ABACUS 3167 FLOATING-POINT COPROCESSOR

July 1990

1. Features

SINGLE-CHIP FLOATING-POINT COPROCESSOR

Used with the Intel 80386

Fits a standard EMC 121-pin socket, which is a superset of the Intel 80387 coprocessor socket

Pin-for-pin compatible with WTL 1167 floating-point coprocessor daughter board

Fully code-compatible with the WTL 1167 and ABACUS 4167 coprocessors

HIGH PERFORMANCE

5.60 single-precision MWhetstones and 1.36 single-precision MFLOPS in hand-coded LINPACK at 25 MHz

HIGH-LEVEL LANGUAGES

Supported by C, FORTRAN, and Pascal compilers under UNIX, MS-DOS protected mode, and Virtual 86 mode

IEEE FORMAT

Conforms to the IEEE standard format for floating-point arithmetic in both single and double precision (ANSI/IEEE Standard 754–1985)

FULL FUNCTION

Add, subtract, multiply, divide, and square root

Integer-to-floating-point conversions

Absolute value

Compare

Transcendental functions supported by run-time libraries

LOW-POWER CMOS

Dissipates 2.0 watts max at 25 MHz

121-pin PGA package

2. Description

The WEITEK ABACUS 3167 is a high-performance single-chip floating-point coprocessor for Intel's 80386 32-bit microprocessor. It delivers two to three times the performance of Intel's 32-bit 80387 numeric coprocessor. (Benchmark results are given in figure 1.)

The interface signals between the ABACUS 3167 and the 80386 are provided by a 121-pin socket, called the *extended math coprocessor* (EMC) socket, which is a superset of the 80387 socket. The EMC socket allows either the ABACUS 3167 or the 80387 to be installed in the system.

Both coprocessors can be used at the same time if the system board contains both an EMC socket and an 80387 socket, or if an EMC daughter board (containing a separate 80387 socket) is used.

The ABACUS 3167 is pin-for-pin compatible with the WTL 1167 floating-point coprocessor daughter board.

C, FORTRAN, and Pascal compilers fully support the ABACUS 3167, allowing programs to be written in high-level languages. The ABACUS 3167 is upwardly code-compatible with the WTL 1167 coprocessor daughter board. Also, all code written for the ABACUS 3167 coprocessor (for 80386-based PCs) will run on the ABACUS 4167 (for 80486-based systems).

The ABACUS 3167 coprocessor is a memory-mapped peripheral. From the system designer's standpoint, integrating the ABACUS 3167 into the system is as simple as adding memory at an upper address. To the 80386 and its application software, the ABACUS 3167 appears to be a segment of memory. Instructions are executed by performing memory moves to and from the coprocessor.

This data sheet is complemented by the *ABACUS Software Designer's Guide*, which provides information on BIOS support, compiler support, and assembly language programming considerations.

Benchmark	Performance
LINPACK*	
single precision	1.36 MFLOPS
double precision	0.60 MFLOPS
WHETSTONE	
single precision	5.60 MWhetstones
double precision	3.70 MWhetstones

*Hand-coded

Figure 1. Benchmark results at 25 MHz

3. Block Diagram

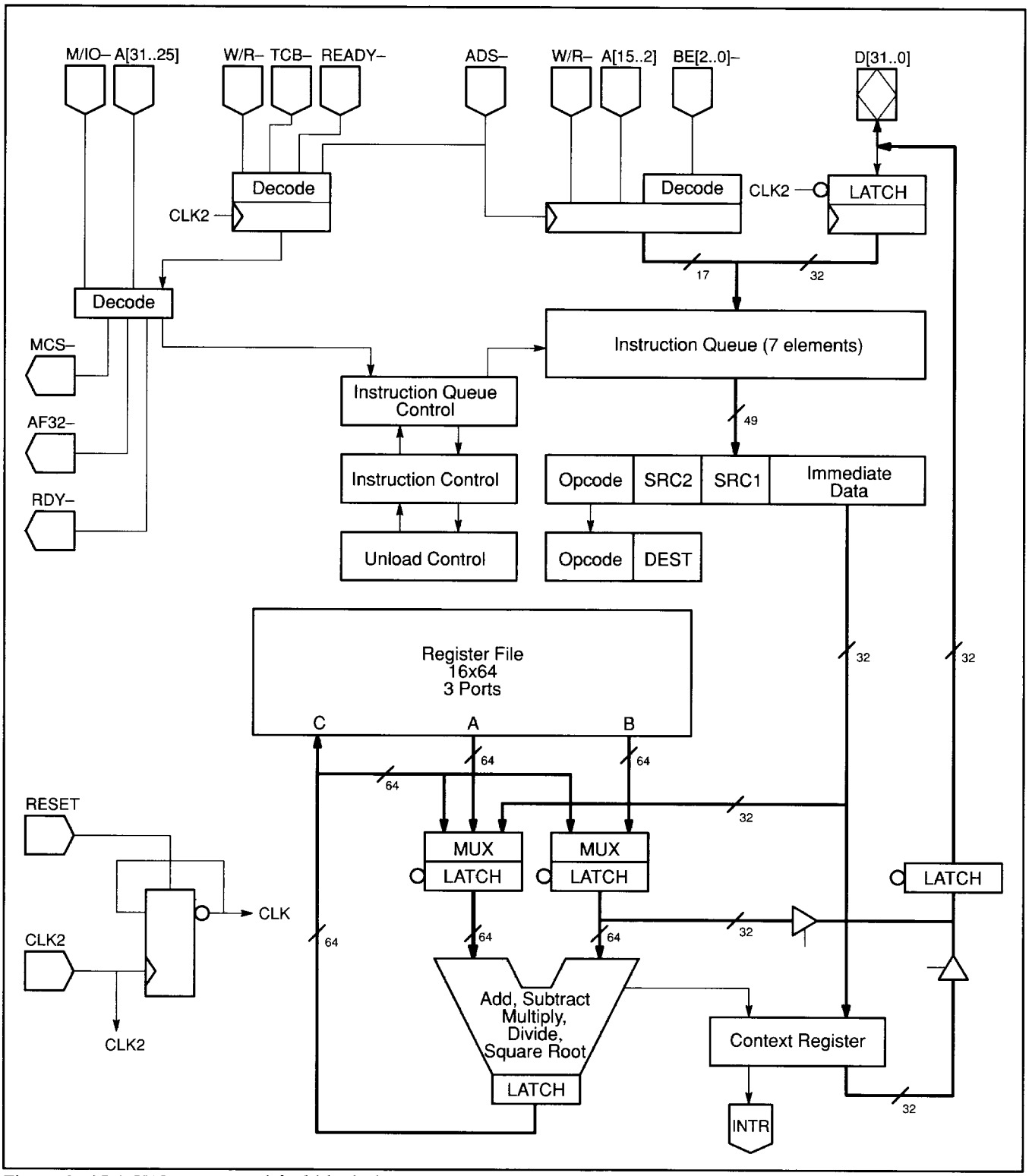


Figure 2. ABACUS 3167 simplified block diagram

4. Signal Description

4.1. BUSES

ADDRESS BUSES

The A[31..2] and BE[2..0]– *address* buses should be connected directly to the 80386 address bus and byte enables, respectively.

DATA BUS

The D[31..0] *data* bus should be connected to the 80386 data bus.

4.2. CONTROL SIGNALS

AF32–

The AF32– *32-bit bus cycle* output signal is used only in implementations based on Chips and Technologies' chip set. In such implementations, AF32– should be connected to VCC through a 10 kΩ resistor and to AF32– on the 82C301 and 82C302 devices. It should otherwise be left unconnected.

COPROCESSOR PRESENT

The PRES– *coprocessor present* signal indicates the presence of a WEITEK coprocessor. This signal should be connected to VCC through a resistor of at least 10 kΩ to ensure a high level when the WEITEK coprocessor is not present.

The hardware designer can use the PRES– output to make sure that the system generates a READY– signal when the ABACUS 3167 is addressed, but is absent (as determined by PRES– being high), in order to avoid system hangs. This allows software to detect the presence of an ABACUS 3167 in an 80386 system by attempting to load data into the coprocessor register file and read it back.

COPROCESSOR READY

The RDY– *coprocessor ready* output signal must be ORed into the logic generating READY– for the 80386, using only combinatorial logic (see figure 3). In implementations using Chips and Technologies' chip set, RDY– should be connected to READY–.

INTERRUPT

The INTR *interrupt* output signal of the ABACUS 3167 must be connected to the system interrupt controller. In AT-compatible systems, for example, the ABACUS 3167 INTR should be ORed to the 80287/80387 interrupt logic and the output should be connected to IRQ13 as shown in figure 4.

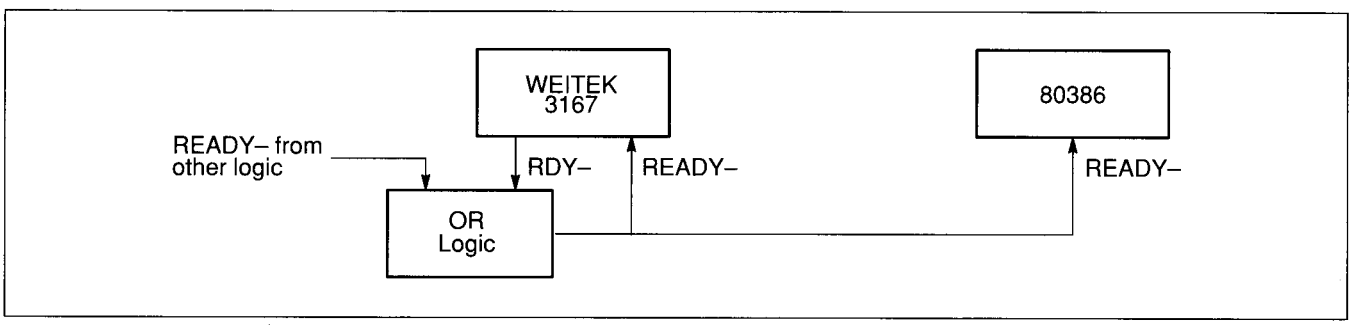


Figure 3. READY– and RDY– connection

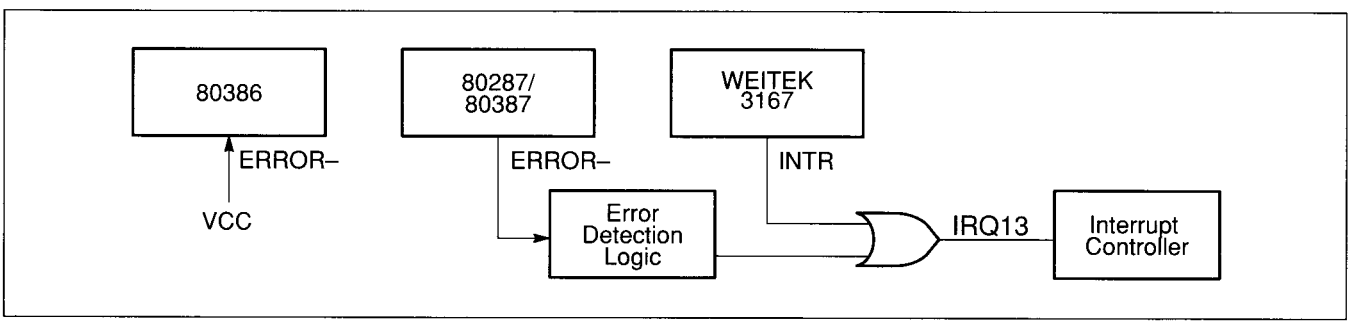


Figure 4. Interrupt output connection in an AT-compatible system

4. Signal Description, continued

MATH COPROCESSOR SELECT

The MCS– *math coprocessor select* output signal is asserted when the current address is meant for the WEITEK 3167 coprocessor. It changes when the 80386 address bus changes. MCS– may be left unconnected or may be used in conjunction with W/R– to disable other 80386 data bus drivers on ABACUS 3167 read cycles. When pipelined addressing is used, special attention must be paid, as MCS– may be deasserted prior to the end of a ABACUS 3167 read cycle, as shown in figure 49.

THREE-CYCLE BUS

The TCB– *three-cycle bus* input signal should be grounded in systems using the Chips and Technologies AT/386 chip set. Otherwise, TCB– should be left unconnected.

4.3. 80386 INTERFACE SIGNALS

The 80386 interface signals are: *address status* (ADS–), *memory I/O control* (M/IO–), *transfer acknowledge* (READY–), *reset* (RESET), and *write/read line* (W/R–).

ADS–, M/IO– READY–, RESET, and WR– should be connected to the 80386 ADS–, M/IO–, READY–, RESET, and W/R– signals, respectively. RESET must be asserted synchronously with the 80386 clock to guarantee proper operation (see figure 50). In implementations using the Chips and Technologies chip set, READY– should be connected to READY– on the 82C301 and 82C302 devices, and to VCC through a 10 kΩ pull-up resistor.

4.4. OTHER SIGNALS

CLOCK

The CLK2 *clock* signal is the clock input to the WEITEK ABACUS 3167. All ABACUS 3167 timing is relative to CLK2. This signal must be the same as CLK2 of the 80386, but the ABACUS 3167 signal should have a dedicated trace.

GROUND

All GND *ground* pins for the ABACUS 3167 must be connected to the same ground plane.

NO CONNECTION

All NC *no connection* pins must be left unconnected.

VCC

All VCC *5-volt (+5.0 V) power supply* pins for the ABACUS 3167 must be connected to system VCC.

4.5. UNUSED SIGNALS

BUSY–, CKM, CMD0, ERROR–, PEREQ, READYO–, STEN, 387 CLK2, and pins L4 and M10 are not used by the ABACUS 3167; they are used by the 80387 only. See the 80387 data sheet for details. Such signals can be left unconnected if the ABACUS 3167 is the only coprocessor in use.

5. Operation

The WEITEK ABACUS 3167 coprocessor is a memory-mapped peripheral that communicates with the 80386 over the same address bus that connects the main memory to the CPU. Instructions are defined by the 14 least-significant address bits (A[15..2]) as well as three of the four byte enables (BE[2..0]–).

The seven most-significant bits of the 80386 address bus (A[31..25]), together with the memory I/O control signal (M/IO–), select the ABACUS 3167 coprocessor. Only the upper seven address bits are decoded to determine when a coprocessor operation is being requested.

The coprocessor responds to memory addresses C0000000 through C1FFFFFF (hexadecimal). Although by convention only addresses C0000000 to C000FFFF (hexadecimal) are used, it is important to be sure that other components in the system do not conflict with the address space decoded by the coprocessor. Writing to this address space causes the ABACUS 3167 to execute instructions and reading this space causes the coprocessor to drive the data bus.

5.1. MEMORY MAPPING

A given address in the range C0000000 to C000FFFF selects the coprocessor, indicates the instruction the ABACUS 3167 is to perform, and specifies the location of Source1 and Source2/Destination. Figure 5 shows how the ABACUS 3167 views a 32-bit address word.

5. Operation, continued

COPROCESSOR SELECT

The most-significant 16 bits of the physical address identify a coprocessor instruction. If the upper bits do not fall in the C0000000–C1FF0000 range, the address does not specify a WEITEK command and is then ignored by the ABACUS 3167. To ensure compatibility with future devices, you must set the coprocessor select field to C0000000 when specifying a ABACUS 3167 instruction.

OPCODE FIELD

The next six bits specify the coprocessor instruction to be executed. Figure 6 provides the binary and hexadecimal offset, the hexadecimal number obtained by placing the six opcode bits into the opcode field of the address, for the ABACUS 3167 instructions.

