SIEMENS

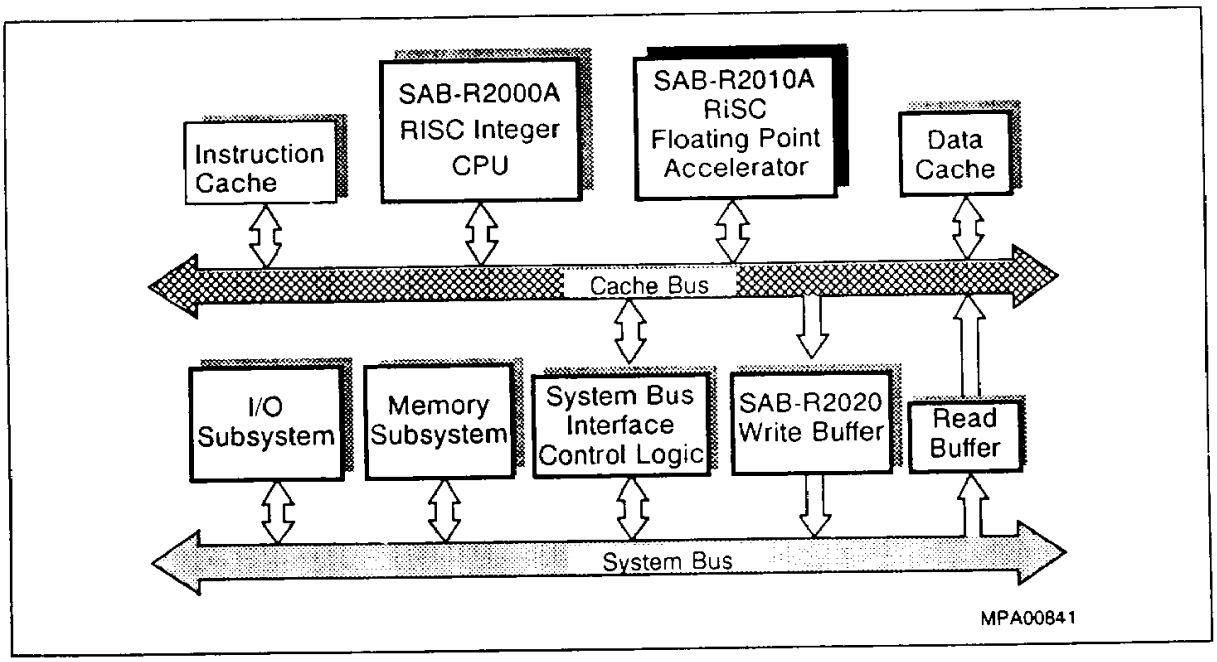
High Performance Floating-Point Coprocessor

SAB-R2010A

Based on advanced RISC architecture with four independent arithmetic functional units

Advance Information

- Fully conforms to ANSI/IEEE standard 754-1985 for binary floating-point arithmetic
- Load/store instruction set
  - single cycle loads and stores
- Four independent functional units
  - Register, Add, Divide and Multiply units
  - allows up to four floating-point instructions to be executed in parallel
- Full 64-bit operation
  - sixteen 64-bit floating-point registers
- Seamless coprocessor interface to SAB-R2000A
- Transparent addition of floating-point extensions to the SAB-R2000A's instruction set
- Fully compatible to all R2010A processors of other manufacturers
- Ceramic package: CL-CC-84



Ordering Information

Type	Ordering code	Package	Description
SAB-R2010A-12-QJ	Q67120-C553	CL-CC-84	32/64-bit Floating-Point Coprocessor, 12.5 MHz
SAB-R2010A-16-QJ	Q67120-C495	CL-CC-84	32/64-bit Floating-Point Coprocessor, 16.67 MHz

