 150pf | 096 | D20 | 100pf | 063 | A22 | 150pf | 030 | LOCK- | 100pf |
| 128 | D02 | 115pf | 095 | A11 | 150pf | 062 | D30 | 100pf | 029 | IOCOUNT0 | 100pf |
| 127 | A18 | 150pf | 094 | D21 | 100pf | 061 | D31 | 100pf | 028 | IOCOUNT1 | 100pf |
| 126 | VDD04 | | 093 | VDD09 | | 060 | VDD14 | | 027 | VDD19 | |
| 125 | D13 | 100pf | 092 | A21 | 150pf | 059 | DTRI- | I | 026 | BM0- | 100pf |
| 124 | VSS05 | | 091 | VSS10 | | 058 | VSS15 | | 025 | VSS20 | |
| 123 | A07 | 150pf | 090 | D22 | 100pf | 057 | CLK23 | I | 024 | BM1- | 100pf |
| 122 | D14 | 100pf | 089 | A12 | 150pf | 056 | STOP- | I | 023 | BM2- | 100pf |
| 121 | A25 | 100pf | 088 | D23 | 100pf | 055 | CLK34 | I | 022 | BM3- | 100pf |
| 120 | VDD05 | | 087 | VDD10 | | 054 | VDD15 | | 021 | VDD20 | |
| 119 | D15 | 100pf | 086 | A29 | 100pf | 053 | RESET- | I | 020 | W/R- | 100pf |
| 118 | A26 | 100pf | 085 | D24 | 100pf | 052 | TCK | I | 019 | NCACHE- | 100pf |
| 117 | D16 | 100pf | 084 | A30 | 100pf | 051 | TRST- | I | 018 | D/T- | 100pf |

## 9. TIMING SPECIFICATIONS

This section contains preliminary diagrams of the signal timing that are based upon worst case slow simulations with the 0.9µm 2 level metal technology used with the ATT2100.

### 9.1 AC Load Specification

All preliminary timing specifications for output and input/output (IO) pins are based upon preliminary ADVICE simulations under worst case conditions in a 132 FPT plastic package with the respective load identified in Table 9-1.

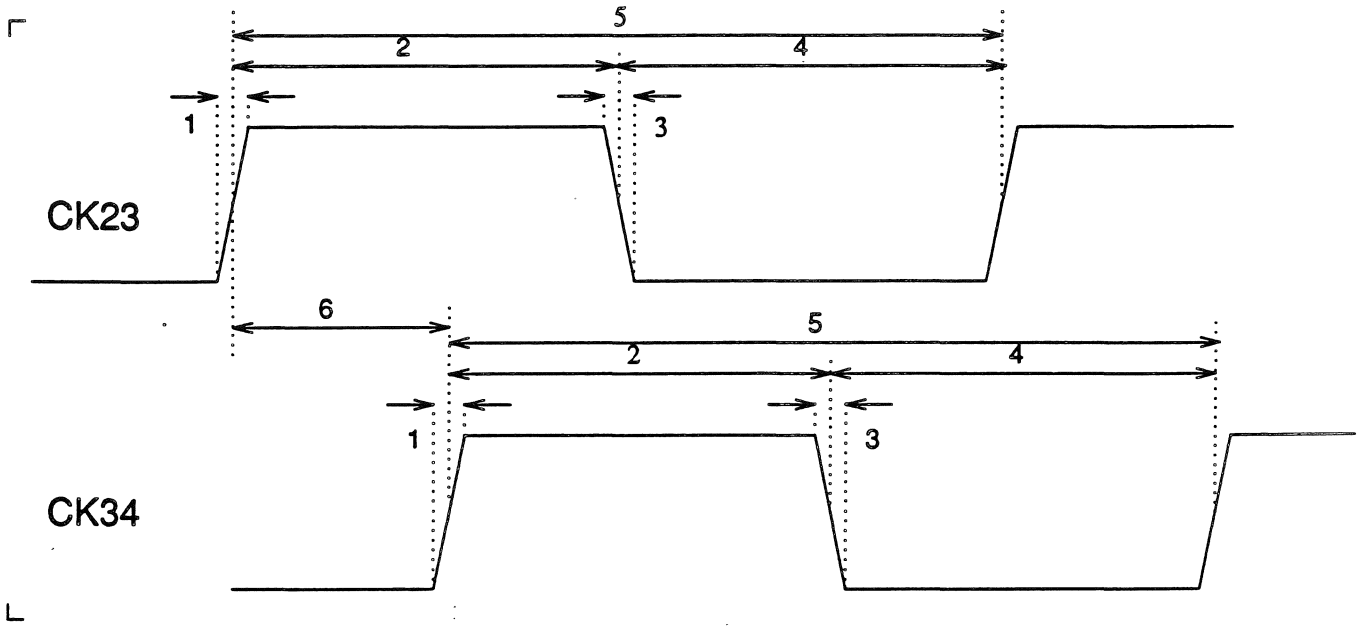
### 9.2 Load Specifications

**TABLE 9-1.** Loading Specifications

| Signal | Load (pf) |
|---|---|
| A<11:2> | 100 |
| A<25:12> | 150 |
| A<31:26> | 100 |
| BGACK- | 100 |
| BM<3:0>- | 100 |
| BREQ- | 100 |
| D<31:8> | 100 |
| D<7:0> | 115 |
| D/T- | 100 |
| IOCOUNT<1:0> | 100 |
| LOCK- | 100 |
| NCACHE- | 100 |
| STC- | 100 |
| TDO | 100 |
| W/R- | 100 |