OPERAND FIELDS

The five bits of the Source1 and Source2/Destination fields identify the registers that provide sources and destinations for the instruction. If Source1 is set to zero, the Source1 data is moved over the system data bus. To take advantage of the 80386 block-move instruction (refer to Block Moves on page 7), the Source1 field is split into three-bit and two-bit fields. The two-bit field occupies the two least-significant bits of the address.

INSTRUCTION ENCODING

Figures 6 and 7 show the coprocessor instruction mnemonics and the encoding for instruction opcode and operands (Source1 and Source2/destination).

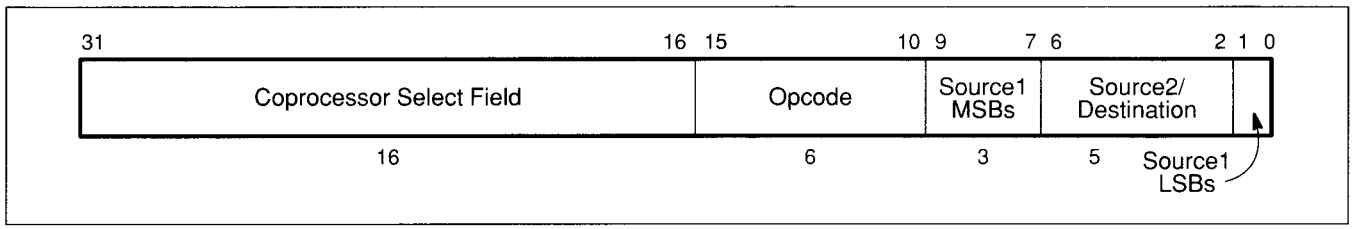


Figure 5. ABACUS 3167 view of 80386 address word

Opcode Mnemonic	Binary Value A[15..10]	Hexadecimal Offset	Opcode Mnemonic	Binary Value A[15..10]	Hexadecimal Offset
ADD.S*	000000	0000	ADD.D	100000	8000
LOAD.S	000001	0400	LOAD.D	100001	8400
MUL.S	000010	0800	MUL.D	100010	8800
STOR.S	000011	0C00	STOR.D	100011	8C00
SUBR.S	000100	1000	SUBR.D	100100	9000
DIV.S	000101	1400	DIV.D	100101	9400
MULN.S	000110	1800	MULN.D	100110	9800
FLOAT.S	000111	1C00	FLOAT.D	100111	9C00
CMPT.S	001000	2000	CMPT.D	101000	A000
TSTT.S	001001	2400	TSTT.D	101001	A400
NEG.S	001010	2800	NEG.D	101010	A800
ABS.S	001011	2C00	ABS.D	101011	AC00
CMP.S	001100	3000	CMP.D	101100	B000
TST.S	001101	3400	TST.D	101101	B400
AMUL.S	001110	3800	AMUL.D	101110	B800
FIX.S	001111	3C00	FIX.D	101111	BC00
CVTS.D	010000	4000	LDCTX	110000	C000
CVTD.S	010001	4400	STCTX	110001	C400
MAC.S	010010	4800	MACD.S	110010	C800
SQRT.S	010011	4C00	SQRT.D	110011	CC00
MACD.D	010100	5000	LOADD.D	110100	D000
SUB.S	010101	5400	STORD.D	110100	D000
			SUB.D	110101	D400

*.S in the opcode field stands for single precision while .D stands for double precision.

Figure 6. Opcode encoding

5. Operation, continued

Source1 Register	Decimal Value	Hexadecimal Offset	Source2/Dest. Register	Decimal Value	Hexadecimal Offset
F0	0	00	T0	0	00
F1	1	01	T1	1	04
F2	2	02	T2	2	08
F3	3	03	T3	3	0C
F4	4	80	T4	4	10
F5	5	81	T5	5	14
F6	6	82	T6	6	18
F7	7	83	T7	7	1C
F8	8	100	T8	8	20
F9	9	101	T9	9	24
F10	10	102	T10	10	28
F11	11	103	T11	11	2C
F12	12	180	T12	12	30
F13	13	181	T13	13	34
F14	14	182	T14	14	38
F15	15	183	T15	15	3C
F16	16	200	T16	16	40
F17	17	201	T17	17	44
F18	18	202	T18	18	48
F19	19	203	T19	19	4C
F20	20	280	T20	20	50
F21	21	281	T21	21	54
F22	22	282	T22	22	58
F23	23	283	T23	23	5C
F24	24	300	T24	24	60
F25	25	301	T25	25	64
F26	26	302	T26	26	68
F27	27	303	T27	27	6C
F28	28	380	T28	28	70
F29	29	381	T29	29	74
F30	30	382	T30	30	78
F31	31	383	T31	31	7C

Figure 7. Operand encoding

GENERATING ABACUS 3167 INSTRUCTIONS WITH 80386 MEMORY MOVES

Suppose that two single-precision numbers, stored in the ABACUS 3167 registers F1 and T2, need to be added and the result stored in T2. Since the coprocessor is mapped in the memory range C0000000–C000EFFF, the instruction is specified by the following coprocessor select, opcode, and operand address fields:

COPROCESSOR SELECT =	C000 0000 hex
OPCODE = ADD.S =	0000 hex
Source1 = F1 =	01 hex
Source2/Destination = T2 =	08 hex

The 80386 address specifying the floating-point instruction is then given by:

A[31..0] = C0000009 hex

An 80386 move instruction that generates a physical address of C0000009 causes the ABACUS 3167 to execute the floating-point addition.

A LOW-LEVEL REPRESENTATION OF THE ABACUS 3167 INSTRUCTION SET

It is clear from the previous example that the single-precision WFADD instruction appears in the 80386's memory space as an array of 1,024 consecutive addresses, one for each combination of 32x32 operands. Thus, there is a natural low-level representation of ABACUS 3167 instructions as arrays of memory addresses. The array-starting location is determined by the specific opcode shown in figure 6. The elements of the array can be represented by the operand offset values provided in figure 7.

5. Operation, continued

Returning to this example, the single-precision addition instruction has the low-level mnemonic ADD.S in figure 6. If we declare ADD.S as a memory array starting at location 0C0000000, and we declare the operand offsets F1 and T2 as 01h and 08h, respectively, the coding of the ADD.S ws2, ws1 instruction becomes:

```
MOV ADD.S[T5 + F1], AL
```

It is important to notice that when Source1 is set to zero (F0), it actually specifies an operand that does not reside in the register file and instead is provided by the data bus. When Source2/Destination is set to zero (T0), it looks like any other register in the register file.

OVERLAPPED DOUBLE-WORD REFERENCES

There is a pitfall to avoid while using the low-level mnemonics to code ABACUS 3167 instructions: you will obtain incorrect results if you indiscriminately choose to access ABACUS 3167 memory with a double-word transfer when a byte-sized transfer would have sufficed. This pitfall applies to instructions involving only ABACUS 3167 registers, and not any data on the 80386 bus, as in the previous example.

For example, suppose you want to provide a low-level encoding for AMUL.S ws8,ws14. In this instruction, the 80386 data bus is ignored by the ABACUS 3167. So any memory access to the address AMUL.S[T8 + F14] causes the multiplication to be performed. The instruction with the shortest encoding is:

```
MOV AMUL.S[T8 + F14], AL
```

To understand the pitfall, look what happens if you instead code an unnecessary double-word memory access:

```
MOV AMUL.S[T8 + F14], EAX
```

By adding the offsets of AMUL.S, T8, and F14 to the ABACUS 3167 base address 0C0000000h, you have encoded a double-word write to memory location 0C00039A2h. The memory address is not a multiple of four, so the double-word being written is not aligned on a double-word boundary. When double-words are not aligned, the 80386 splits the memory write into two operations, as shown in figure 8. First it writes the bottom half of EAX to the top half of 0C00039A0; then it writes the top half of EAX to the bottom half of 0C00039A4. The ABACUS 3167 misinterprets this as two consecutive floating-point instructions, instead of the single AMUL.S that was intended.

BLOCK MOVES

The 80386 has an instruction, REP MOVSD, that moves a block of double-words from one memory location to another. Three 80386 registers must be initialized before REP MOVSD is executed: ECX holds the number of double-words to be moved, ESI points to the source of the move, and EDI points to the destination. (The instruction can also be executed with 8086-style addressing, using the registers CX, SI, and DI.)

The operand fields of an ABACUS 3167 address are arranged so that the address can be given as either the source or the destination at a REP MOVSD instruction. Due to the positioning of the destination operand slot two bits from the bottom of the address, T-offsets increase by four for successive ABACUS 3167 registers. The registers appear as successive double-word addresses in the 80386 memory space, allowing the REP MOVSD instruction to work correctly.

PHYSICAL VERSUS LOGICAL ADDRESSES

The 80386 has three distinct address spaces: logical, linear, and physical. A logical address consists of a selector and an offset. The segmentation unit translates the logical address space into a 32-bit linear address space. If the paging unit is not enabled, then the 32-bit linear address corresponds to the physical address. Otherwise, the paging unit translates the linear address space into the physical address space. The physical address is what appears on the address pins and is responsible for specifying ABACUS 3167 instructions. See figure 9. (For more details refer to the Intel 80386 data sheet and the *80386 Programmer's Reference Manual*). The logical-to-physical address translation is fully transparent to the applications programmer. Applications programmers need only know which logical addresses will be mapped into ABACUS 3167 physical addresses.

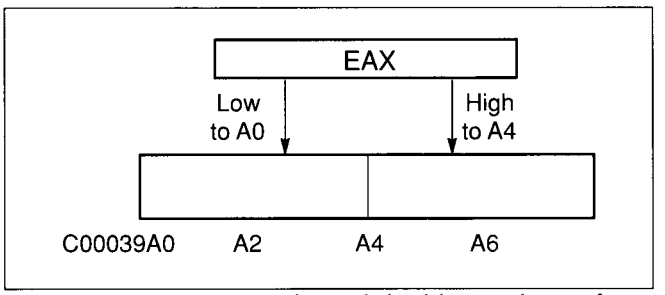


Figure 8. Erroneous overlapped double-word transfer to the ABACUS 3167 space

5. Operation, continued

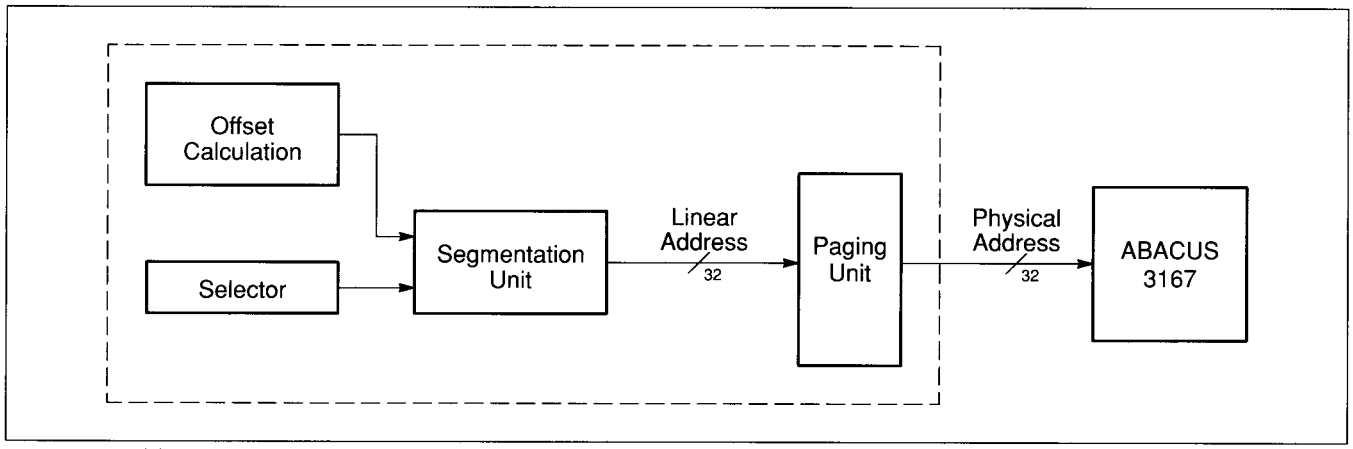


Figure 9. Address translation

EXECUTION TIMES

Use the table in figure 10 to estimate ABACUS 3167 performance. The double-precision memory-to-register estimates include a load ws1 instruction.

Figures 10 through 12 assume that new instructions are sent to the ABACUS 3167 within six cycles of the acknowledgment of a transfer by the coprocessor.

Figure 11 gives the execution times for procedures in WEITEK's transcendental function library. The exact times may vary according to the values of the operands handed to the functions; the times given in the table are average times. Transcendental routines are provided to compiler vendors with ABACUS 3167 support. Figures 12 and 13 give ASCII-float conversions.

Instruction Type	32-Bit Reg.-to-Reg.	64-Bit Reg.-to-Reg.
LOAD, Compare, ABS	3 cycles	3 cycles
ADD, SUB, NEG, Conversion	6 cycles	6 cycles
MUL	6 cycles	10 cycles
AMUL	9 cycles	13 cycles
MULN	12 cycles	16 cycles
DIV	38 cycles	66 cycles
SQRT	60 cycles	118 cycles
MAC	12 cycles	16 cycles
MACD.S	N/A	12 cycles
STORE*	3 cycles	N/A

*Source operations require a variable number of cycles since they cannot be performed if another operation is in progress.

Figure 10. Instruction latency

5. Operation, continued

Function	Single Precision	Double Precision	Absolute Accuracy	Relative Accuracy	Monotonicity	Notes
SQRT	117 cycles	285 cycles	N/A	5.00 ULPs	TDT	1, 2, 3, 4, 17
SIN	146 cycles	292 cycles	1.60 ULPs	5.00 ULPs	TDT	1, 5, 17, 18
COS	140 cycles	285 cycles	2.20 ULPs	5.00 ULPs	TDT	1, 5, 17
ATAN	157 cycles	298 cycles	3.00 ULPs	5.00 ULPs	TDT	1, 6, 17
EXP	179 cycles	401 cycles	2.20 ULPs	5.00 ULPs	TDT	1, 7, 17
LOG	171 cycles	365 cycles	2.70 ULPs	5.00 ULPs	TDT	1, 8, 17
TAN	188 cylces	340 cylces	N/A	5.00 ULPs	N/A	1, 3, 9, 10, 11, 17
COTAN	150 cycles	372 cycles	N/A	5.00 ULPs	N/A	1, 3, 9, 10, 11, 17
ASIN	175 cyces	267 cyces	N/A	5.00 ULPs	N/A	1, 3, 11, 12, 17
ACOS	175 cyces	467 cyces	N/A	5.00 ULPs	N/A	1, 3, 11, 12, 17
SINH	185 cycles	400 cycles	N/A	5.00 ULPs	N/A	1, 3, 11, 13, 14, 17
COSH	185 cycles	400 cycles	N/A	5.00 ULPs	N/A	1, 3, 11, 13, 14, 17
TANH	194 cycles	350 cycles	N/A	5.00 ULPs	N/A	1, 3, 11, 15, 17
REM	N/A	N/A	N/A	5.00 ULPs	N/A	1, 3, 6, 11, 17
MOD	N/A	N/A	N/A	5.00 ULPs	N/A	1, 3, 6, 11, 17
ASCII to BINARY	N/A	N/A	0.01 ULPs	0.01 ULPs	To 0.01 ULPs	1, 11, 16, 17
BINARY to ASCII	N/A	N/A	0.01 ULPs	0.01 ULPs	To 0.01 ULPs	1, 11, 16, 17

1. As determined by Alex Liu's "Elefunt" program
2. Square root can be implemented much faster using the SQRT instruction. The routine is used when running code written for the WTL 1167. The number shown is an average for 100,000 uniformly distributed numbers from 0 through 50,000.
3. Absolute accuracy tests do not exist for these functions.
4. TDT is an abbreviation for "to the degree tested."
5. Average for 50,000 uniformly distributed numbers from 0 through $\pi/4$, 25,000 uniformly distributed numbers from $\pi/4$ to $\pi/2$, and 25,000 uniformly distributed numbers in the range of $\pi/2$ through π.
6. Average for 100,000 uniformly distributed numbers from −1 through 1.
7. Average for 100,000 uniformly distributed numbers from −10 through 10.
8. Average for 100,000 uniformly distributed numbers from e^{-10} through e^{10}.
9. Average from 0 to 4.1×10^3
10. Average from 0 to 6.7×10^7
11. Monotonicity has yet to be determined.
12. Average from 0 to 1
13. Average from 0 to 89
14. Average from 0 to 710
15. Average from 0 to ∞
16. See figures 12 and 13
17. As determined by Cody and Waite's transcendental routines.
18. ULP is an abbreviation for "units in the last place."