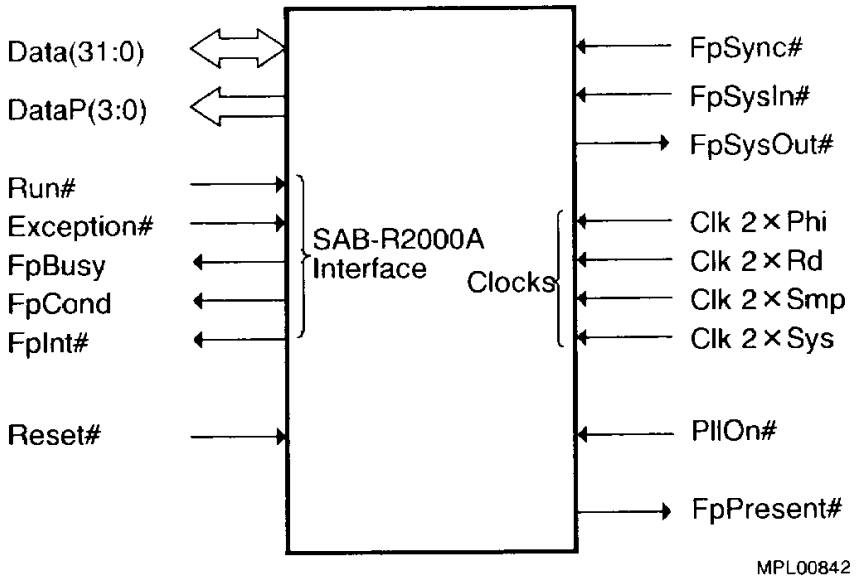
Introduction

The SAB-R2010A is a high performance Floating-Point Accelerator (FPA) which is implemented as a full-custom VLSI CMOS chip. It serves as a coprocessor to the SAB-R2000A RISC microprocessor. It transparently extends the SAB-R2000A's instruction set to perform floating-point operations, by cointerpreting the common instruction stream. The FPA, with associated system software, fully conforms to the requirements of ANSI/IEEE standard 754-1985 "IEEE Standard for Binary Floating-Point Arithmetic". In addition the SAB-R2010A fully supports the standard recommendations. The SAB-R2010A's architecture is organised as follows – hardware directly implements the essential floating-point operations of addition, subtraction, division, multiplication, comparison, conversion between formats, absolute value and negation. These operations are highly optimized. System software supplies the more complex functions and while doing so benefits from the underlying fast arithmetic hardware. Figure 1 illustrates the SAB-R2010A Logic symbol.