### 9.3 Timing Diagrams

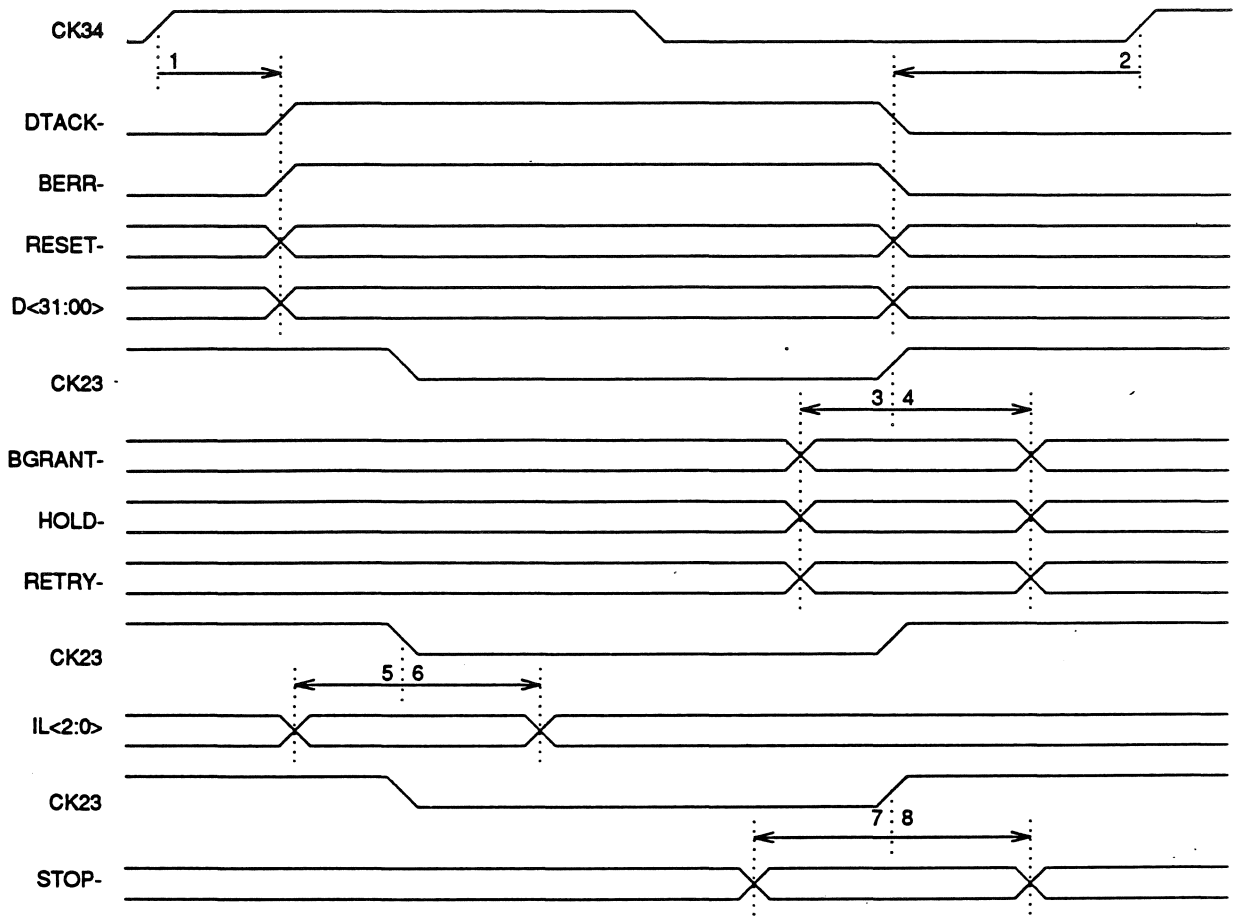
The following figures give preliminary timing specifications.




| CK23 and CK34 Input Timing | | | | | |
|---|---|---|---|---|---|
| No. | Description | Min. (ns) | Nom. (ns) | Max. (ns) | Var. (ns)* |
| 1 | Rise Time | - | 2.0 | 3.0 | ±1.0 |
| 2 | Pulse High | 23.5 | 25.0 | 26.5 | ±1.5 |
| 3 | Fall Time | - | 2.0 | 3.0 | ±1.0 |
| 4 | Pulse Low | 23.5 | 25.0 | 26.5 | ±1.5 |
| 5 | Period | 50.0 | - | 100.0 | - |
| 6 | Delay | 11.5 | 12.5 | 13.5 | ±1 |

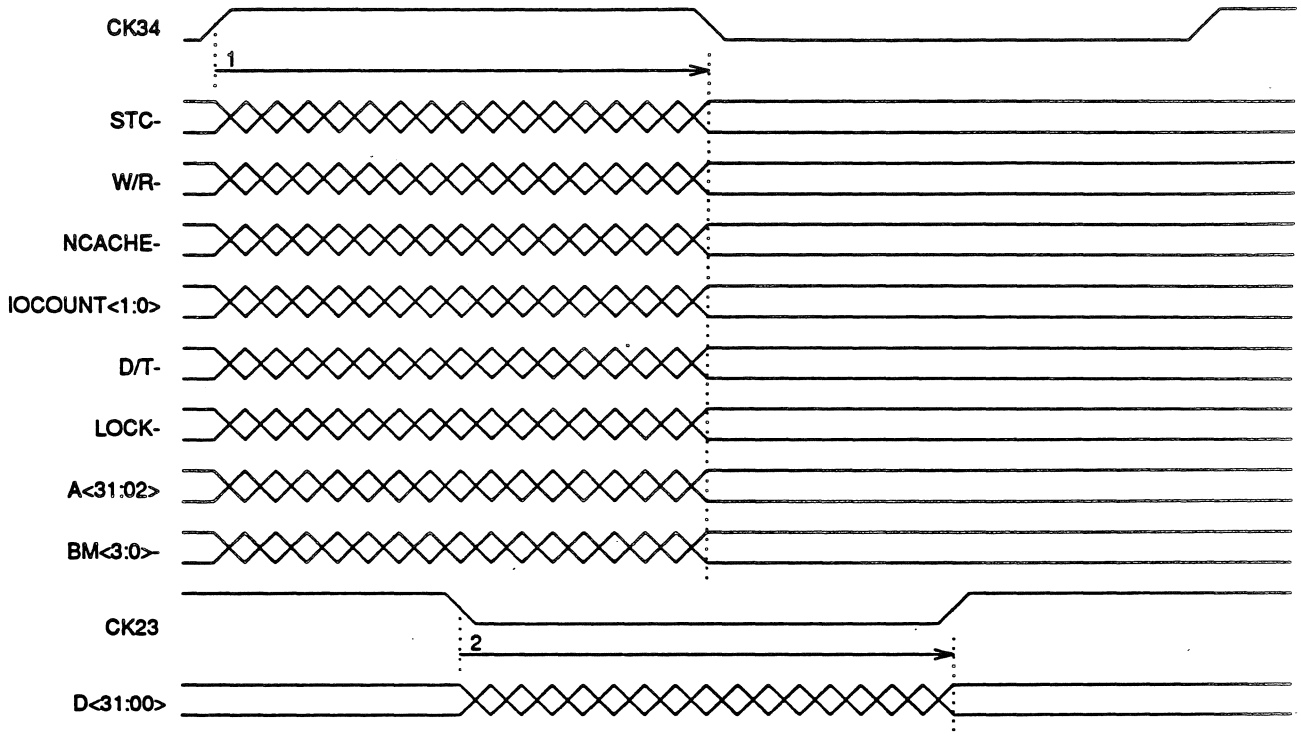
* The shortest phase permitted with these variations is 11.5 ns.
** The clocks may be stopped in phase 1.