Figure 11. Average execution times for transcendental functions

String	Conversion Time in Cycles 32-Bit	64-Bit	Comment
0	310	310	
1	384	416	
1.23456	704	704	6 Digits
123456.	672	672	6 Digits
123456789012345.	1120	1152	15 Digits
1234567890.12345	1088	1184	15 Digits
12345678901234567890.	1376	1696	20 Digits (opt. exp.)
1234567890.1234567890	1344	1696	20 Digits (opt. exp.)
1234567890.123456789e10	1568	1888	20 Digits (opt. exp.)
12345678901234567890.e10	1568	1856	20 Digits (opt. exp.)
1.2345678901234567890e38	1600	1856	20 Digits (opt. exp.)
12.345678901234567890e–38	1664	2048	20 Digits (opt. exp.)
1.23456e15	832	864	6 Digits
1.23456e–15	896	896	6 Digits

*Counts are plus or minus 30 cycles.

Figure 12. ASCII-to-float conversion times (cycles)

Format	Number	32-Bit	64-Bit	Format	Number	32-Bit	64-Bit
f9.2	.12345	672	704	g9.2	1e–37	1048	1512
	1	672	736		.01	1048	1088
	1234.567	736	768		.5	736	776
f17.10	.00000000001	576	576		90	744	776
	.0001	800	832		1000	1056	1160
	1.23456789	864	992		1e38	1056	1528
	12345.6	864	1056	g17.10	1e–37	1208	1760
f27.20	1e–20	768	800		1e–10	1208	1352
	1e–10	960	1088		.01	1208	1344
	1	960	1344		.5	904	1088
	12345.6	928	1440		1000	912	1040
					1e9	936	1056
e9.2	1	936	960		1e38	1232	1600
	1e10	936	1040	g27.20	1e–307	—	1936
	1e38	944	1408		1e–37	1264	1944
e17.10	1	1104	1216		.01	1264	1776
	1e10	1104	1240		.5	960	1376
	1e38	1112	1488		1000	968	1400
e27.10	1	1152	1560		1e19	984	1488
	1e10	1160	1592		1e38	1312	1768
	1e38	1168	1648		1e308	—	1968

*Counts are plus or minus 30 cycles.

Figure 13. Float-to-ASCII conversion times (cycles)

6. Software Tools Overview

Once the WEITEK ABACUS 3167 has been designed into the mother board and the ROM BIOS has been modified, the new system can take advantage of the wide selection of software tools and applications supporting the WEITEK coprocessors.

The ABACUS 3167 coprocessor is supported by the UNIX operating system (System V release 3.0). Operating system support includes coprocessor addressing, presence detection at power-up, context-switch handling, and emulation. See *ABACUS 3167 & ABACUS 4167 Questions and Answers* for a comprehensive list of compiler and operating environment vendors who support the ABACUS 3167.

The WEITEK coprocessor can be supported under protected-mode MS-DOS (using DOS extenders like Phar Lap or Eclipse) and Virtual 86 mode (using expanded memory manager drivers), as well.

C, FORTRAN, and Pascal Compilers for the 80386 and ABACUS 3167 under UNIX V.3 and MS-DOS protected mode are also available.

The ABACUS 3167 is fully transparent to the programmer using these compilers, as the floating-point operations are specified with familiar high-level language commands. The compilers include a run-time library for transcendental operations.

Operating system developments, compiler designers, and programmers who intend to write ABACUS 3167 assembly code should refer to the *ABACUS Software Designer's Guide*. Systems programmers who need to modify existing operating systems to support the ABACUS 3167 should refer to section 9 in this data book.

6.1. TRANSCENDENTAL ROUTINES LIBRARY

WEITEK provides a library of transcendental routines to compiler developers. Routines are available through a simple license agreement.

6.2. TESTING THE DESIGN

A set of diagnostic routines that test the coprocessor design for both UNIX and DOS environments, is available from WEITEK. No programming knowledge is required to run the diagnostics software. Contact your WEITEK sales representative for a copy of the diagnostics. (Refer to figure 57 on page 40 for the product's part number.)

7. Programming the ABACUS 3167

This section provides an overview to programming the WEITEK ABACUS 3167 coprocessor in 80386 assembly language. A complete programming description is given in the *ABACUS Software Designer's Guide.*

7.1. REGISTERS

The ABACUS 3167 provides a register set of 32 single-precision registers, named ws0 through ws31. Pairs of ABACUS 3167 registers can be used for double-precision operations, allowing up to 16 double-precision registers, numbered wd0, wd2, wd4... wd30. The most-significant word (MSW) is stored in the even register and the least-significant word (LSW) is stored in the next contiguous odd register (that is, MSW in wsN, LSW in wsN+1). Se figure 14. In addition, any 80386 double-word register can be used to move data, or as the source operand to an arithmetic instruction. The use of register ws0 is restricted. (Refer to section 5.1, Operand Fields on page 5 for more details on register ws0.)

7.2. PROCESS CONTEXT REGISTER

The ABACUS 3167 also provides a 32-bit process context register (PCR), which can be written to control rounding modes and exception handling. The context register can also be read to save control settings and read various status flags. The format of this register is defined in figure 15.

MODE SELECTION FIELD

The uppermost byte of the process context register includes the mode selection field and the mode field.

The mode selection field (MDSEL) is used during system initialization to set the mode register in the floating-point coprocessor. If MDSEL is set to 1100 when loading the context register, only the exception mask (EM), condition code (CC), and accumulated exception (AE) fields are updated. If MDSEL is set to 0000 the EM, CC, and AE fields, as well as the rounding mode field (MD) are updated. (See figure 16.)

MODE FIELD

The mode field (MD) is used to specify rounding options. Two bits specify one of four rounding modes as defined by the IEEE standard (RN, RZ, RP, RM). A third bit determines the rounding mode used in floating-point-to-integer conversion instructions. It selects either the current rounding mode or the round-to-zero mode. The least-significant bit of the mode field must always be set to 1.

The mode field is shown in figure 16.

ACCUMULATED EXCEPTION FIELD

The accumulated exception field (AE) contains the five exception flags required by the IEEE standard and three additional ABACUS 3167-specific exceptions. The AE field is cleared by writing zeros into the corresponding bits of byte zero of the PCR. The accumulated exception flags are formed by the logic OR of the AE field and the current instruction's exception status. The flags accumulate all exceptions that have occurred since the user last cleared the AE field. Exceptions are accumulated regardless of the value of the corresponding exception mask field.

The AE field is shown in figure 17.

CONDITION CODE FIELD

The condition code field (CC) is updated only when test or compare instructions are executed. The CC field is updated to reflect the status of the compare operation. At the end of the compare operation the coprocessor status output is encoded and stored in PCR[15..8]. The encoding is shown in figure 18.

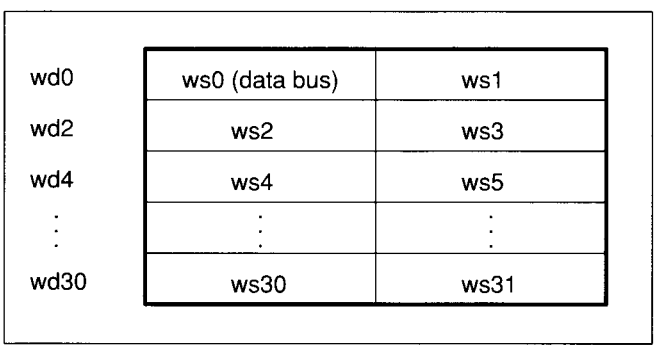


Figure 14. ABACUS 3167 register file

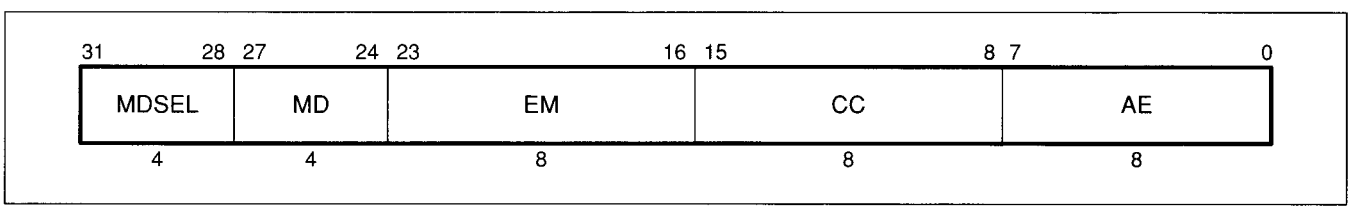


Figure 15. Process context register

7. Programming the ABACUS 3167, continued

EXCEPTION MASK FIELD

The next lower PCR byte is the exception mask field (EM). Seven bits are used to enable exception traps. At the conclusion of an instruction, the accumulated exception field is updated and, if an exception occurred and the corresponding bit in the EM field is set low, the ABACUS 3167 generates an 80386 interrupt by driving the interrupt request output high. The exception mask field is shown in figure 19.

Most of the exceptions have the same name as a corresponding 80387 exception and work the same way. The ABACUS 3167 has an undefined opcode exception, flagged whenever the instruction broadcast by the 80386 is not recognized as a WEITEK instruction. The invalid operation exception is flagged when an invalid operation occurs. The data chain exception is never flagged by the ABACUS 3167. It has been documented for consistency with the WTL 1167. (For a detailed description of ABACUS 3167 exception handling, refer to section 8.3.)

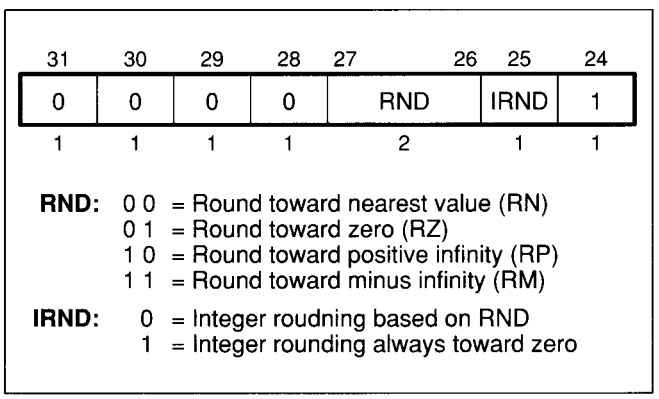


Figure 16. Mode field

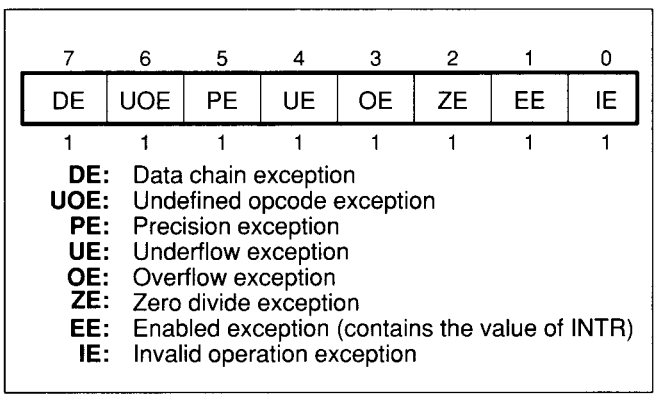


Figure 17. Accumulated exception field

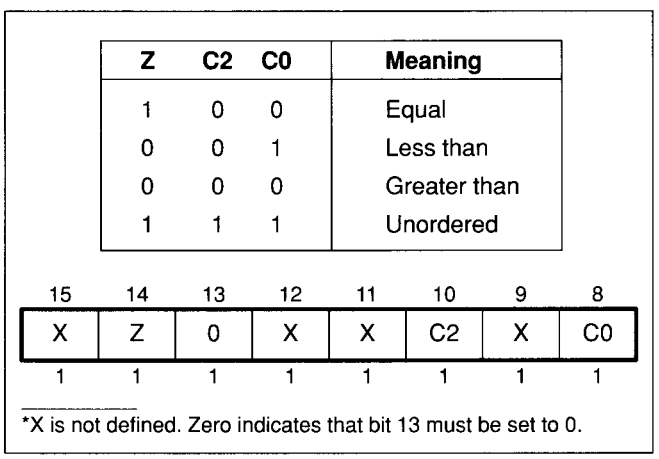


Figure 18. Condition code field

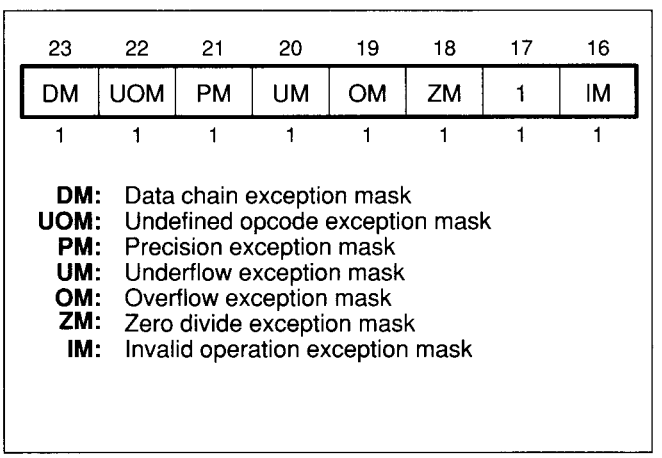


Figure 19. Exception mask field

7. Programming the ABACUS 3167, continued

7.3. MACRO INSTRUCTION SET

The remainder of this section uses WEITEK's macro assembly mnemonics to describe ABACUS operations. These macros are part of a macro definition package that runs on the Phar Lap assembler.

WEITEK ABACUS 3167 instructions can be divided into:

1. data movement instructions
2. format conversion instructions
3. arithmetic instructions
4. compare and test instructions
5. sign manipulation instructions

Most ABACUS 3167 instructions operate on either two ABACUS 3167 registers or on one ABACUS 3167 register and the contents of the 80386 data bus. WEITEK coprocessor macro definitions have the format:

```
OPCODE      Source2/Destination, Source1
```

Source1 and Source2/Destination specify the operand addresses. The operation result is always stored in the same location as Source2. While Source2/Destination always specifies one of the 32 ABACUS 3167 internal registers, Source1 can specify either an internal register (for register-to-register operations), an immediate constant, or the contents of a 80386 register (for memory-to-register operations).

DATA MOVEMENT INSTRUCTIONS

Data movement instructions move data between the 80386 and a ABACUS 3167 register, or between two ABACUS 3167 registers. See figure 20.