Pin Names

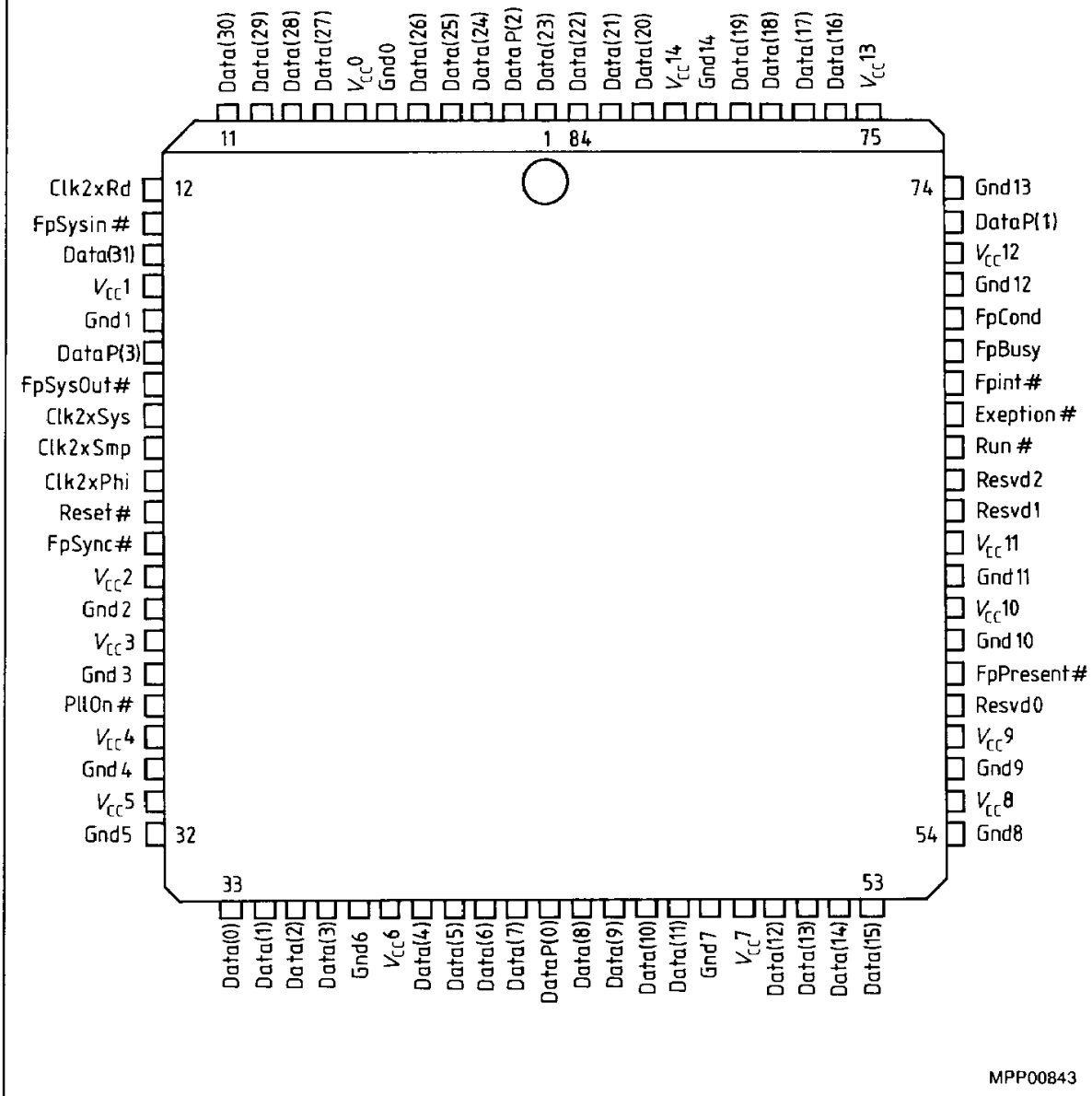
Data(31:0)	Data Bus
DataP(3:0)	Even parity for Data Bus
Run#	System in Run or Stall state
Exception#	Exception related information
FpBusy	Floating-point busy stall
FpCond	Floating-point condition
FpInt#	Floating-point Interrupt
Reset#	Synchronous Initialization
FpSync#	Floating-point Synchronize
FpSysIn#	Floating-point System clock in
FpSysOut#	Floating-point System clock out
PllOn#	Phase Lock Loop On
FpPresent#	Floating-point present

Figure 1
Logic Symbol



Pin Configurations

Figure 2
CL-CC-84 (Top View)



Pin Definitions and Functions

Symbol	Pin Number	Input (I) Output (O)	Function
Data(31:0)	14,11,10,9,8,5,4, 3,1,84,83,82,79, 78,77,76,53,52, 51,50,47,46,45, 44,42,41,40,39, 36,35,34,33	I/O	A multiplexed 32-bit bus used for instruction and data transfers on phases 1 and 2, respectively.
DataP(3:0)	17,2,73,43	O	A 4-bit bus containing even parity over the data bus. Parity is generated by the FPC on stores.
Run#	66	I	Input to the FPC which indicates whether the processor-coprocessor system is in the run or stall state.
Exception#	67	I	Input to the FPC which indicates exception related status information.
FpBusy	69	O	Signal to the CPU indicating a request for a coprocessor busy stall.
FpCond	70	O	Signal to the CPU indicating the result of the last comparison operation.
FpInt#	68	O	Signal to the CPU indicating that a floating-point exception has occurred for the current FPC instruction.
Reset#	22	I	Synchronous initialization input used to distinguish the processor-FPC synchronization period from the execution period. Reset# must be synchronized by the leading edge of SysOut from the CPU.
PllOn#	28	I	Input which during the reset period determines whether the phase lock mechanism is enabled, and during the execution period determines the output timing model.