**Figure 9-1.** Clock Input Timing.



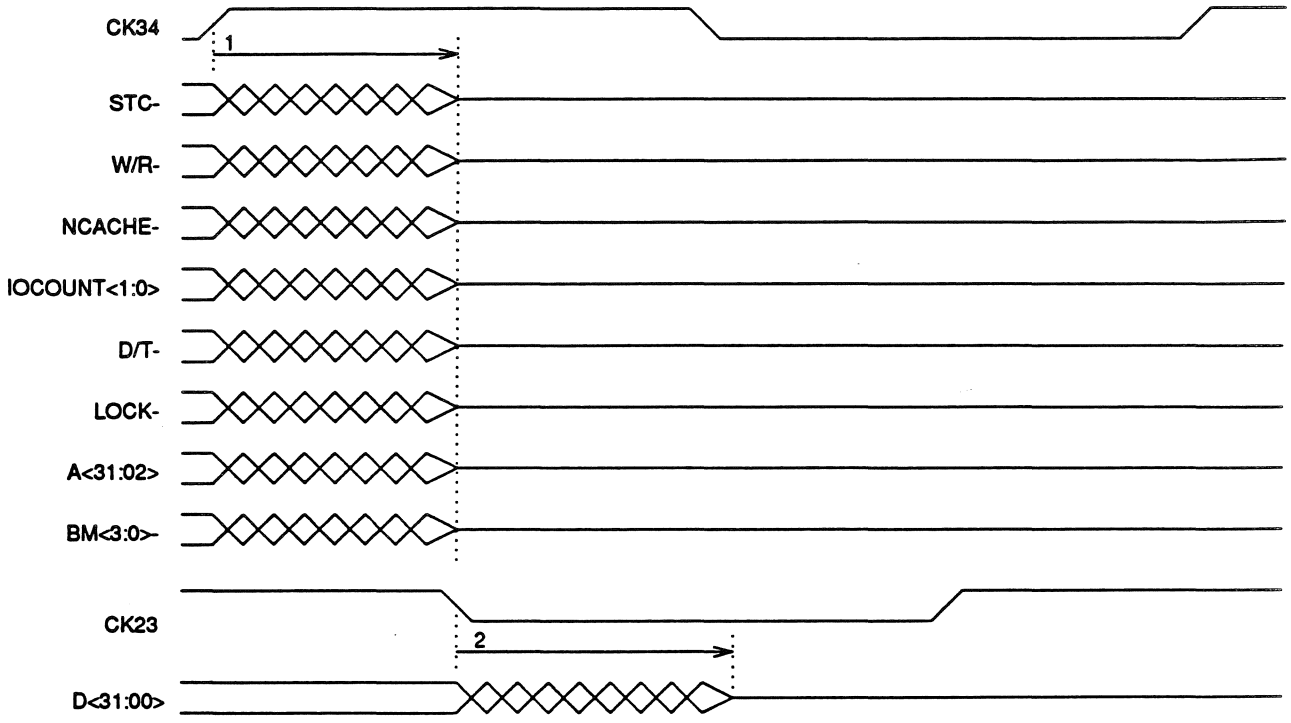

| Synchronous Input Timing | | | | | | |
|---|---|---|---|---|---|---|
| | | | Min. (ns) | | Max. (ns) | |
| No. | Description | Reference | 3.135V | 4.75V | 3.135V | 4.75V |
| 1 | DTACK-, BERR-, RESET-, D<31:00> Hold | CK34 rise | 6.0 | 5.0 | - | - |
| 2 | DTACK-, BERR-, RESET-, D<31:00> Set-up | CK34 rise | 4.0 | 3.0 | - | - |
| 3 | BGRANT-, HOLD-, RETRY- Set-up | CK23 rise | 4.0 | 3.0 | - | - |
| 4 | BGRANT-, HOLD-, RETRY- Hold | CK23 rise | 6.0 | 5.0 | - | - |
| 5 | IL<2:0> Set-up | CK23 fall | 4.0 | 3.0 | - | - |
| 6 | IL<2:0> Hold | CK23 fall | 6.0 | 5.0 | - | - |
| 7 | STOP- Set-up | CK23 fall | 4.0 | 3.0 | - | - |
| 8 | STOP- Hold | CK23 rise | 6.0 | 5.0 | - | - |