FORMAT CONVERSION INSTRUCTIONS

The ABACUS 3167 provides instructions for converting between the supported data types (single precision, double precision, and 32-bit integer). See figure 21.

```
WFLD      ws1, ws2          ; load ws1 from ws2
WFLD      ws21, EAX         ; load ws21 from EAX
WFLD      ws4, PI           ; load ws4 with constant PI (declared elsewhere)
WFLD      wd4, wd12         ; load ws4 from ws12, then load ws5 from ws13
WFLDCTX   EAX               ; load Context Register from EAX
WFPOP     ws1               ; pop a number from 386 stack to ws1
WFLDSD    ws1, ARRAY, 31    ; load 31 numbers from ARRAY to registers ws1 through ws31
WFLDSD    ws10, ESI, ECX    ; load ECX numbers from ESI to registers starting with ws10
WFST      EDX, ws21         ; store ws21 to EDX
WFSTCTX   EAX               ; store Context register to EAX
WFPUSH    ws1               ; push ws1 onto the 386 stack
WFSTSD    ws0, ARRAY, 32    ; store all 32 registers to ARRAY
WFSTSD    ws10, EDI, ECX    ; store ECX registers from ws10 to EDI
WFSTRL    EAX               ; store revision level to EAX
```

Figure 20. Examples of data movement instructions

```
WFLOAT    ws1, ws10         ; convert integer ws10 to single-precision ws1
WFLOAT    wd4, ws13         ; convert integer ws13 to double-precision wd4
WFLOAT    ws3, EAX          ; convert integer EAX to single-precision ws3
WFLOAT    wd6, EBX          ; convert integer EBX to double-precision wd6
WFLOAT    wd10, 123456      ; load wd10 with the constant 123456.0
WFIX      ws1, ws4          ; convert single-precision ws4 to integer ws1
WFIX      ws3, wd10         ; convert double-precision wd10 to integer ws3
WFIX      ws5, EBX          ; convert single-precision EBX to integer ws5
WFCVT     ws1, wd14         ; convert double-precision wd14 to single-precision ws1
WFCVT     ws8, EBX          ; convert double-precision (EBX, ws1) to single-precision ws8
WFCVT     wd10, ws9         ; convert single-precision ws9 to double-precision wd10
WFCVT     wd26, EAX         ; convert single-precision EAX to double-precision wd26
```

Figure 21. Examples of format conversion instructions

7. Programming the ABACUS 3167, continued

ARITHMETIC INSTRUCTIONS

The ABACUS 3167 provides the four basic arithmetic functions as well as square root. See figure 22.

In the subtraction instruction, the Source2/Destination operand is subtracted from the Source1 operand. The reverse subtraction instruction reverses the operands from the standard subtract instruction.

The division instruction divides the Source1 operand by the Source2/Destination.

The single-precision multiply/accumulate operation multiplies the operands specified by Source1 and Source2 and adds the result to the contents of register ws2.

The double-precision multiply/accumulate operation with single-precision inputs multiplies the single-precision operands specified by Source1 and Source2 and adds the result to the contents of register wd2. The double-precision multiply/accumulate operation with double-precision inputs multiplies the double-precision operands specified by Source1 and Source2 and adds the result to the contents of register wd2.

```
WFADD     ws6, ws13       ; add ws13 into ws6
WFADD     wd14, wd20      ; add wd20 into wd14
WFADD     ws3, EAX        ; add EAX into ws3
WFADD     wd2, EBX        ; add (EBX, ws1) into wd2
WFADD     ws1, 9.0        ; add the constant 9.0 into ws1
WFSUBR    ws5, ws30       ; set ws5 to ws30 – ws5
WFSUBR    wd12, wd14      ; set wd12 to wd14 – wd12
WFSUBR    ws3, EDX        ; set ws3 to EDX – ws3
WFSUB     ws5, ws30;      ; set ws5 to ws5 – ws30
WFSUB     ws8, EDX        ; set ws8 to ws8 – EDX
WFMUL     ws1, ws2        ; multiply ws2 into ws1
WFMULN    wd4, wd6        ; set wd4 to (– wd4 x wd6)
WFAMUL    ws5, EAX        ; set ws5 to the absolute value of ws5 x EAX
WFMUL     ws23, 2.0       ; multiply the constant 2.0 into ws23
WFMAC     ws10, ws11      ; add ws10 x ws11 into ws2
WFMAC     ws9, EAX        ; add ws9 x EAX into ws2
WFMACD    ws13, ws29      ; add ws13 x ws29 into wd2
WFMACD    ws1, EBP        ; add ws1 x EBP into wd2
WFMACD    wd12, wd28      ; add wd12 x wd28 into wd2
WFDIVR    ws3, ws5        ; set ws3 to ws5 ÷ ws3
WFDIVR    wd16, wd18      ; set wd16 to wd18 ÷ wd16
WFDIVR    ws2, EAX        ; set ws2 to EAX ÷ ws2
WFDIVR    ws7, PI         ; set ws2 to PI ÷ ws7
WFSQRT    ws3, ws5        ; set ws3 to SQRT(ws5)
WFSQRT    wd10, wd12      ; set wd10 to SQRT(wd12)
```

Figure 22. Examples of arithmetic instructions

7. Programming the ABACUS 3167, continued

COMPARE AND TEST INSTRUCTIONS

Compare and test instructions either compare two floating-point values or compare a single floating-point value to zero (see figure 23). The compare instructions compare Source1 to Source2.

Besides comparing the operand to zero, as does the test operation (wftst), test with trap (wftstt) generates an invalid operation exception if the operand is not a valid number (not-a-number or NaN). Test instructions always operate on Source1. Compare and test instructions affect condition code bits 14, 10, and 8 of the process context register as shown in figure 18 on page 13.

SIGN MANIPULATION INSTRUCTIONS

The ABACUS 3167 has two functions that manipulate the sign of a floating-point number: negate and absolute value. See figure 24.

```
WFCMPR    ws3, ws4      ; perform reversed comparison
WFCMPRT   wd8, wd10     ; perform reversed comparison and generate
                        ; invalid exception if one (or both) of the operands is not a
                        ; valid number
WFTST     wd4           ; perform the test of wd4
WFTSTT    ws1           ; perform the test of ws1 and generate
                        ; invalid exception if the operand is not a valid number
```

Figure 23. Examples of compare and test instructions

```
WFNEG     ws1, ws1      ; negate ws1
WFNEG     ws1, ws2      ; set ws1 to –ws2
WFNEG     wd4, wd6      ; set wd4 to –wd6
WFNEG     ws3, EAX      ; set ws3 to –EAX
WFABS     ws3, ws4      ; set ws3 to the absolute value of ws4
WFABS     wd10, wd10    ; coerce wd10 to its absolute value
```

Figure 24. Examples of sign manipulation instructions

7. Programming the ABACUS 3167, continued

7.4. PROGRAMMING EXAMPLE

The example given in figure 26 shows the code for a 4x4 matrix transformation written using the macros provided by WEITEK.

The matrix coefficients a_{11}, ..., a_{44} are assumed to be already stored in the ABACUS 3167 registers ws16..ws31. The variables x, y, z, and w are in memory locations X, Y, Z, and W. The variables x', y', z', and w' are stored back in memory location X, Y, Z, and W.

$$\begin{bmatrix} x & y & z & w \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} x_i & y_i & z_i & w_i \end{bmatrix}$$

Figure 25. Matrix multiplication

```
MOV     EAX, X          ; load x into 386 EAX register
WFLD    ws4, EAX        ; load x into ws4
MOV     EAX, Y          ; load y into 386 EAX register
WFLD    ws5, EAX        ; load y into ws5
MOV     EAX, Z          ; load z into 386 EAX register
WFLD    ws6, EAX        ; load z into ws6
MOV     EAX, W          ; load w into 386 EAX register
WFLD    ws7, EAX        ; load w into ws7
WFLD    ws2, ws16       ; move a11 into ws2
WFMUL   ws2, ws4        ; ws2 = a11 x x
WFMAC   ws17, ws5       ; ws2 = (a11 x x) + (a21 x y)
WFMAC   ws18, ws6       ; ws2 = (a11 x x) + (a21 x y) + (a31 x z)
WFMAC   ws19, ws7       ; ws2 = (a11 x x) + (a21 x y) + (a31 x z) + (a41 x w)
WFST    EAX, ws2        ; store xi
MOV     X, EAX          ; store xi into memory location X
WFLD    ws2, ws20       ; move a12 into R2
WFMUL   ws2, ws4        ; ws2 = a12 x x
WFMAC   ws21, ws5       ; ws2 = (a12 x x) + (a22 x y)
WFMAC   ws22, ws6       ; ws2 = (a12 x x) + (a22 x y) + (a32 x z)
WFMAC   ws23, ws7       ; ws2 = (a12 x x) + (a22 x y) + (a32 x z) + (a42 x w)
WFST    EAX, ws2        ; store yi
MOV     Y, EAX          ; store yi into memory location Y
WFLD    ws2, ws24       ; move a13 into ws2
WFMUL   ws2, ws4        ; ws2 = a13 x x
WFMAC   ws25, ws5       ; ws2 = (a13 x x) + (a23 x y)
WFMAC   ws26, ws6       ; ws2 = (a13 x x) + (a23 x y) + (a33 x z)
WFMAC   ws27, ws7       ; ws2 = (a13 x x) + (a23 x y) + (a33 x z) + (a43 x w)
WFST    EAX, ws2        ; store zi
MOV     Z, EAX          ; store zi into memory location Z
WFLD    ws2, ws28       ; move a14 into ws2
WFMUL   ws2, ws4        ; ws2 = a14 x x
WFMAC   ws29, ws5       ; ws2 = (a14 x x) + (a24 x y)
WFMAC   ws30, ws6       ; ws2 = (a14 x x) + (a24 x y) + (a34 x z)
WFMAC   ws31, ws7       ; ws2 = (a14 x x) + (a24 x y) + (a34 x z) + (a44 x w)
WFST    EAX, ws2        ; store wi
MOV     W, EAX          ; store wi into memory location W
```

Figure 26. 4x4 matrix transformation in assembly language

7. Programming the ABACUS 3167, continued

7.5. MACRO INSTRUCTION SUMMARY

Figures 27 and 28 summarize the ABACUS 3167 instruction set macros. All ABACUS 3167 register names begin with "w." We follow the "w" with either "s" for single, "d" for double, or "x" meaning either "s" or "d." The register name ends with the letter "t" or "f." "T" stands for "to" and "f" stands for "from." For most instructions, wxt is the destination register and wxf is the source register.

```
Data Movement
WFLD      wst, wsf          ; load: wst = wsf
WFLD      wst, data         ; load: wst = 386 data
WFLD      wdt, wdf          ; load: wdt = wdf
WFLDCTX   ereg              ; load: CTX = 386 E-register
WFPOP     wst               ; pop wst from the 386 stack
WFPOP     wdt               ; pop two double-words from the 386 stack to wdt
WFLDSD    wst, addr, count  ; block move: wst array = 386 memory

WFST      ereg, wst         ; store: 386 E-register = wst
WFST      ereg, wst, opcode ; store: 386 ereg = ereg <opcode> wst
WFSTCTX   ereg              ; store: 386 E-register = CTX
WFSTCTX   ereg, opcode      ; store: 386 ereg = ereg <opcode> CTX
WFPUSH    wst               ; push wst onto the 386 stack
WFPUSH    wdt               ; push wdt (two double-words) onto the 386 stack
WFSTSD    wst, addr, count  ; block move: 386 memory = wst array
WFSTRL    EAX               ; store revision level to EAX

Format Conversion
WFLOAT    wxt, wsf          ; convert integer wsf to floating wxt
WFLOAT    wxt, data         ; convert integer 386 data to floating wxt
WFIX      wst, wxf          ; convert floating wxf to integer wst
WFIX      wst, data         ; convert floating (386 data) to integer wst

WFCVT     wst, wdf          ; convert wdf to wst
WFCVT     wst, data         ; convert double-precision (386 data and ws1) to wst
WFCVT     wdt, wsf          ; convert wsf to wdt
WFCVT     wdt, data         ; convert single-precision 386 data to wdt
```

Figure 27. The ABACUS 3167 instruction set macros

7. Programming the ABACUS 3167, continued

Four-Function Arithmetic

```
WFADD    wxt, wxf        ; add: wxt = wxt + wxf
WFADD    wxt, data       ; add: wxt = wxt + (386 data)

WFSUBR   wxt, wxf        ; reversed subtract: wxt = wxf – wxt
WFSUBR   wxt, data       ; reversed subtract: wxt = (386 data) – wxt
WFSUB*   wxt, wxf        ; subtract: wxt = wxt – wxf
WFSUB*   wxt, data       ; subtract: wxt = wxt – (386 data)

WFMUL    wxt, wxf        ; multiply: wxt = wxt x wxf
WFMUL    wxt, data       ; multiply: wxt = wxt x (386 data)
WFMULN   wxt, wxf        ; negative multiply: wxt = –wxt x wxf
WFMULN   wxt, data       ; negative multiply: wxt = –wxt x (386 data)
WFAMUL   wxt, wxf        ; absolute multiply: wxt = |wxt x wxf|
WFAMUL   wxt, data       ; absolute multiply: wxt = |wxt x (386 data)|

WFMAC    wst, wsf        ; multiply and accumulate: ws2 = ws2 + wst x wsf
WFMAC    wst, data       ; multiply and accumulate: ws2 = ws2 + wst x (386 data)
WFMACD*  wst, wsf        ; multiply and accumulate: wd2 = wd2 + wst x wsf
WFMACD*  wst, data       ; multiply and accumulate: wd2 = wd2 + wst x (386 data)
WFMACD*  wdt, wdf        ; multiply and accumulate: wd2 = wd2 x wdf

WFDIVR   wxt, wxf        ; reversed divide: wxt = wxf x wxt
WFDIVR   wxt, data       ; reversed divide: wxt = (386 data)/wxt
WFSQRT*  wxt, wxf        ; square root: wxt = sqrt(wxf)
WFSQRT*  wxt, data       ; square root: wxt = sqrt(data)
```

Compare and Test

```
WFCMPR   wxt, wxf;       ; reversed compare: set CTX flags for (wxf – wxt)
WFCMPR   wxt, data       ; reversed compare: set CTX for (386 data) – wxt
WFCMPRT  wxt, wxf        ; reversed compare with trap: set CTX flags for (wxf – wxt)
WFCMPRT  wxt, data       ; reversed compare with trap: set CTX for (386 data) – wxt

WFTST    wxf             ; test: set CTX flags for (wxf – 0)
WFTST    data            ; test: set CTX flags for (386 data) – 0
WFTST    ata, ws1        ; test: set CTX flags for double-precision (386 data, ws1) – 0
WFTSTT   wxf             ; test with trap: set CTX flags for (wxf – 0)
WFTSTT   data            ; test with trap: set CTX flags for (386 data) – 0
WFTSTT   data, ws1       ; test with trap: set CTX flags for (386 data, ws1) – 0
```

Sign Manipulation

```
WFNEG    wxt, wxf        ; negate: wxt = –wxf
WFNEG    wxt, data       ; negate: wxt = –(386 data)
WFABS    wxt, wxf        ; absolute value: wxt = |wxf|
WFABS    wxt, data       ; absolute value: wxt = |386 data|
```

Paging Directives

```
WFSPAGE                  ; force next wfld/wfst to single-precision page
WFDPAGE                  ; force next wfld/wfst to double-precision page
```

*These instructions are not available on the WTL 1167

Figure 28. The ABACUS 3167 instruction set macros, continued

8. IEEE Considerations

8.1. DATA TYPES

The WEITEK ABACUS 3167 floating-point coprocessor provides compatibility with the formats specified in IEEE Standard 754, Version 10.0. Several number types are required to implement the standard. The types supported by the ABACUS 3167 are described below.

NORMALIZED NUMBERS

Most calculations are performed on normalized numbers (NRMs). Single-precision normalized numbers have an exponent that ranges from binary 00000001 to binary 11111110 (1 to 254) and a normalized fraction field (the leftmost or hidden bit is a 1). In decimal notation, this allows you to represent a range of both positive and negative numbers from roughly 10^{+38} to 10^{-38} with accuracy to seven decimal places. Double-precision numbers have an exponent ranging from one to 2,046 and a normalized fraction field.