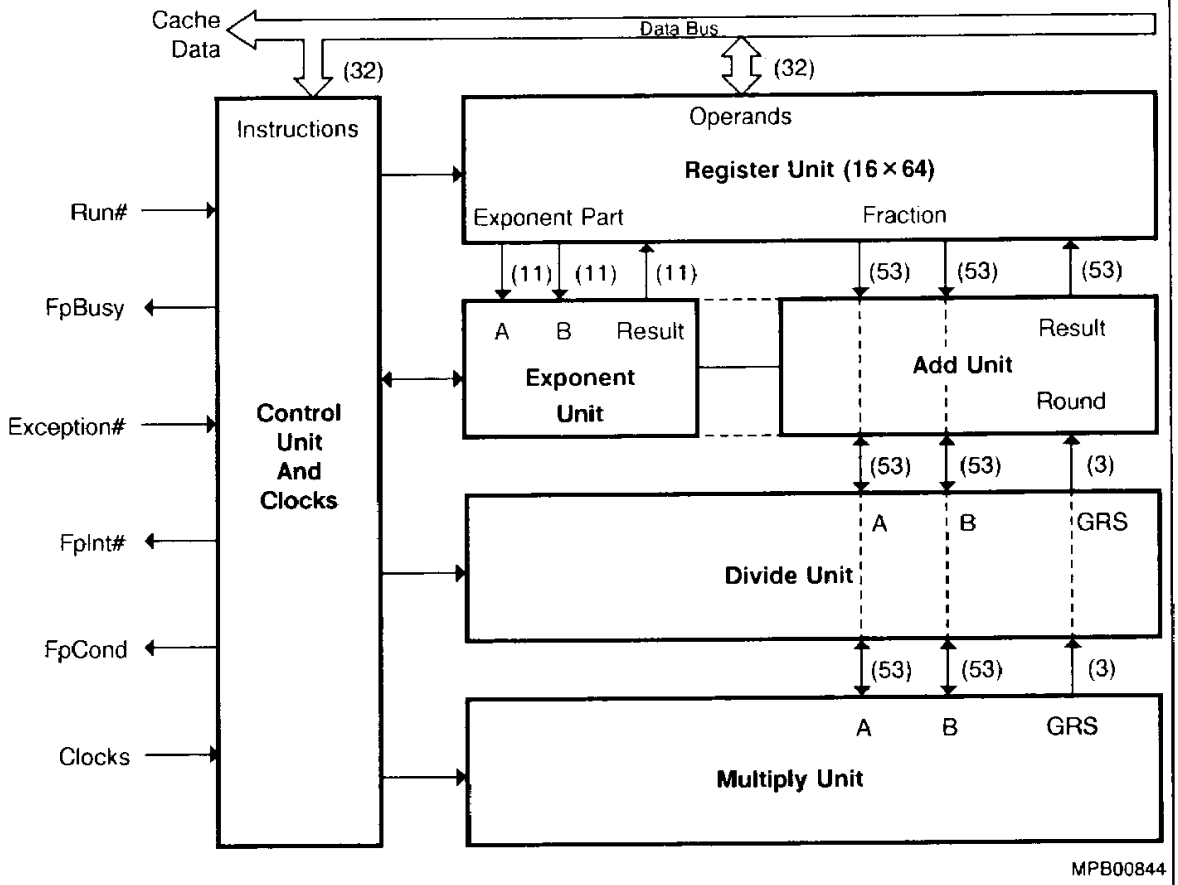
Pin Definitions and Functions (cont'd)