**Figure 9-2.** Synchronous Input Timing.



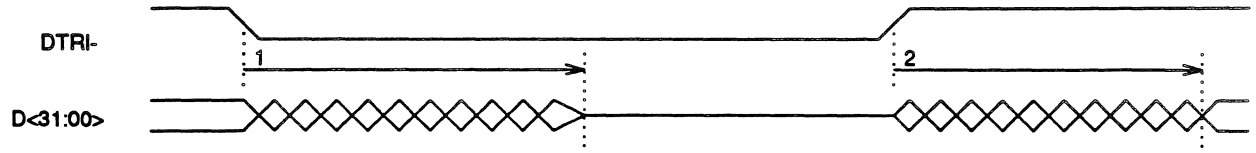

| Output Timing | | | | | | |
|---|---|---|---|---|---|---|
| | | | Min. (ns) | | Max. (ns) | |
| No. | Description | Reference | 3.135V | 4.75V | 3.135V | 4.75V |
| 1 | Address and Data Transfer Output Valid | CK34 rise | 12.5 | 9.0 | 25.0 | 18.0 |
| 2 | Data Output Valid | CK23 fall | 12.5 | 9.0 | 25.0 | 18.0 |

**Figure 9-3.** Output Timing.



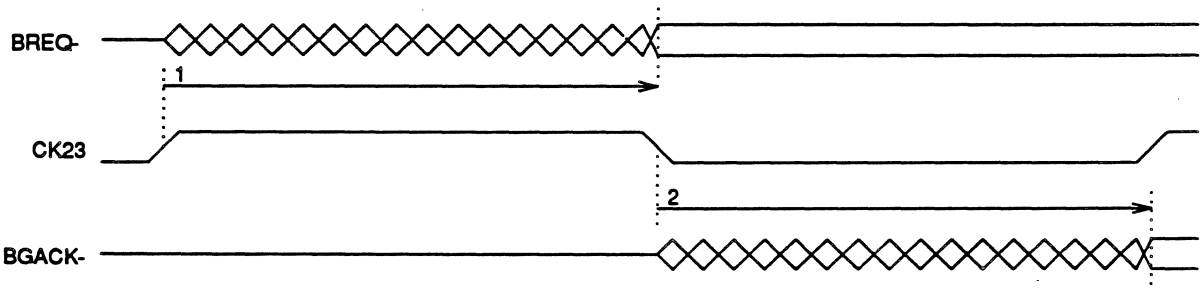

| Bus Relinquish Cycle Output Timing | | | | | | |
|---|---|---|---|---|---|---|
| | | | Min. (ns) | | Max. (ns) | |
| **No.** | **Description** | **Reference** | **3.135V** | **4.75V** | **3.135V** | **4.75V** |
| 1 | Address and Data Transfer Output Tri-state | CK34 rise | 6.2 | 4.5 | 12.5 | 9.0 |
| 2 | Data Output Tri-state | CK23 fall | 6.2 | 4.5 | 12.5 | 9.0 |

**Figure 9-4.** Bus Relinquish Cycle Output Timing.




| DTRI- to Data Tri-state Output Timing | | | | | | |
|---|---|---|---|---|---|---|
| | | | Min. (ns) | | Max. (ns) | |
| No. | Description | Reference | 3.135V | 4.75V | 3.135V | 4.75V |
| 1 | D<31:00> Tri-state | DTRI- fall | 12.5 | 9.0 | 25.0 | 18.0 |
| 2 | D<31:00> Valid | DTRI- rise | 12.5 | 9.0 | 25.0 | 18.0 |

**Figure 9-5.** DTRI- to Data Tri-state Output Timing.




| BREQ- and BGACK- Output Timing | | | | | | |
|---|---|---|---|---|---|---|
| | | | Min. (ns) | | Max. (ns) | |
| No. | Description | Reference | 3.135V | 4.75V | 3.135V | 4.75V |
| 1 | BREQ- Output Valid | CK23 rise | 12.5 | 9.0 | 25.0 | 18.0 |
| 2 | BGACK- Output Valid | CK23 fall | 12.5 | 9.0 | 25.0 | 18.0 |

**Figure 9-6.** BREQ- and BGACK- Output Timing.

## 10. Testability

the ATT2100 is a highly testable design providing access to all testability features via the IEEE 1149.1/D5 interface. The features which are accessible include:

- Single clock delay by-pass.
- Boundary-scan of I/O signals.
- Embedded memory Built-In-Test (BIT) and scan features.
- Embedded PLA BIT features.

### 10.1 Conformance

The Test Access Port (TAP) provided conforms to all aspects of the IEEE 1149.1/D5 except for TCK and TRST-.

In IEEE 1149.1/D5, TCK is required to be a free-running clock with any gating performed within the device. Due to the tight specification of the clocks within the ATT2100 design, this feature is **not** provided. It is required that gating of TCK be performed externally.

In IEEE 1149.1/D5, an unconnected TRST- is to be terminated in the **inactive** mode. In the ATT2100, an unconnected TRST- is terminated in the **active** mode holding the TAP state-machine in reset.

### 10.2 Test Access Port (TAP)

The Test Access Port (TAP) consists of five I/O pins and a sequential 16 state controller.

#### *10.2.1 TAP I/O*

The signals in the TAP are defined

| | |
|---|---|
| TCK | Test Clock. Input. An externally gated clock signal with a 50% duty cycle. The changes on the TAP input signals (TMS and TDI) are clocked into the TAP controller, instruction register or selected test data register on the rising edge of TCK. Changes at the TAP output signal (TDO) occur on the falling edge of TCK. This signal **does not** conform to IEEE 1149.1/D5 requirement of TCK being a free running clock. TCK must be stopped at 1. The TCK input has a built in pull-up resistor to ensure a high signal is seen on an unconnected input. |
| TMS | Test Mode Select. Input. TMS is a serial control input which is clocked into the TAP controller on the rising edge of TCK. The TMS input has a built in pull up resistor to ensure a high signal value is seen on an unconnected input. |
| TDI | Test Data Input. Input. TDI is clocked into the LSB of the selected register—data or instruction—on the rising edge of TCK. The TDI input has a built in pull up resistor to ensure a high signal value is seen on an unconnected input. |
| TDO | Test Data Output. Output. The contents of the MSB of the selected register—data or instruction—is shifted out of the TDO on the falling edge of TCK. TDO is tri-stated except when scanning of data is in progress. |
| TRST- | Test Reset. Active low input. TRST- is the reset input to the TAP controller. Assertion of this input forces the TAP controller into the reset state. The TRST- input has a built in pull down resistor to ensure a low signal value is seen on an unconnected input to force the TAP controller into the reset state. |

### *10.2.2 TAP Controller (TAPC)*

The TAPC is a synchronous finite state machine whereby sequencing through the various operations of the testability circuitry occurs under control of the TMS signal.