INFINITY

Infinity (INF) has an exponent of all ones and a fraction field equal to zero. Both positive and negative infinity are allowed.

ZERO

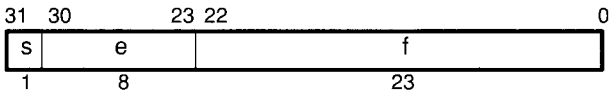
ZERO has an exponent of zero, a hidden bit equal to zero, and a value of zero in the fraction field. Both +0 and –0 are supported.

NOT A NUMBER

Not-a-number (NaN) is a special data format usually used as a flag for data-flow control, for uninitialized variables, or to signify an invalid operation such as zero times infinity. The format for a NaN is an exponent of all ones and a non-zero fraction.

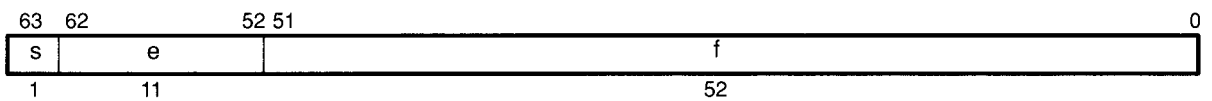
DENORMALIZED NUMBERS

Denormalized numbers (DNRMs) have a zero exponent and a denormalized (hidden bit equal to zero) non-zero fraction field. They represent numbers smaller than 2^{-127} or 2^{-1023} (single precision and double precision, respectively).



e	f	Value	Name
255	Not 0	None	NaN
255	0	$(-1)^s$ x infinity	Infinity
1..254	Any	$(-1)^s$ x 23^{e-127} x (1.f)	NRM
0	0	$(-1)^s$ x 0	Zero

Figure 29. Single-precision IEEE data types



e	f	Value	Name
2,047	Not 0	None	NaN
2,047	0	$(-1)^s$ x infinity	Infinity
1..2,046	Any	$(-1)^s$ x 23^{e-1023} x (1.f)	NRM
0	0	$(-1)^s$ x 0	Zero

Figure 30. Double-precision IEEE data types

8. IEEE Considerations, continued

8.2. ROUNDING OPTIONS

The WEITEK ABACUS 3167 supports all four rounding modes of the IEEE standard: round to nearest, round toward zero, round toward plus infinity, and round toward minus infinity. Rounding may be biased or unbiased. Biased rounding introduces a small offset in the direction of the bias. Positive bias, negative bias, and a bias toward zero are specified in the IEEE format. Unbiased rounding rounds the result to the nearest representable number. In the case of a number exactly halfway between two representable numbers, the number is rounded toward the closest even number, resulting in half of the numbers rounding up and half rounding down, on average.

ROUND TO NEAREST

Round-toward-nearest (RN) rounds the result to the nearest representable value. If two numbers are equally near the result, the even number is chosen.

ROUND TOWARD ZERO

Round-toward-zero (RZ) rounds the result to the value closest to but not greater than the magnitude of the result.

ROUND TOWARD PLUS INFINITY

Round-toward-plus-infinity (RP) rounds the result to the value closest to but not less than the result.

ROUND TOWARD MINUS INFINITY

Round-to-minus-infinity (RM) rounds the result to the value closest to but not greater than the result.

8.3. EXCEPTION HANDLING

While the IEEE floating-point formats are supported by the ABACUS 3167, some features of the IEEE standard are not provided due to the design focus on high speed.

The occurrence of an enabled exception causes an interrupt. Due to extensive instruction overlapping, the exact location of an exception is not maintained. In debugging, it is possible to identify the instruction that caused the exception by performing a store context operation after every floating-point instruction and then testing the enabled exception bit.

The following exceptions are flagged by the ABACUS 3167.

1. *Undefined Opcode Exception.* Whenever an illegal opcode is detected, the undefined opcode exception (UOE) is set. On a read bus operation, for example, only store opcodes are allowed. If a read bus operation specifies any other instruction, such as MUL.S, then the undefined opcode exception bit is set.
2. *Precision Exception.* The precision exception (PE) flag of the accumulated exception field is set whenever there is a loss of accuracy. The coprocessor data paths compute results to higher precision than the number of mantissa bits that appear in the result. If any of the fraction bits less than the least-significant bit was equal to one prior to rounding, then the PE bit is set high. The precision exception is also signaled if there is a partial or complete loss of significance in a float-to-fixed operation.
3. *Overflow Exception.* An overflow exception (OE) is generated when the result of a floating-point operation overflows the largest representable number. The result produced at the output is either infinity or the largest representable positive or negative number, depending upon the rounding mode as shown in figure 31.

   Overflow is also generated when converting float-to-fixed operations and the result overflows the 32-bit format.
4. *Underflow Exception.* When the result of an operation after rounding is less than the minimum normalized number in the destination format, an underflow exception (UE) is asserted and the result is flushed to zero. A result of exactly zero does not underflow.
5. *Zero Divide Exception.* The ABACUS 3167 asserts a zero divide exception (ZE) when performing division on a normalized dividend and a zero divisor. The result is a properly signed infinity.
6. *Invalid Operation Exception.* An invalid exception (IE) is asserted if a NaN input or an invalid operation occurs. The invalid ABACUS 3167 operations are $\infty \times 0$, $0/0$, ∞/∞, subtraction of like infinities ($\infty - \infty$), and addition of opposite infinities $\infty + (-\infty)$. The result of any invalid operation is a NaN with a fraction and an exponent of all ones. The sign bit is zero.

Condition	Result
RM or RZ and the result is positive	Largest positive normalized number
RP or RZ and the result is negative	Largest negative normalized number
RN or RP and the result is positive	Plus infinity
RN or RM and the result is negative	Minus infinity

Figure 31. Results returned on overflow

8. IEEE Considerations, continued

8.4. FAST MODE

The ABACUS 3167 always operates in Fast mode: denormalized inputs to either the multiplier or ALU as well as denormalized outputs are flushed to zero. The minimum normalized number has an exponent of one and a fraction field of zero. ZERO has an exponent of zero and a fraction field of all zeros. This allows you to represent numbers between the smallest normalized number and zero. These numbers are known as denormalized numbers (DNRM). Since DNRMs are very close to zero, most applications can substitute zero for a DNRM without a significant loss of accuracy.

8.5. OPERATION STATUS AND RESULT

Figures 32 through 36 show the results of combinations of input data formats and rounding options. The format used in these tables is: (status) result. When OK is indicated for the status, no exception is flagged.

Figure 34 shows the compare status for different input combinations; the compare status is encoded in the condition code field of the PCR.

Source1	Source2				
	ZERO	**DNRM**	**NRM**	**INF**	**NaN**
NaN	(IE) NaN	(IE) NaN	(IE) NaN	(IE) NaN	(IE) NaN
INF	(OK) INF	(OK) INF	(OK) INF	(OK) INF (1) (IE) NaN (2)	(IE) NaN
NRM	(OK) NRM	(OK) NRM	(OE) (4) (OK) NRM (UE) ZERO (OK) ZERO	(OK) INF	(IE) NaN
DNRM	(OK) ZERO (3)	(OK) ZERO	(OK) NRM	(OK) INF	(IE) NaN
ZERO	(OK) ZERO (3)	(OK) ZERO (3)	(OK) NRM	(OK) INF	(IE) NaN

1. +INF+INF → +INF
   −INF−INF → −INF
2. +INF−INF NaN (invalid operation)
   −INF+INF → NaN (invalid operation)
3. +ZERO+ZERO → +ZERO (RN,RZ,RP,RM)
   −ZERO−ZERO → −ZERO (RN,RZ,RP,RM)
   +ZERO−ZERO → +ZERO (RN,RZ,RP)
   +ZERO−ZERO → −ZERO (RM)
   −ZERO+ZERO → +ZERO (RN,RZ,RP)
   −ZERO+ZERO → −ZERO (RM)
4. OVF will produce INF or maximum normalized number (MAX.NRM), depending upon the rounding mode:
   +MAX.NRM IF [(RM,RZ) AND (RESULT IS +)]
   −MAX.NRM IF [(RP,RZ) AND (RESULT IS −)]
   +INF IF [(RN,RP) AND (RESULT IS +)]
   −INF IF [(RN,RM) AND (RESULT IS −)]

Figure 32. Status and result output for add and subtract

8. IEEE Considerations, continued

Source1	Source2				
	ZERO	DNRM	NRM	INF	NaN
NaN	(IE) NaN	(IE) NaN	(IE) NaN	(IE) NaN	(IE) NaN
INF	(OK) INF	(OK) INF	(OK) INF	(IE) NaN	(IE) NaN
NRM	(ZE) INF	(ZE) INF	(OE) (4) (OK) NRM (UE) ZERO	(OK) ZERO	(IE) NaN
DNRM	(IE) NaN	(IE) NaN	(OK) ZERO	(OK) ZERO	(IE) NaN
ZERO	(IE) NaN	(IE) NaN	(OK) ZERO	(OK) ZERO	(IE) NaN

Figure 33. Operation status and result output for divide

Source1	Source2							
	NaN	–INF	–NRM	–DNRM	ZERO	+DNRM	+NRM	+INF
NaN	U	U	U	U	U	U	U	U
+INF	U	G	G	G	G	G	G	E
+NRM	U	G	G	G	G	G	0, 1, 2	L
+DNRM	U	G	G	E	E	E	L	L
ZERO	U	G	G	E	E	E	L	L
–DNRM	U	G	G	E	E	E	L	L
–NRM	U	G	0, 1, 2	L	L	L	L	L
–INF	U	E	L	L	L	L	L	L

U : Unordered
E : Source1 = Source2
L : Source1 < Source2
G : Source1 > Source2
0, 1, 2 may be: Source1 = Source2, Source1 < Source2, or Source1 > Source2

Figure 34. Status for floating-point compare

Source1	Source2/Destination	Status	Comments
7FFFFFFF	41DFFFFF FFC00000	OK	Largest positive integer
00000001	3FF00000 00000000	OK	+1
00000000	00000000 00000000	OK	ZERO
FFFFFFFF	BFF00000 00000000	OK	–1
80000000	C1E00000 00000000	OK	Largest negative integer

Figure 35. Integer to double-precision conversions (I32→F64)

Source1	Source2/Destination	Status	Comments
7FFFFFFF	4F000000	OK	Largest positive integer
7FFFFFC0	4F000000	PE	Inexact
7FFFFF80	4EFFFFFF	OK	Exact
00000001	3F800000	OK	+1
00000000	00000000	OK	ZERO
FFFFFFFF	BF800000	OK	–1
80000080	CEFFFFFF	OK	Exact
80000040	CF000000	PE	Inexact
80000000	CF000000	OK	Largest negative integer

Figure 36. Integer to single-precision conversions (I32→F32)

9. Systems Programming for the ABACUS 3167

The system software is responsible for:

1. mapping logical addresses to the physical address space of the ABACUS 3167
2. detecting the presence of the WEITEK coprocessor
3. handling exceptions
4. saving the coprocessor registers when switching between tasks
5. emulating the device when it is not present

These tasks are described in the *ABACUS Software Designer's Guide.*

In MS-DOS, only address mapping and presence detection need be performed.

10. ROM BIOS Support

Simply adding the WEITEK ABACUS 3167 coprocessor socket to the system mother board is not sufficient to offer complete support for the WEITEK coprocessor.

PC manufacturers must also provide a presence-detection routine, residing in the system ROM BIOS. The routine detects the presence of the WEITEK coprocessor during the power-on self-test and modifies the interrupt 11H service routine so that bit 24 of the value returned in register EAX is set if the ABACUS 3167 coprocessor is present. Existing MS-DOS applications use the interrupt mechanism to determine whether the WEITEK coprocessor is present in the system.

10.1. MODIFYING THE ROM BIOS

ROM BIOS programmers can use several detection routines. If the hardware designer has connected the PRES– signal to an I/O port, presence detection can be accomplished by simply reading from such an I/O port. Otherwise the ROM BIOS programmer can use a software sequence that loads a WEITEK coprocessor register with a specific data pattern and then reads it back. To access the WEITEK coprocessor, the ROM BIOS programmer must first go into protected mode and set up page tables to address the ABACUS 3167.

A code fragment that implements the detection routine is presented in figure 37. It assumes that the system is in protected mode and page tables have been set up to address the WEITEK coprocessor.

```
; WEITEK ABACUS Presence Detection.
;
; This routine assumes the FS selector is pointing to the base of the
; WEITEK ABACUS physical address space. If desired, C0000000h may be
; added to the offsets rather than using the FS selector.
;
; Returns with EAX=1 if the ABACUS is present, else EAX=0.
;
; This routine is set up for protected mode.
; The definition for WTLSEG must be changed for other modes

WTLSEG    segment    use 16 public
W_OPS     equ         FS:0
WTLSEG    ends

ASSUME FS:WTLSEG

wtkpres proc near
          push       ebx
          push       ecx
          push       edx
          push       esi                                  ;
;
; Save contents of memory being tested in case the ABACUS is not present.
;
          cli                                   ;hold interrupts a bit...
          mov        ecx, W_OPS[0404h]          ; save original values, just in case.
          mov        edx, W_OPS[0408h]
          mov        esi, W_OPS[2809h]

;
; Try negate as a test since we must presume that the ABACUS is not initialized yet.
;
          mov        eax,40000000h                        ; 2.0
          mov        W_OPS[0404h], eax          ; wfld   ws1, eax
          mov        W_OPS[0408h], eax          ; wfld   ws2, eax
          mov        W_OPS[2809h], al           ; wfneg  ws2, ws1
          or         eax, (1 shl 31)            ;
          mov        ebx, W_OPS[0808h]          ; wfst   ebx, ws2
          cmp        eax, ebx                   ;
          mov        eax, 1                     ; return EAX=1  (ABACUS present)
          je         wtkexit                    ; ABACUS is present!!!  Take off.

;
; If not present, restore the memory that may have changed.
;
          mov        W_OPS[0404h], ecx
          mov        W_OPS[0408h], edx
          mov        W_OPS[2809h], esi
          mov        eax, 0                     ; return EAX=0  (ABACUS not present)

wtkexit label  near
          sti                         ; let the interrupts go...
          pop        esi
          pop        edx
          pop        ecx
          pop        ebx
          ret
wtkpres endp
```

Figure 37. Test for the presence of the WEITEK ABACUS 3167 coprocessor

11. Debugging the System

Once the coprocessor has been designed into the mother board and the ROM BIOS have been modified, the system is ready to be debugged.

WEITEK supplies diagnostics and demo software both in the UNIX and DOS environments.

The diagnostics software tests for coprocessor presence by calling interrupt 11H, then initializes and exercises the coprocessor.

12. WTL 1167 Compatibility

This section describes the differences between the ABACUS 3167 and the WTL 1167 coprocessor daughter board.

12.1. HARDWARE COMPATIBILITY

The single-chip ABACUS 3167s is pin-for-pin compatible with the WTL 1167 and fits into the standard 121-pin extended math coprocessor (EMC) socket. Hardware developers can offer the option of using both the ABACUS 3167 and 80387 coprocessors in their system by featuring two separate sockets on the mother board.

12.2. APPLICATION SOFTWARE COMPATIBILITY

The ABACUS 3167 is upwardly object-code-compatible with the WTL 1167. The application programs all of the software tools available for the WTL 1167 daughter board will run as is on the ABACUS 3167 — the ABACUS 3167 simply responds in less time.

The ABACUS 3167 features some new instructions — square root, reverse subtract, and double-precision multiply accumulate — that trigger an invalid opcode exception (UOE) if used with the WTL 1167.