Symbol	Pin Number	Input (I) Output (O)	Function
FpPresent#	59	O	Output which is pulled to ground through an impedance of approximately 0.5 kΩ. By providing an external pull-up on this line an indication of the presence or absence of the FPC can be obtained.
Clk2 × Sys	19	I	A double frequency clock input used for generating FpSysOut#.
Clk2 × Smp	20	I	A double frequency clock input used to determine the sample point for data coming into the FPC.
Clk2 × Rd	12	I	A double frequency clock input used to determine the disable point for the data drivers.
Clk2 × Phi	21	I	A double frequency clock input used to determine the position of the internal phases 1 and 2.
FpSysOut#	18	O	Synchronization clock from the FPC.
FpSysIn#	13	I	Input used to receive the synchronization clock from the FPC.
FpSync#	23	I	Input used to receive the synchronization clock from the CPU.
GND14-1	80,74,71,62,60, 56,54,48,37,32, 30,27,25,16,6		Ground
V_{CC}14-1	81,75,72,63,61, 57,55,49,38,31, 29,26,24,15,7		Power Supply (+ 5 V)
Resvd2-0	65,64,58		Reserved

Functional Description

The SAB-R2010A contains four independent arithmetic functional units (Register, Add, Divide and Multiply) which interact with a scheduling and managing control unit. Figure 3 shows the block diagram of the SAB-R2010A.

Figure 3
Functional Block Diagram



Note: A hash sign # indicates an Active-LOW Signal

Basic Architecture

As figure 3 shows, the SAB-R2010A consists of five main units.

The **Control Unit** continually monitors the transactions between the SAB-R2000A (with which the SAB-R2010A shares the data bus) and the instruction cache (i.e. the instruction stream). If an instruction does not apply to the SAB-R2010A, it ignores it. When an instruction does apply, it interprets it. Synchronization between the coprocessor and the main processor is also managed by the control unit. The control unit monitors the signals Run# and Exception# to see what state the SAB-R2000A is in. Run# is used to track pipeline disruptions due to non-exceptional events (i.e. CPU stalls) such as cache misses, write busy etc. Exception# is used to track pipeline disruptions due to exceptional events such as virtual to physical address translation misses, interrupts etc. When either of these cases occurs, the SAB-R2010A's pipeline is shut down (stalled), in such away that unfinished instructions can be restarted later without numerical inconsistencies. Whenever the SAB-R2000A requires the result of a floating-point instruction which is not yet completed, the control unit signals the CPU to wait through the assertion of the FpBusy signal. The control unit also schedules the execution of each instruction with the four arithmetic units.

The **Register Unit's** register file can perform two 64-bit operand reads, one 64-bit write result and one memory load/data write in one cycle. This implies that four ports exist. Physically, a two-port design, which is accessed twice per cycle, implements the register file.

The **Add Unit** executes add, subtract, convert, compare, negate and absolute value instructions as well as the final IEEE rounding step of multiply and divide operations. The exponent data path (exponent unit) is included in this unit and it computes the 8-bits (single-precision) or 11-bits (double-precision) of exponents for all arithmetic operations.

The **Divide Unit** uses a radix-4, SRT-division algorithm to produce four quotient bits per cycle. This method uses a redundant encoding of the quotient as a sum of digits with values 2, 1, 0, -1 and -2. A double-precision divide requires a total of 19 cycles (12 cycles for a single-precision divide).

The **Multiply Unit** computes the product of the mantissa portions of its operands (refer to the Data Formats section). In the case of double-precision, the multiplier computes the product of two 53-bit operands in less than four cycles. It retains the most significant 56-bits of the 106-bit product.

Results of both the multiplier and divider are returned to the add unit over the two operand buses (A and B) for final carry propagation and rounding. A separate path exists for the guard, round and sticky bits (GRS) required for IEEE rounding. The interaction of the five units and the width of the major data buses can be seen in figure 3.