#### *10.2.2.1 TAPC State Diagram*

The state diagram for the TAPC is given in Figure 10-1. There are 16 states in this state machine with advancement of state dependent upon the value of TMS at the rising edge of TCK. All operations of the test logic occur on the rising edge of TCK following the entry into a controller state. Changes at TDO occur on the falling edge of TCK following entry into a controller state which selects TDO.

The states of the TAPC are defined in Table 10-1 and in Figure 10-1.

**TABLE 10-1.** TAP Controller State Table

| State | Name | Description |
|---|---|---|
| 0x0 | *Exit(2)-DR* | This is a temporary controller state. All test data registers and the instruction register retain their previous state. A high signal on the TMS line while in this state causes termination of the scanning process; a low causes entry into the *Shift-DR* state. |
| 0x1 | *Exit(1)-DR* | This is a temporary controller state. All test data registers and the instruction register retain their previous state. A high signal on the TMS line while in this state causes termination of the scanning process; a low causes entry into the *Pause-DR* state. |
| 0x2 | *Shift-DR* | In this controller state, the selected data register shifts data one stage towards its serial output on each rising edge of TCK. All registers other than the selected test data register retain their previous state. |
| 0x3 | *Pause-DR* | This controller state allows shifting of the selected test data register to be temporarily halted. All test data registers and the instruction register retain their previous state. The controller remains in this state while TMS is low. When TMS goes high, the controller advances to the *Exit(2)-DR* state. |
| 0x4 | *Select-IR-Scan* | This is a temporary controller state in which all test logic retains its previous state. If TMS is held low when the controller is in this state, then a scan sequence for the instruction register is initiated. |
| 0x5 | *Update-DR* | During this controller state, data is transferred from each shift-register stage into the corresponding parallel output latch (if the selected test data register includes a parallel output latch). All shift-register stages in the selected register retain their previous state. |
| 0x6 | *Capture-DR* | In this controller state data is parallel loaded into the selected test data register. If the register does not have a parallel input, or if capturing is not required for the selected test, the register retains its previous state unchanged. |

| | | |
|---|---|---|
| 0x7 | *Select-DR-Scan* | This is a temporary controller state in which all test logic retains its previous state. If TMS is held low when the controller is in this state, then a scan sequence for the selected test data register is initiated. |
| 0x8 | *Exit(2)-IR* | This is a temporary controller state. All test data registers and the instruction register retain their previous state. A high signal on the TMS line while in this state causes termination of the scanning process; a low causes entry into the *Shift-IR* state. |
| 0x9 | *Exit(1)-IR* | This is a temporary controller state. All test data registers and the instruction register retain their previous state. A high signal on the TMS line while in this state causes termination of the scanning process; a low causes entry into the *Pause-IR* state. |
| 0xA | *Shift-IR* | In this controller state, the instruction register shifts data one stage towards its serial output on each rising edge of TCK. |
| 0xB | *Pause-IR* | This controller state allows shifting of the instruction register to be temporarily halted. All test data registers and the instruction register retain their previous state. The controller remains in this state while TMS is low. When TMS goes high, the controller advances to the *Exit(2)-DR* state. |
| 0xC | *Run-Test/Idle* | The controller state between scan operations where an internal test previously selected by setting the instruction register may be executed. Registers not involved in the application of the test retain their previous state. If the data in the instruction register does not indicate that a test should be executed, then all test logic must retain their previous state. Once entered, the controller will remain in the *Run-Test/Idle* state as long as TMS is held low. |
| 0xD | *Update-IR* | During this controller state, the instruction is transferred from each shift-register stage of the instruction register into the parallel output latch of the instruction register . All shift-register stages in the instruction register retain their previous state. |
| 0xE | *Capture-IR* | In this controller state data is parallel loaded into the instruction register. If the register does not have a parallel input, or if capturing is not required for the selected test, the register retains its previous state unchanged. |
| 0xF | *Test-Logic-Reset* | While in this state all test circuitry is disabled. The Instruction Register (IR) is reset to select the by-pass register. The controller remains in this state as long as TMS is high or TRST- is asserted. |