12.3. SYSTEM SOFTWARE COMPATIBILITY

Initialization, presence detection, addressing, exception handling, context switching, and coprocessor emulation for the ABACUS 3167 are identical to same operations for the WTL 1167. Therefore, the ABACUS 3167 works in all the operation system environments that support the WTL 1167.

13. Specifications

Parameter	Value
Supply voltage	–0.5 V to 7.0V
Input voltage	–0.5V to V_{CC} + 0.5
Output voltage	–0.5V to V_{CC}
Storage temperature range	–65° C to 150° C

Figure 38. Absolute maximum ratings

Parameter	Commercial Min	Commercial Max	Unit
V_{CC} Supply voltage	4.75	5.25	V
T_{CASE} Operating temperature	0	85	° C

Figure 39. Recommended operating conditions

14. DC Electrical Characteristics

Parameter		Test Conditions	Commercial Min	Commercial Max	Unit
V_{IH}	High-level input voltage	V_{CC} = MAX	2.00		V
V_{IL}	Low-level input voltage	V_{CC} = MIN		0.80	V
V_{IHC}	CLK2 input high voltage	V_{CC} = MIN/MAX	V_{CC}–1.05	V_{CC}–0.50	V
V_{ILC}	CLK2 input low voltage	V_{CC} = MIN		0.80	V
V_{OH}	High-level output voltage	V_{CC} = MIN, I_{OH} = –2.0 mA	2.40		V
V_{OL}	Low-level output voltage	V_{CC} = MIN, I_{OL} = 4.0 mA		0.40	V
I_{IHC}	CLK2 high-level input current	V_{CC} = MAX, V_{IN} = V_{CC}		±10	µA
I_{ILC}	CLK2 low-level input current	V_{CC} = MAX, V_{IN} = 0V		±10	µA
I_{IH}	High-level input current	V_{CC} = MAX, V_{IN} = V_{CC}		±10	µA
I_{IL}	Low-level input current	V_{CC} = MAX, V_{IN} = 0V		±10	µA
I_{CC}	Supply current (33 MHz)	V_{CC} = MAX		450	mA
I_{CC}	Supply current (20 & 25 MHz)	V_{CC} = MAX		350	mA
C_{INC}	CLK2 input capacitance (33 MHz)	f = 1 MHz, V_{CC} = MAX, T_{AMB} = 25° C		30	pF
C_{INC}	CLK2 input capacitance (20 & 25 MHz)	f = 1 MHz, V_{CC} = MAX, T_{AMB} = 25° C		47	pF
C_{IN}	Input capacitance	f = 1 MHz, V_{CC} = MAX, T_{AMB} = 25° C		15	pF
C_{OUT}	Output capacitance	f = 1 MHz, V_{CC} = MAX, T_{AMB} = 25° C		15	pF

WARNING! Remove power before insertion or removal.

Figure 40. DC electrical characteristics over recommended temperature range

15. AC Switching Characteristics

Symbol	Test Conditions	3167–020		3167–025		3167–033		Unit	Note
		Min	Max	Min	Max	Min	Max		
$2T_{CY}$	Clock cycle time	50.0		40.0		30.0		ns	
T_{CY}	CLK2 period	25.0		20.0		15.0		ns	
T_{CHa}	CLK2 high time	8.0		7.5		6.7		ns	At 2V
T_{CHb}	CLK2 high time	7.0		6.5		5.5		ns	At 2.5V
T_{CLa}	CLK2 low time	8.0		7.5		6.7		ns	At 2V
T_{CLb}	CLK2 low time	7.0		6.5		5.5		ns	At 0.8V
T_R	Clock rise time		8.0		7.0		4.0	ns	
T_F	Clock fall time		8.0		7.0		4.0	ns	
T_1	ADS–, W/R– setup time	17.0		15.0		13.0		ns	1, 3, 5
T_2	ADS– hold time	4.0		4.0		4.0		ns	1, 3
T_3	A[15..2], BE[2..0]– setup time	18.0		15.0		13.0		ns	1, 3, 5
T_4	A[15..2], BE[2..0]– hold time	4.0		4.0		4.0		ns	1, 3, 6
T_5	M/IO–, A[31..25] setup time	8.0		7.0		7.0		ns	1, 3, 5
T_6	M/IO–, A[31..25] hold time	4.0		4.0		4.0		ns	1, 3, 6
T_7	D[31..0] setup time	20.0		20.0		20.0		ns	1, 3, 4, 5
T_8	D[31..0] hold time	2.0		2.0		2.0		ns	1, 3, 4, 6
T_9	READY– setup time	12.0		9.0		7.0		ns	1, 3
T_{10}	READY– hold time	4.0		4.0		4.0		ns	1, 3
T_{11}	D[31..0] output delay		38.0		30.0		25.0	ns	1, 3, 12
T_{12}	D[31..0] valid output	6.0		5.0		5.0		ns	1, 3
T_{13}	D[31..0] float delay		27.0		22.0		20.0	ns	1, 3, 13
T_{14}	RESET setup time	12.0		10.0		8.0		ns	1, 3
T_{15}	RESET hold time	4.0		3.0		3.0		ns	1, 3

Functional operating range: VCC = 5V ± 5%; T_{CASE} = 0° C to 85° C

1. All parameters are specified at 1.5V unless otherwise noted.
2. All output delays are specified at 1.5V with 50 pF loading unless otherwise specified.
3. Relative to CLK2 rising edge.
4. Write bus cycle only.
5. Referenced to the end of the cycle when ADS– is deasserted.
6. Hold time referenced to the end of the cycle when ADS– is asserted.
7. Delay is measured with respect to M/IO–, A[31..25].
8. One CLK2 period.
9. Spec applies only when TCB– is tied high.
10. Spec applies only when TCB– is tied low.
11. 85 pF loading.
12. 120 pF loading.
13. Tri-state timing is guaranteed, but not tested.

Figure 41. AC switching characteristics

15. AC Switching Characteristics, continued

Symbol	Test Conditions	3167–020		3167–025		3167–033		Unit	Note
		Min	Max	Min	Max	Min	Max		
T_{16}	INTR output delay		62.0		50		40.0	ns	1, 3
T_{17}	INTR valid output	6.0		5.0		5.0		ns	1, 3
T_{18}	MCS– output delay		20.0		17		17.0	ns	1, 7
T_{19}	RDY– output delay		24.0		22		15.0	ns	1, 3, 9, 11
T_{20}	RDY– valid output	4.0		4.0		4.0		ns	1, 3, 9, 11
T_{21}	RDY– tri-state to valid		25.0		20		20.0	ns	3, 10, 11, 13
T_{22}	RDY– tri-state delay	5.0	25.0	5.0	20	5.0	20.0	ns	3, 10, 11, 13
T_{23}	AF32– tri-state to valid		25.0		20		20.0	ns	3, 10, 13
T_{24}	AF32– tri-state delay		25.0		20		20.0	ns	3, 10, 13
T_{25}	AF32– output delay		25.0		20		20.0	ns	3, 10
T_{26}	AF32– valid output	3.0		2.0		2.0		ns	3, 10

Functional operating range: VCC = 5V ± 5%; T_{CASE} = 0° C to 85° C

1. All parameters are specified at 1.5V unless otherwise noted.
2. All output delays are specified at 1.5V with 50 pF loading unless otherwise specified.
3. Relative to CLK2 rising edge.
4. Write bus cycle only.
5. Referenced to the end of the cycle when ADS– is deasserted.
6. Hold time referenced to the end of the cycle when ADS– is asserted.
7. Delay is measured with respect to M/IO–, A[31..25].
8. One CLK2 period.
9. Spec applies only when TCB– is tied high.
10. Spec applies only when TCB– is tied low.
11. 85 pF loading.
12. 120 pF loading.
13. Tri-state timing is guaranteed, but not tested.

Figure 42. AC switching characteristics, continued

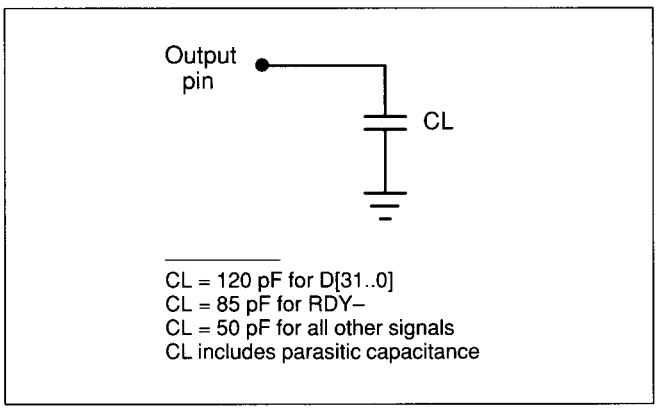


Figure 43. Test load for delay measurement

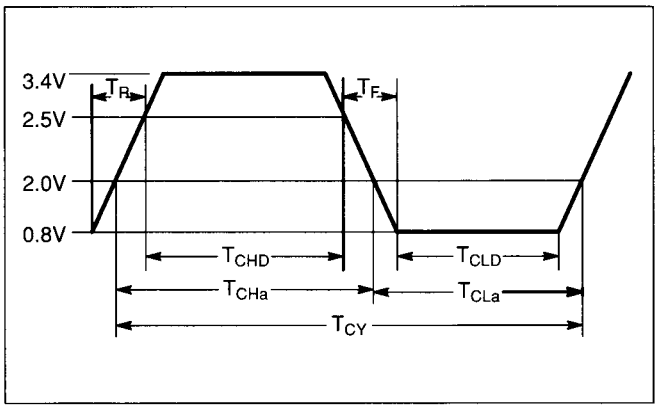


Figure 44. CLK2 timing diagram

16. Timing Diagrams

16.1. NON-PIPELINED BUS TIMING

Figure 45 shows two WEITEK ABACUS 3167 write cycles with TCB– either tied high or not connected. Write cycles are performed every time the 80386 broadcasts instructions to the coprocessor. The RDY– output of the ABACUS 3167 handles the handshaking between the coprocessor and the 80386. To acknowledge the current bus cycle, the ABACUS 3167 asserts RDY– and the 80386 terminates the bus cycle. The first bus write operation does not have the READY– input delayed, while the second operation does. In the delayed READY– write operation, even though the bus does not advance and D[31..0] is held constant, it is latched in the same cycle it would be if READY– were not delayed. Thus, if the data changes in the time slots indicated with crosshatching in figure 45, the new data is not used by the ABACUS 3167.

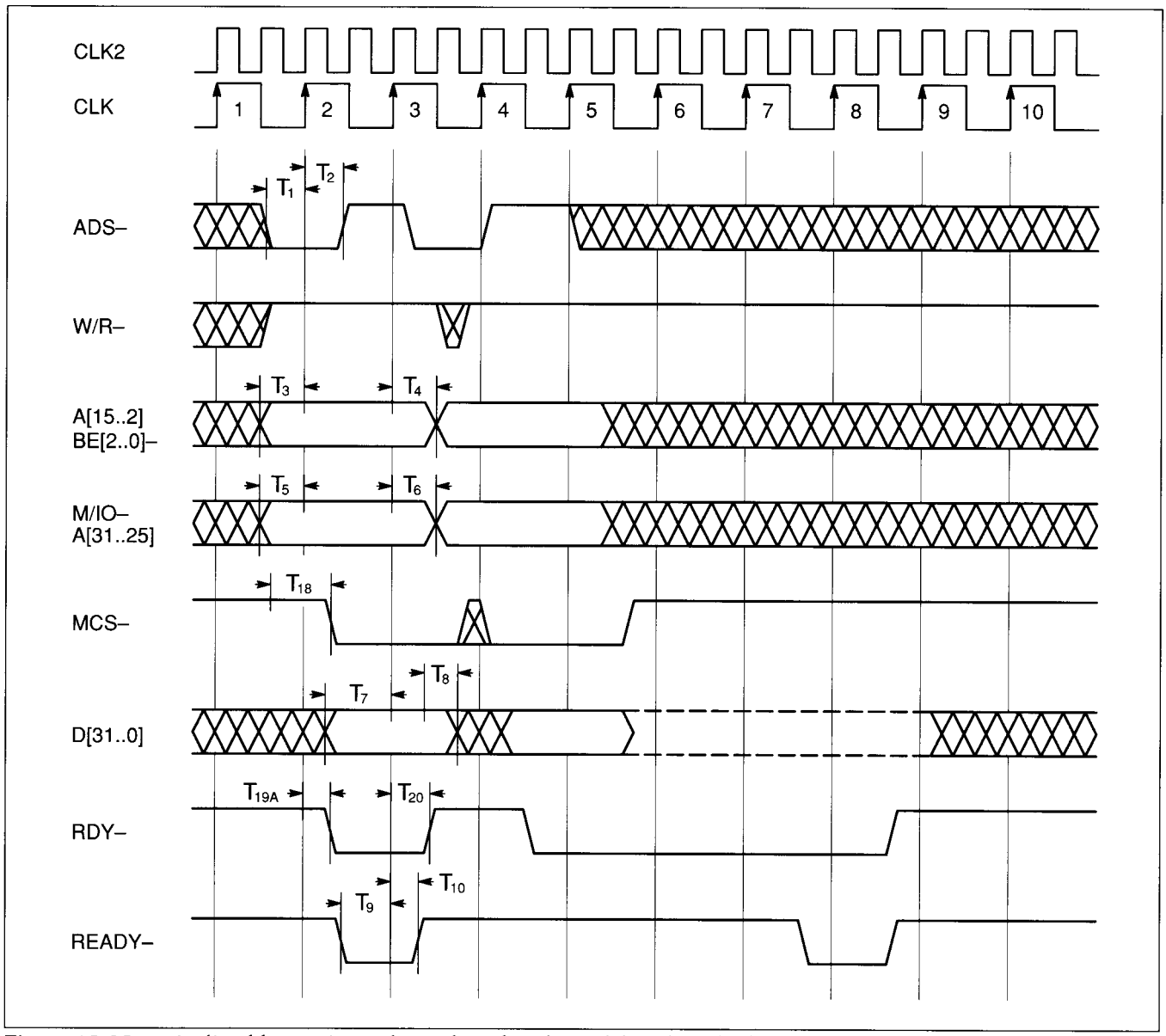


Figure 45. Non-pipelined bus write cycle, with and without delayed ready, for TCB– tied high (or not connected)

16. Timing Diagrams, continued

Figure 46 shows a new pipelined bus write cycle with TCB– tied low. The bus cycle now takes a minimum of three clock cycles.