The autonomy of the four arithmetic units enables them to run in parallel. Concurrently executing instructions generally do not conflict for resources – except at the beginning and end cycles of each operation, mainly due to simultaneous requests for the add unit (see the Pipeline Architecture section). Based on the latency of each operation, the control unit schedules instructions to ensure that no two will need, for example, the exponent unit or rounding function of the add unit at the same time. The floating-point architecture requires that instructions must appear to complete in the order they were issued. However the control unit recognizes the special cases of operations with exceptional results, conflicts for one arithmetic unit and data dependencies between operations – therefore it will reschedule instructions for maximum pipeline efficiency. In such instances it adjusts the issue schedule to maintain the illusion of in-order instruction completion. Refer to the Pipeline Architecture section for more details.

Pipeline Architecture

The SAB-R2010A has an instruction pipeline which mirrors that of the SAB-R2000A processor. However, there is a difference: the FPA (SAB-R2010A) has a 6-stage pipeline in contrast to the 5-stage pipeline of the SAB-R2000A. It uses an extra pipestage to provide efficient coordination of exception responses between the FPA and the CPU (SAB-R2000A).The six stages of the SAB-R2010A pipeline are:

(1) IF — Instruction Fetch:
The CPU calculates the instruction address required to read an instruction from the instruction cache. The instruction address is generated and output during phase 2 of this pipestage. No action is required by the SAB-R2010A during this pipestage since the main processor is responsible for address generation. Note that the instruction is not actually read into the processor until the beginning of the RD pipestage. Refer to figure 4.

(2) RD — Register Fetch/Instruction Decode:
The instruction is present on the Data bus during phase 1 of this pipestage and the FPA decodes the data on the bus to determine whether it is an instruction for the FPA. The FPA reads any required operands from its registers (RF in figure 4) while decoding the instruction.

(3) ALU — ALU Operation:
If an instruction is one for the FPA, execution commences during this pipestage. If the instruction causes an exception, the FPA notifies the SAB-R2000A of the exception during this pipestage by asserting the FpInt# signal. If the SAB-R2010A determines that it requires additional time (i.e. more than 1 cycle) to complete this instruction, it initiates a stall during this pipestage.

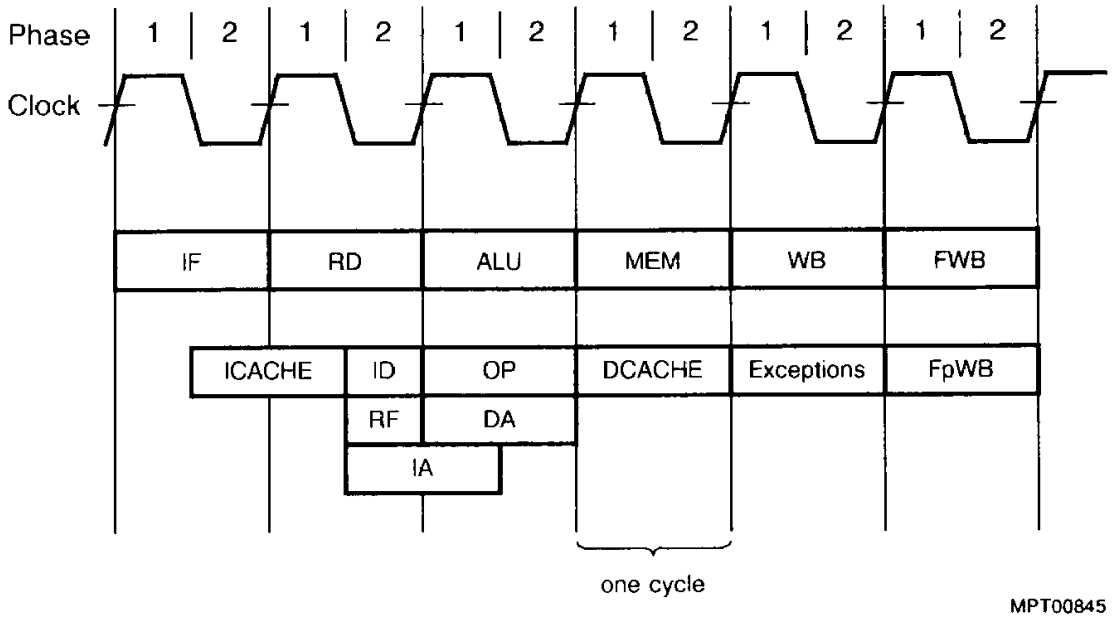
(4) MEM — Memory Access:
If it is a coprocessor Load or Store instruction, the FPA presents or captures the data during phase 2 of this pipestage. If an interrupt is taken by the main processor, it notifies the SAB-R2010A during phase 2 of this pipestage (via the Exception# signal).