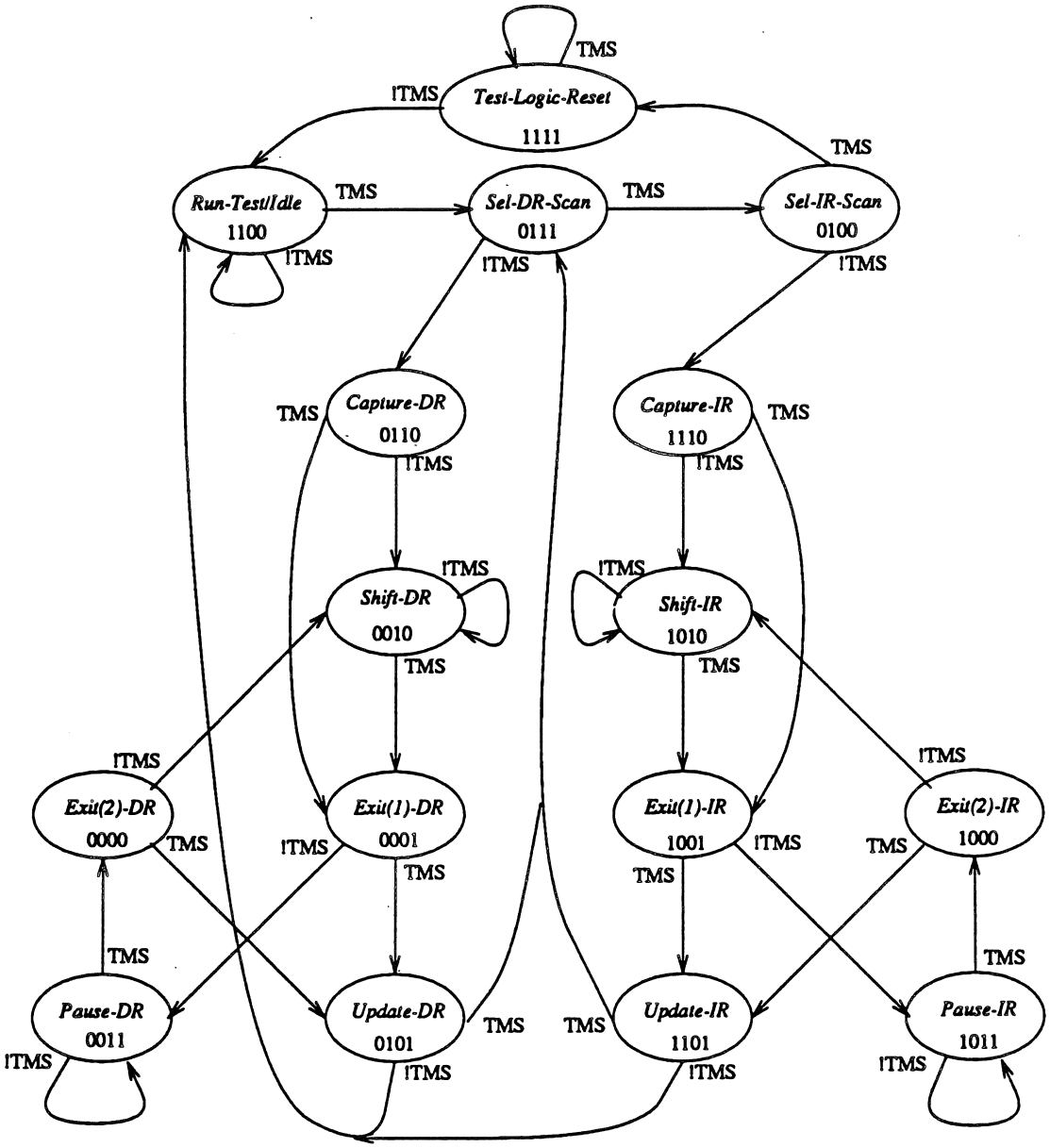



**Figure 10-1.** TAP Controller State Diagram

### 10.3 IEEE 1149.1/D5 Registers

The following registers are prescribed in the IEEE 1149.1/D5 specification.

#### *10.3.1 Instruction Register (IR)*

The instruction register (IR) allows a test instruction to be shifted into the ATT2100. The IR is used to select the test to be performed or the test data register to be accessed. The IR in the ATT2100 is seven bits in length. Table 10-2 identifies the instruction encodings.

**TABLE 10-2.** Instruction Register Encodings

| Instruction MSB→LSB | Register Selected | Instruction Mnemonic | Description |
|---|---|---|---|
| 0000000 | BS | EXTEST | BS selected with BS external test. |
| 0000001 | BS | SAMPLE | BS selected with BS sample. |
| 0000010 | BS | INTEST | BS selected with BS internal test. |
| 0000011 | PPLA | IRPPLA | PPLA selected with PPLA self test. |
| 0000100 | ICD | IRICD | Instruction Cache Data selected with ICD self test. |
| 0000101 | SC | IRSC | Stack Cache selected with SC self test. |
| 0000110 | PFD | IRPFD | Prefetch Cache Data selected with PFD self test. |
| 0000111 | PFT | IRPFT | Prefetch Cache Tag selected with PFT self test. |
| 0001xxx | NA | NA | Reserved. |
| 001xxxx | BP | BP | BP selected with all self test. |
| 01xxxxx | BP | BP | BP selected and BS sample. |
| 10xxxxx | ID | ID | ID selected and BS sample. |
| 11xxxxx | BP | BP | BP selected and BS sample. |

### *10.3.2 By-pass Register (BR)*

The by-pass (BP) register provides a single TCK delay path from TDI to TDO.

When the BP register is selected, a 0 is loaded on the rising edge of TCK in the *Capture-DR* controller state. When the *Test-Logic-Reset* controller state is entered the BP register retains its last value.

### *10.3.3 Boundary-scan Register (BS)*

The boundary-scan register allows testing of circuitry external to the ATT2100. Additionally, BS provides for sampling and examination of the I/O values without impacting the operation of the system logic.

90 shift elements are in the boundary scan shift chain. 91 TCKs are required to shift the entire chain from TDI through to TDO. Position is given from TDI to TDO.

| Position | Name | Description | Position | Name | Description |
|---|---|---|---|---|---|
| 1 | RESET- | Input | 2 | CK23 | Sample only input |
| 3 | STOP- | Sample only input | 4 | CK34 | Sample only input |
| 5 | DTRI- | Input | 6 | tridata | Control for IOputs |
| 7 | D31 | IOput | 8 | D30 | IOput |
| 9 | A22 | 3S-Output | 10 | D04 | IOput |
| 11 | A15 | 3S-Output | 12 | D03 | IOput |
| 13 | A03 | 3S-Output | 14 | D29 | IOput |
| 15 | A14 | 3S-Output | 16 | D28 | IOput |
| 17 | A02 | 3S-Output | 18 | D27 | IOput |
| 19 | A13 | 3S-Output | 20 | D26 | IOput |
| 21 | A31 | 3S-Output | 22 | D25 | IOput |
| 23 | A30 | 3S-Output | 24 | D24 | IOput |
| 25 | A29 | 3S-Output | 26 | D23 | IOput |
| 27 | A12 | 3S-Output | 28 | D22 | IOput |
| 29 | A21 | 3S-Output | 30 | D21 | IOput |
| 31 | A11 | 3S-Output | 32 | D20 | IOput |
| 33 | A20 | 3S-Output | 34 | D07 | IOput |
| 35 | A10 | 3S-Output | 36 | D06 | IOput |
| 37 | A19 | 3S-Output | 38 | D05 | IOput |