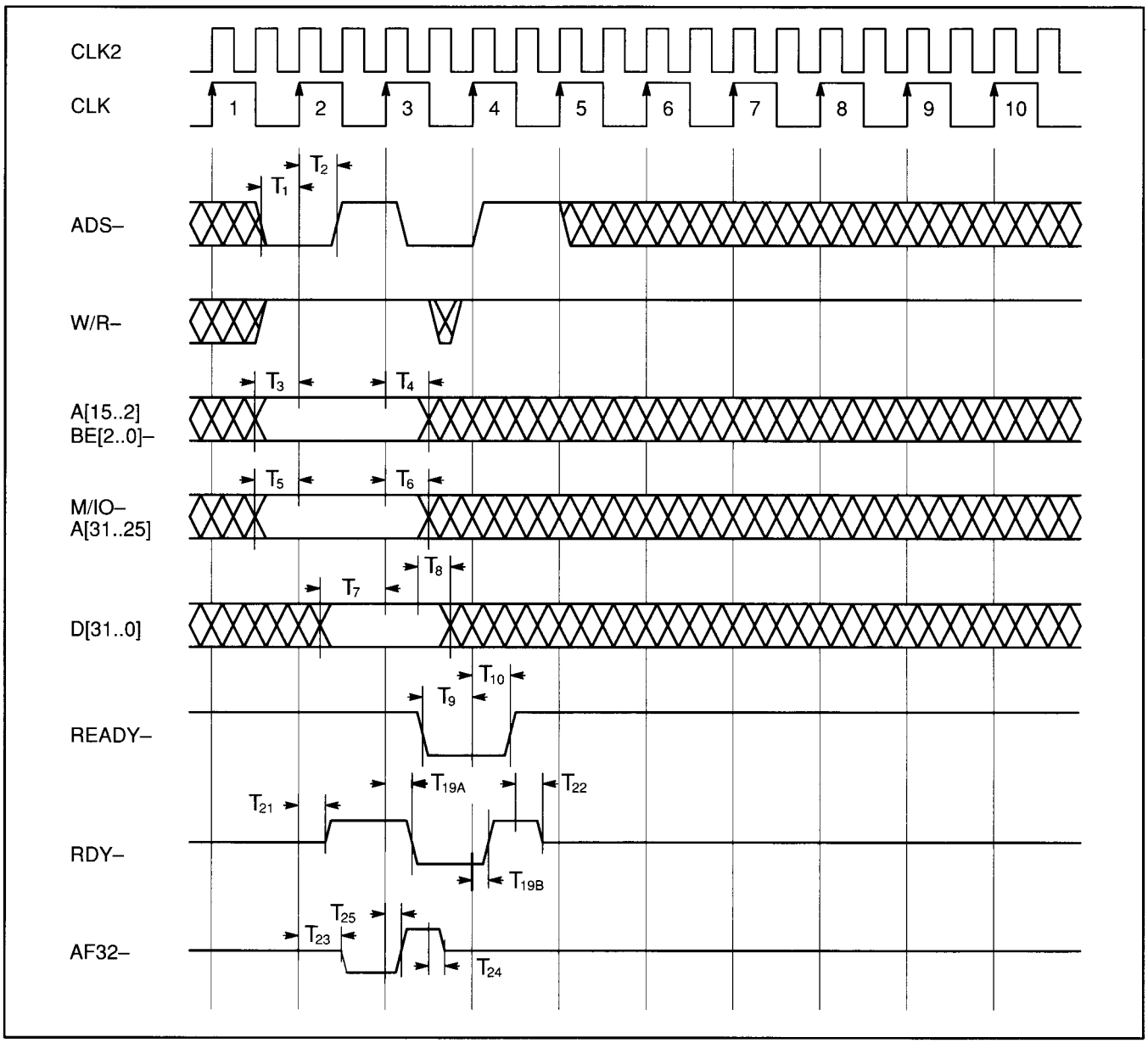


Figure 46. Non-pipelined bus write cycle without delayed ready for TCB– tied low

16. Timing Diagrams, continued

Read cycles are performed every time data must be read from the ABACUS 3167 into the 80386. Figure 47 shows a typical ABACUS 3167 read cycle. At least one wait state is always inserted during a read cycle to allow the ABACUS 3167 time to respond. As shown in figure 47, the minimum read cycle takes the same number of clock cycles, independent of the value of TCB–.

Wait states are fully transparent to the programmer. If READY– is delayed, the data continues to be driven until READY– is asserted. When the ABACUS 3167 receives a bus read operation, it turns on its bus drivers even before the data is ready. Valid data is only present when RDY– is asserted.

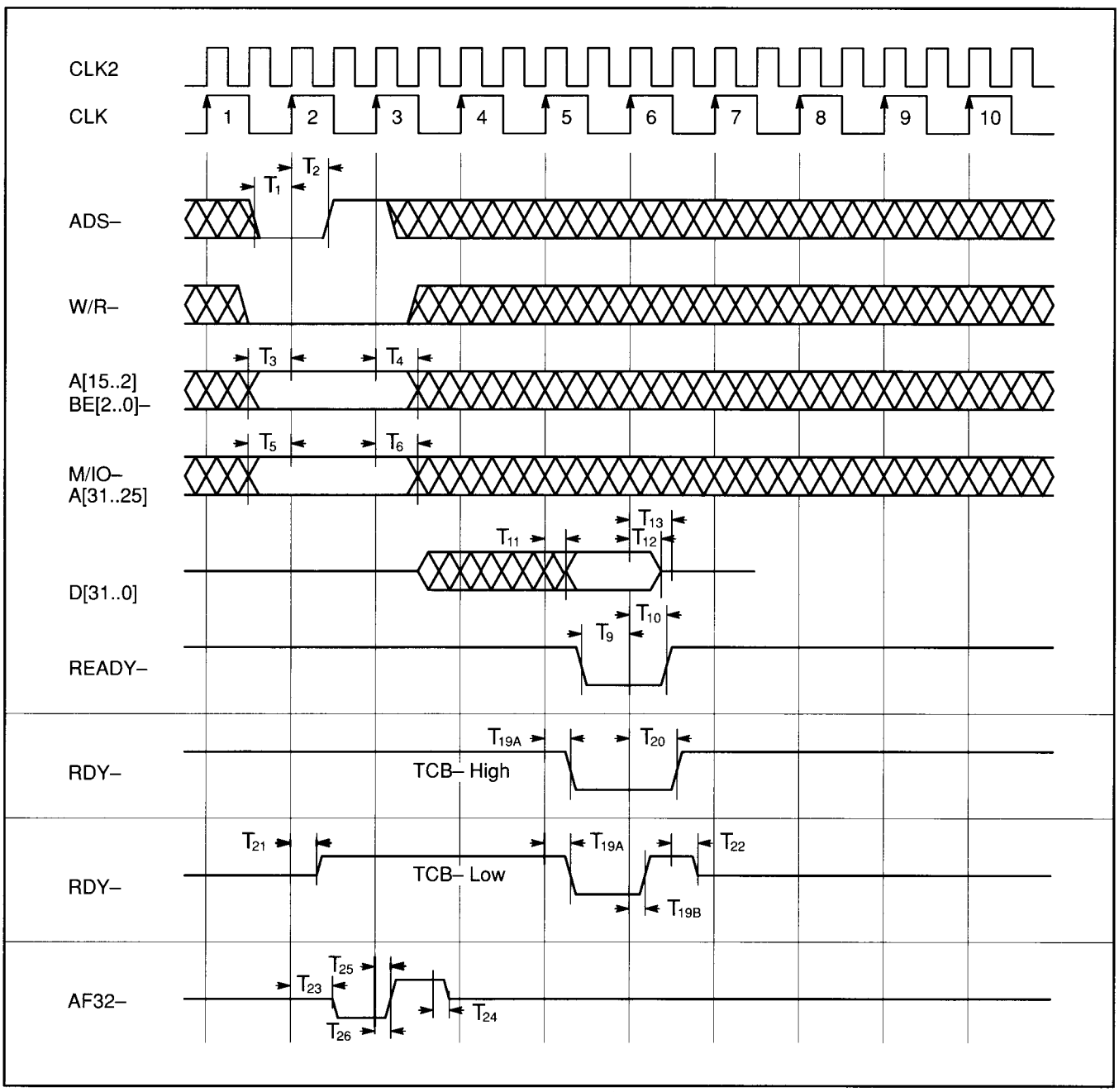


Figure 47. Non-pipelined bus read cycle without delayed ready for TCB– tied high and TCB– tied low

16. Timing Diagrams, continued

16.2. PIPELINED BUS TIMING

The WEITEK ABACUS 3167 is capable of operating with or without address pipelining on a cycle-by-cycle basis. It uses ADS– and READY– from the 80386 to determine when a bus cycle begins. Figure 48 shows a typical pipelined write cycle. The minimum write bus cycle takes the same number of clock cycles, independent of the value of TCB–.

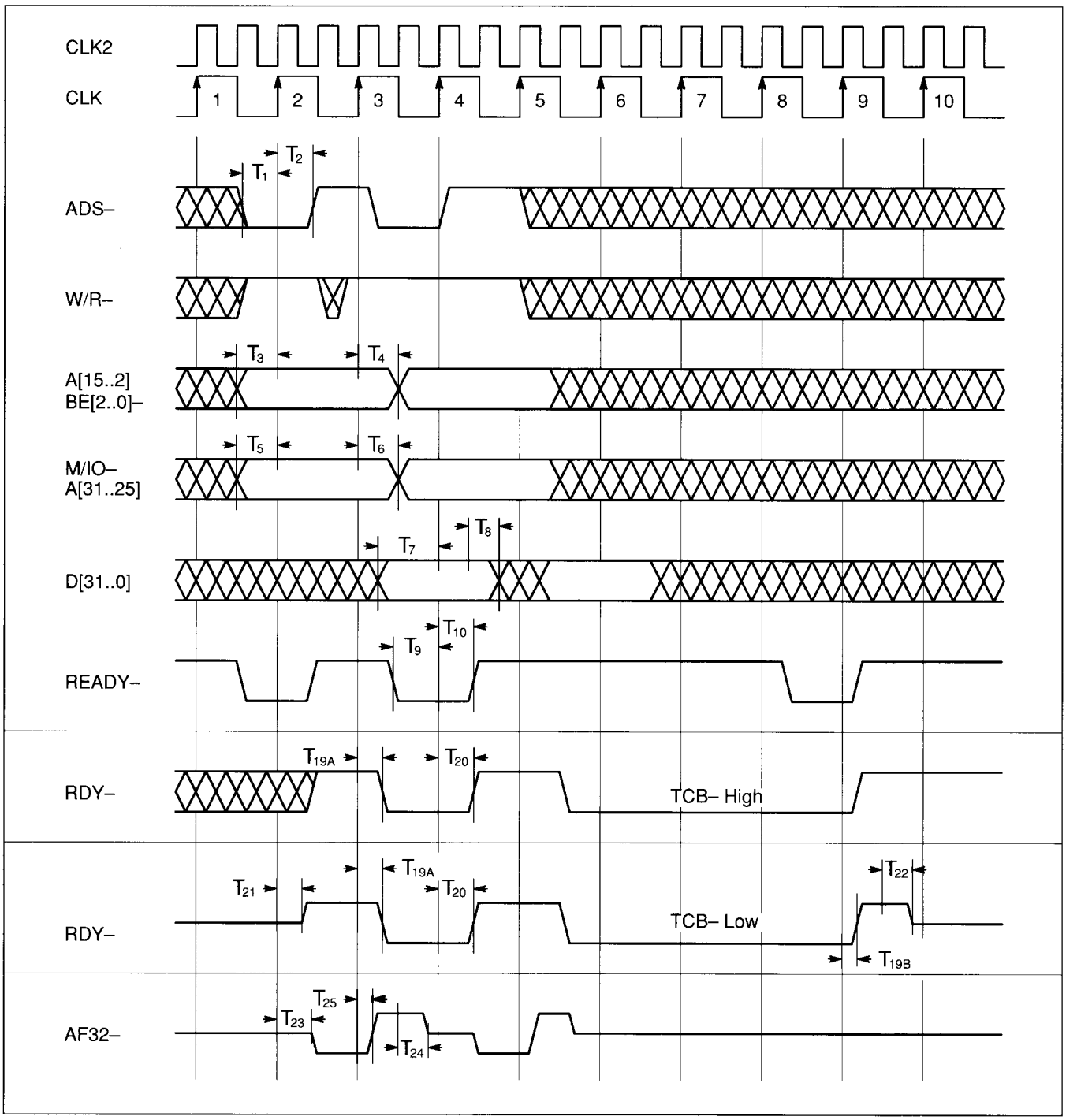


Figure 48. Pipelined bus write cycle, with and without delayed ready, for TCB– tied high and TCB– tied low

16. Timing Diagrams, continued

Figure 49 shows a typical pipelined read cycle.

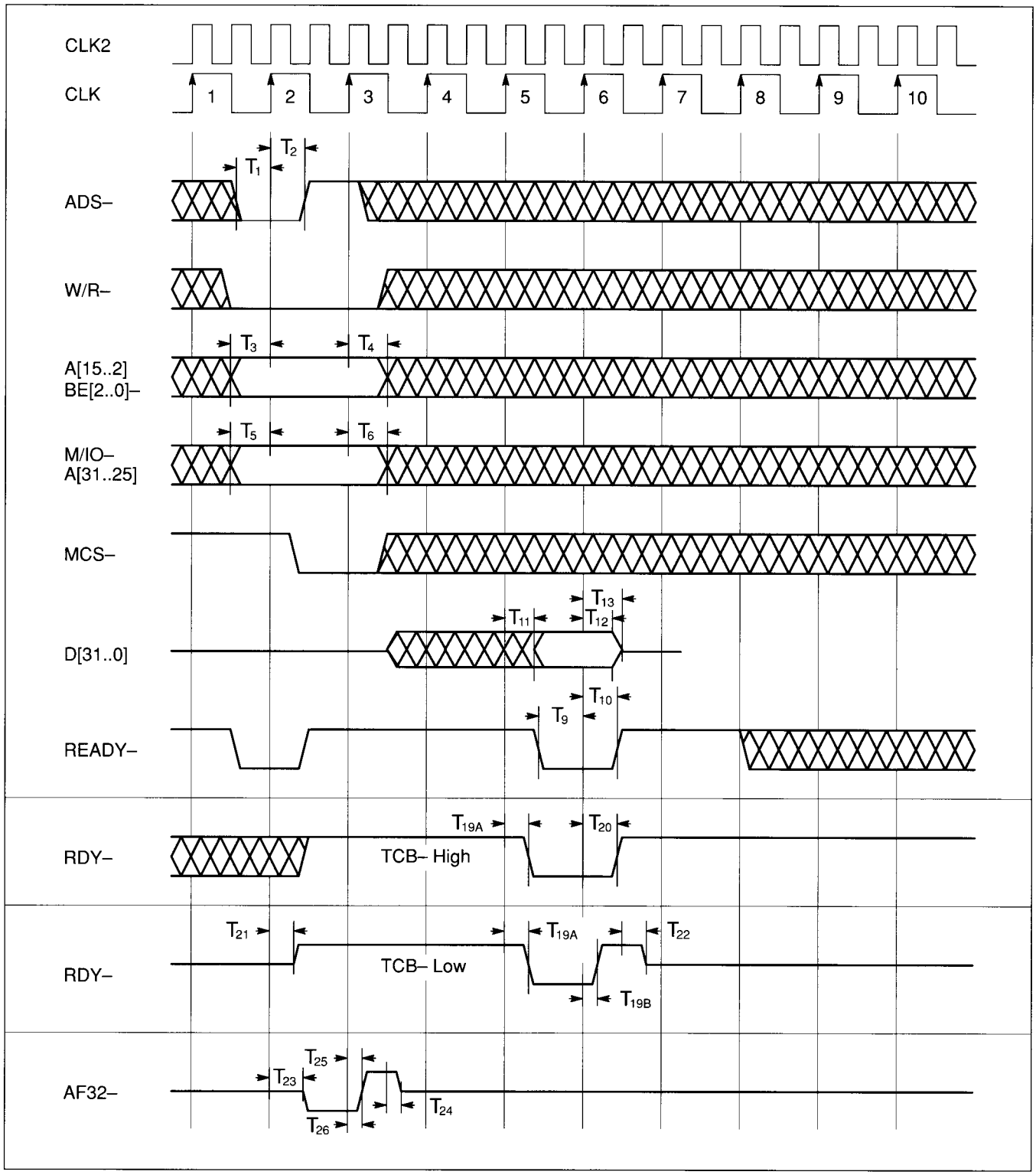


Figure 49. Pipelined bus read cycle without delayed ready for TCB– tied high and TCB– tied low

16. Timing Diagrams, continued

16.3. RESET AND INTERRUPT TIMING

RESET setup and hold time and interrupt valid delays are shown in figures 50 and 51, respectively. RESET must be synchronous with the 80386's clock to guarantee proper operation.