| | | | | | |
|---|---|---|---|---|---|
| 39 | A09 | 3S-Output | 40 | D19 | IOput |
| 41 | A28 | 3S-Output | 42 | D18 | IOput |
| 43 | A08 | 3S-Output | 44 | D17 | IOput |
| 45 | A27 | 3S-Output | 46 | D16 | IOput |
| 47 | A26 | 3S-Output | 48 | D15 | IOput |
| 49 | A25 | 3S-Output | 50 | D14 | IOput |
| 51 | A07 | 3S-Output | 52 | D13 | IOput |
| 53 | A18 | 3S-Output | 54 | D02 | IOput |
| 55 | A06 | 3S-Output | 56 | D01 | IOput |
| 57 | A17 | 3S-Output | 58 | D00 | IOput |
| 59 | A05 | 3S-Output | 60 | D12 | IOput |
| 61 | A16 | 3S-Output | 62 | D11 | IOput |
| 63 | A04 | 3S-Output | 64 | D10 | IOput |
| 65 | A24 | 3S-Output | 66 | D09 | IOput |
| 67 | A23 | 3S-Output | 68 | D08 | IOput |
| 69 | D/T- | 3S-Output | 70 | NCACHE- | 3S-Output |
| 71 | W/R- | 3S-Output | 72 | BM3- | 3S-Output |
| 73 | BM2- | 3S-Output | 74 | BM1- | 3S-Output |
| 75 | BM0- | 3S-Output | 76 | IOCOUNT1 | 3S-Output |
| 77 | IOCOUNT0 | 3S-Output | 78 | LOCK- | 3S-Output |
| 79 | STC- | 3S-Output | 80 | BGACK- | 2S-Output |
| 81 | BREQ- | 2S-Output | 82 | tribus | Control for 3S-Outputs |
| 83 | BGRANT- | Input | 84 | RETRY- | Input |
| 85 | BERR- | Input | 86 | HOLD- | Input |
| 87 | DTACK- | Input | 88 | INT2 | Input |
| 89 | INT1 | Input | 90 | INT0 | Input |

The tristate control bits tridata and tribus are control the tristating of the output side of the data pins and output pins, respectively. A 1 tristates and a 0 enables.

Position 1, the RESET- bit, is closest to TDI. Position 90, the INT0 bit, is closest to TDO.

*10.3.4 Identification Register (ID)*

See Section 2.4.4 for a description of the Identificaiton Register (ID). The ID register is accessable through both the TAP and normal register access.

## 11. APPENDIX

**TABLE 11-1.** One-Parcel Instruction Encodings, Monadics/Dyadics

| opcode<2:0>→ opcode<4:3>↓ | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| 00 | KCALL | CALL | stack‡ | JMP | JMPFN | JMPFY | JMPTN | JMPTY |
| 01 | unimp* | unimp* | MOV.WS | nil† | unimp* | ADD3.WS | AND3.CS | AND.SS |
| 10 | CMPEQ.CS | CMPGT.SS | CMPGT.CS | CMPEQ.SS | ADD.CS | ADD3.CS | ADD.SS | ADD3.SS |
| 11 | MOV.SS | MOV.IS | MOV.SI | MOV.II | MOV.CS | MOVA.SS | SHL3.CS | SHR3.CS |

‡: see Table 11-2.
*: the unimplemented instruction sequence is performed
†: see Table 11-3.

op.XY: X = src, Y = dst
C: 5 bit immediate
I: 5 bit indirect stack offset
S: 5 bit stack offset
W: 5 bit word aligned immediate

**TABLE 11-2.** One-Parcel Instruction Encodings, Stack

| subcode<1:0>→ | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| | ENTER | CATCH | RETURN | POPN |

**TABLE 11-3.** One-Parcel Instruction Encodings, Niladics

| subcode<2:0>→ subcode<9:3>↓ | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| 0000000 | CPU | KRET | NOP | FLUSHI | FLUSHP | CRET | FLUSHD* | unimp* |
| 0000001 | TESTV | TESTC | CLRE | unimp* | unimp* | unimp* | unimp* | unimp* |
| 000001x | unimp* | unimp* | unimp* | unimp* | unimp* | unimp* | unimp* | unimp* |
| 00001xx | unimp* | unimp* | unimp* | unimp* | unimp* | unimp* | unimp* | unimp* |
| 0001xxx | unimp* | unimp* | unimp* | unimp* | unimp* | unimp* | unimp* | unimp* |
| 001xxxx | unimp* | unimp* | unimp* | unimp* | unimp* | unimp* | unimp* | unimp* |
| 01xxxxx | unimp* | unimp* | unimp* | unimp* | unimp* | unimp* | unimp* | unimp* |
| 1xxxxxx | trap† | trap† | trap† | trap† | trap† | trap† | trap† | trap† |

*: the unimplemented instruction sequence is performed
†: the niladic trap through VB + 8

**TABLE 11-4.** Three-Parcel Instruction Encodings

| opcode<2:0>→ opcode<5:3>↓ | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| 000 | monadic† | ORI | ANDI | ADDI | MOVA | UREM | MOV | DQM |
| 001 | FNEXT* | FSCALB* | unimp* | FREM* | TADD | TSUB | unimp* | unimp* |
| 010 | FSQRT* | FMOV* | FLOGB* | FCLASS* | unimp* | unimp* | unimp* | unimp* |
| 011 | FCMPGE* | FCMPGT* | FCMPEQ* | FCMPEQN* | FCMPN* | CMPGT | CMPHI | CMPEQ |
| 100 | SUB | OR | AND | ADD | XOR | REM | MUL | DIV |
| 101 | FSUB* | FMUL* | FDIV* | FADD* | SHR | USHR | SHL | UDIV |
| 110 | SUB3 | OR3 | AND3 | ADD3 | XOR3 | REM3 | MUL3 | DIV3 |
| 111 | FSUB3* | FMUL3* | FDIV3* | FADD3 | SHR3 | USHR3 | SHL3 | unimp* |

*: the unimplemented instruction sequence is performed
†: see Table 11-5.

**TABLE 11-5.** Three-Parcel Instruction Monadic Subcodings

| subcode<2:0>→ subcode<3>↓ | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| 0 | KCALL | CALL | RETURN | JMP | JMPFN | JMPFY | JMPTN | JMPTY |
| 1 | CATCH | ENTER | LDRAA | FLUSHPTE | FLUSHPBE | FLUSHDCE* | unimp* | POPN |

*: the unimplemented instruction sequence is performed

**TABLE 11-6.** Five-Parcel Instruction Encodings

| opcode<2:0>→ opcode<5:3>↓ | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| 000 | unimp* | ORI | ANDI | ADDI | MOVA | UREM | MOV | DQM |
| 001 | FNEXT* | FSCALB* | unimp* | FREM* | TADD | TSUB | unimp* | unimp* |
| 010 | FSQRT* | FMOV* | FLOGB* | FCLASS* | unimp* | unimp* | unimp* | unimp* |
| 011 | FCMPGE* | FCMPGT* | FCMPEQ* | FCMPEQN* | FCMPN* | CMPGT | CMPHI | CMPEQ |
| 100 | SUB | OR | AND | ADD | XOR | REM | MUL | DIV |
| 101 | FSUB* | FMUL* | FDIV* | FADD* | SHR | USHR | SHL | UDIV |
| 110 | SUB3 | OR3 | AND3 | ADD3 | XOR3 | REM3 | MUL3 | DIV3 |
| 111 | FSUB3* | FMUL3* | FDIV3* | FADD3* | SHR3 | USHR3 | SHL3 | unimp* |

*: the unimplemented instruction sequence is performed

**TABLE 11-7.** General Addressing Mode Encodings

| mode | code | description |
|---|---|---|
| *$addr:B | 0x0 | byte absolute |
| *$addr:UB | 0x1 | unsigned byte absolute |
| *$addr:H | 0x2 | half-word absolute |
| *$addr:UH | 0x3 | unsigned half-word absolute |
| Roffset:B | 0x4 | byte stack offset |
| Roffset:UB | 0x5 | unsigned byte stack offset |
| Roffset:H | 0x6 | half-word stack offset |
| Roffset:UH | 0x7 | unsigned half-word stack offset |
| *Roffset:B | 0x8 | byte stack offset indirect |
| *Roffset:UB | 0x9 | unsigned byte stack offset indirect |
| *Roffset:H | 0xA | half-word stack offset indirect |
| *Roffset:UH | 0xB | unsigned half-word stack offset indirect |
| *$addr:W | 0xC | word absolute |
| Roffset:W | 0xD | word stack offset |
| *Roffset:W | 0xE | word stack offset indirect |
| $data | 0xF | immediate |

**TABLE 11-8.** Floating Point Addressing Mode Encodings

| mode | code | description |
|---|---|---|
| *$addr:F | 0x0 | single precision absolute |
| *$addr:D | 0x1 | double precision absolute |
| *$addr:X | 0x2 | extended precision absolute |
| Reserved | 0x3 | Reserved addressing mode |
| Roffset:F | 0x4 | single precision stack offset |
| Roffset:D | 0x5 | double precision stack offset |
| Roffset:X | 0x6 | extended precision stack offset |
| Reserved | 0x7 | Reserved addressing mode |
| *Roffset:F | 0x8 | single precision stack offset indirect |
| *Roffset:D | 0x9 | double precision stack offset indirect |
| *Roffset:X | 0xA | extended precision stack offset indirect |
| Reserved | 0xB | Reserved addressing mode |
| *$addr:W | 0xC | word absolute |
| Roffset:W | 0xD | word stack offset |
| *Roffset:W | 0xE | word stack offset indirect |
| $data | 0xF | single precision immediate |

**TABLE 11-9.** Call/Jmp Addressing Mode Encodings

| mode | code | description |
|---|---|---|
| **$addr | 0xC | absolute indirect |
| *Roffset | 0xD | stack offset indirect |
| Label | 0xE | program counter relative |
| *$addr | 0xF | absolute |

**TABLE 11-10.** Register Addressing Mode Encodings

| mode | code | description |
|---|---|---|
| register | 0x7 | CPU prefixed |
| *$addr:W | 0xC | word absolute |
| Roffset:W | 0xD | word stack offset |
| *Roffset:W | 0xE | word stack offset indirect |
| $data | 0xF | immediate |

**TABLE 11-11.** Register Access Codes

| register | code |
|---|---|
| MSP | 0x1 |
| ISP | 0x2 |
| SP | 0x3 |
| CONFIG | 0x4 |
| PSW | 0x5 |
| SHAD | 0x6 |
| VB | 0x7 |
| STB | 0x8 |
| FAULT | 0x9 |
| ID | 0xA |
| TIMER1 | 0xB |
| TIMER2 | 0xC |
| unimp | 0xD |
| unimp | 0xE |
| unimp | 0xF |
| FPSW | 0x10 |

**TABLE 11-12.** Exception Identifiers

| exception | code |
|---|---|
| integer zero-divide | 0x1 |
| trace | 0x2 |
| illegal instruction | 0x3 |
| alignment fault | 0x4 |
| privilege violation | 0x5 |
| unimplemented register | 0x6 |
| fetch fault | 0x7 |
| read fault | 0x8 |
| write fault | 0x9 |
| text fetch I/O bus error | 0xA |
| data access I/O bus error | 0xB |