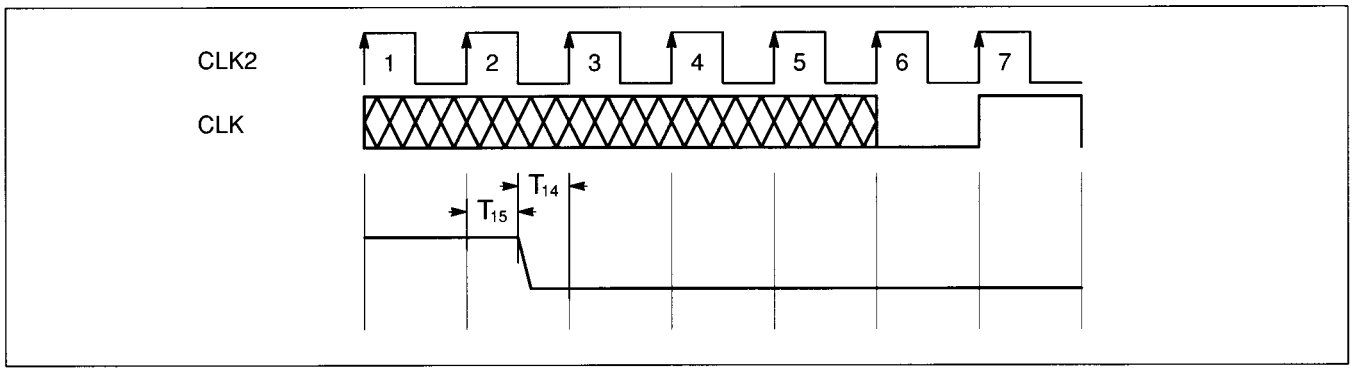


Figure 50. RESET timing

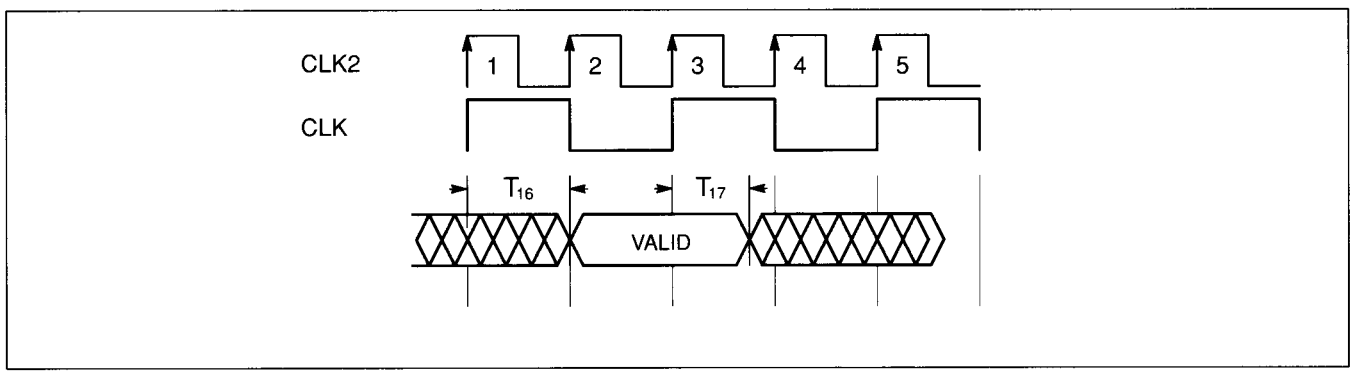


Figure 51. Interrupt timing

17. Pin Configuration

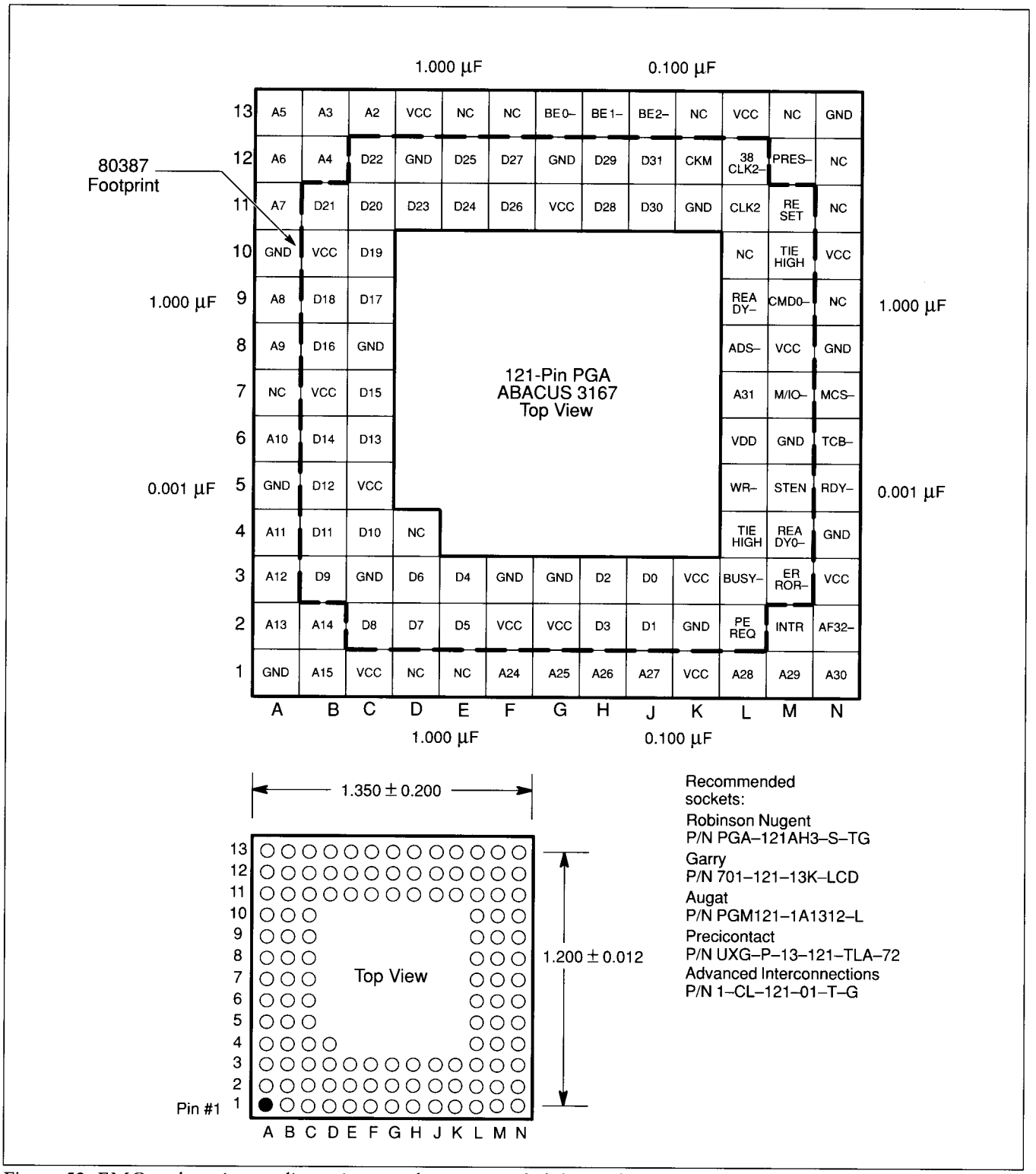
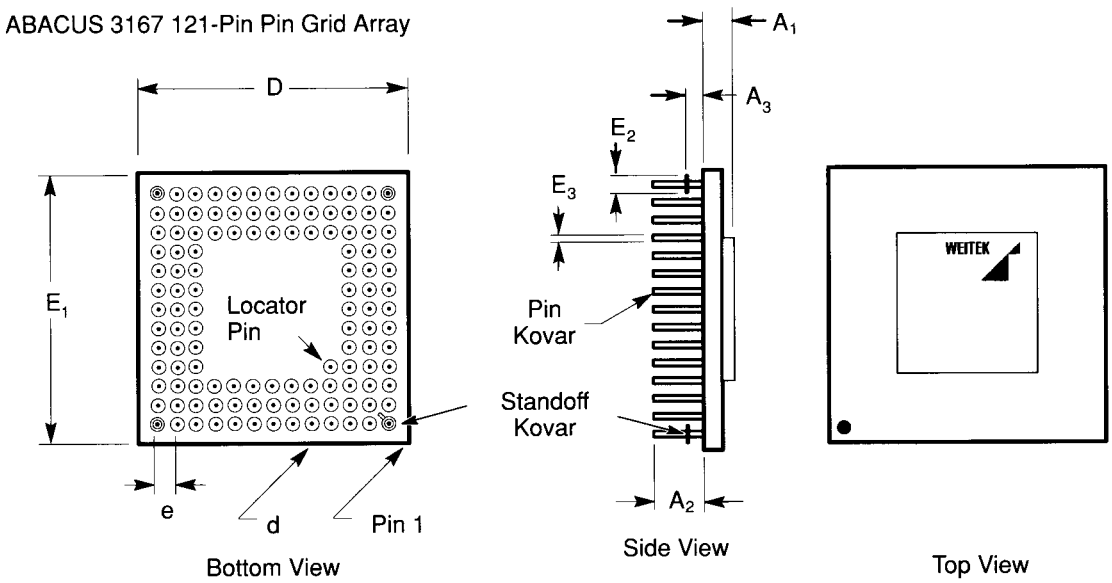


Figure 52. EMC socket pinout, dimensions, and recommended decoupling

18. Physical Dimensions



Symbol	Inches		Millimeters	
	MAX	MIN	MAX	MIN
A1	0.135	0.080	3.43	2.03
A2	0.210	0.175	5.33	4.46
A3	0.080	0.035	2.03	0.89
D	1.400	1.280	35.56	32.51
E1	1.200 TYP		30.48 TYP	
E2	0.055	0.035	1.40	0.89
E3	0.020	0.016	0.51	0.41
d	0.075	0.035	1.91	0.89
e	0.100 TYP		2.54 TYP	

Figure 53. ABACUS 3167 physical dimensions

18. Physical Dimensions, continued

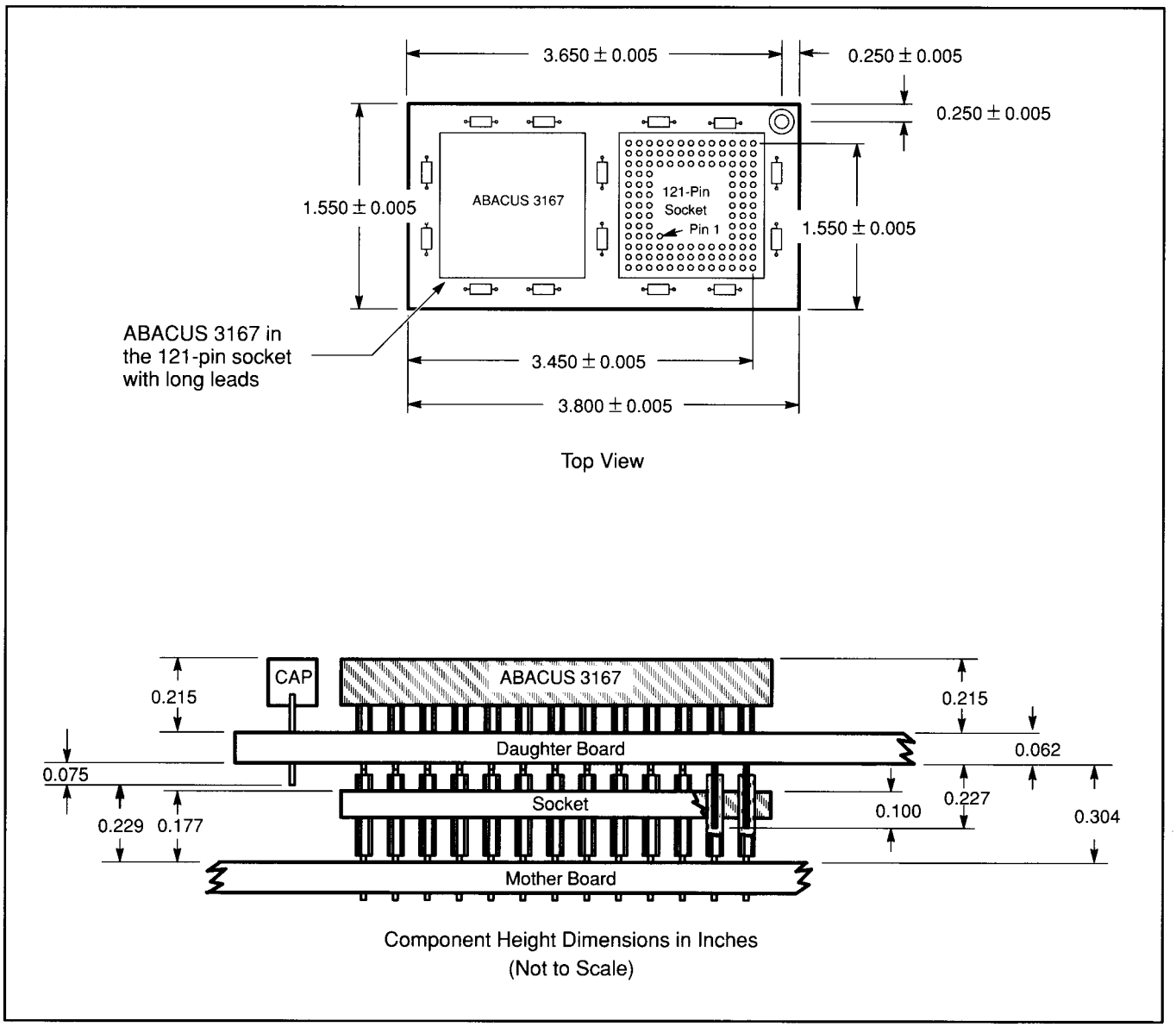


Figure 54. ABACUS 3167 coprocessor board physical dimensions

19. Ordering Information

19.1. COPROCESSOR

The ABACUS 3167 is offered in speed grades of 20, 25, and 33 MHz. Use the order numbers in figure 55.

19.2. COPROCESSOR BOARD

The ABACUS 3167 can be ordered preinstalled in a small daughter board. This daughter board plugs into the system EMC socket and allows the ABACUS 3167 and an 80387 to be used at the same time. See figure 56.

19.3. UTILITIES AND DIAGNOSTICS

WEITEK also supplies diagnostic software for UNIX and combined utility/diagnostic software (including demos and macro instruction definitions) for DOS. See figure 57.

Part Description	Temperature Range	Order Number
20 MHz ABACUS 3167 Coprocessor	T_{CASE} = 0 to 85° C	3167–020–GCU
25 MHz ABACUS 3167 Coprocessor	T_{CASE} = 0 to 85° C	3167–025–GCU
33 MHz ABACUS 3167 Coprocessor	T_{CASE} = 0 to 85° C	3167–033–GCU

Figure 55. ABACUS 3167 coprocessor ordering information

Part Description	Temperature Range	Order Number
20 MHz ABACUS 3167 Coprocessor Board	T_{CASE} = 0 to 85° C	3167–020–BRD
25 MHz ABACUS 3167 Coprocessor Board	T_{CASE} = 0 to 85° C	3167–025–BRD
33 MHz ABACUS 3167 Coprocessor Board	T_{CASE} = 0 to 85° C	3167–033–BRD

Figure 56. ABACUS 3167 coprocessor board ordering information

Product	Order Number
UNIX Diagnostics	4800–1167–02
DOS Utilities	4800–3167–00

Figure 57. WEITEK-supplied support software