(5) WB Write back:
If the instruction that is currently in the write back (WB) stage caused an exception, the main processor notifies the FPA by asserting the Exception# signal during this pipestage. Thus, the FPA uses this pipestage solely to deal with exceptions.

(6) FWB Floating-Point Write back:
The SAB-R2010A uses this pipestage to write back ALU results to its register file. This stage is the equivalent of the WB stage in the SAB-R2000A pipeline.

Figure 4 illustrates the 6 stages of the SAB-R2010A pipeline. Each step requires approximately one machine cycle.

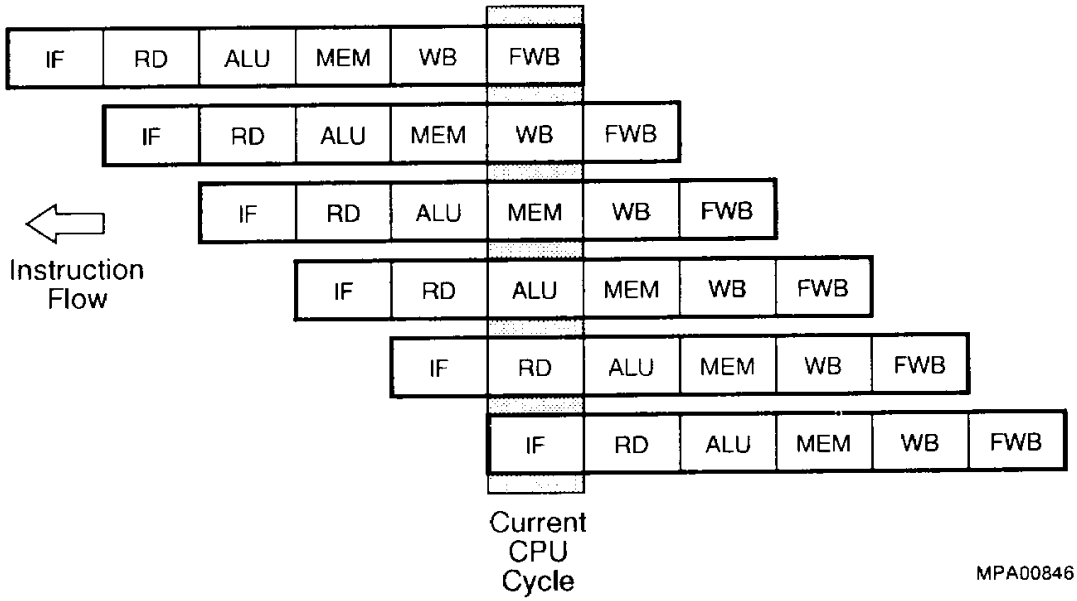
Figure 4
Instruction Execution Sequence



ICACHE	:	Instruction cache access
ID	:	Instruction decode
RF	:	Register operand fetch
IA	:	Instruction address calculation and translation
OP	:	Operation
DA	:	Data address calculation and translation
DCACHE	:	Data cache access
FpWB	:	Floating-point write back to register file
FpWB	:	Floating-point WB

The executions of six instructions are overlapping as shown in figure 5.

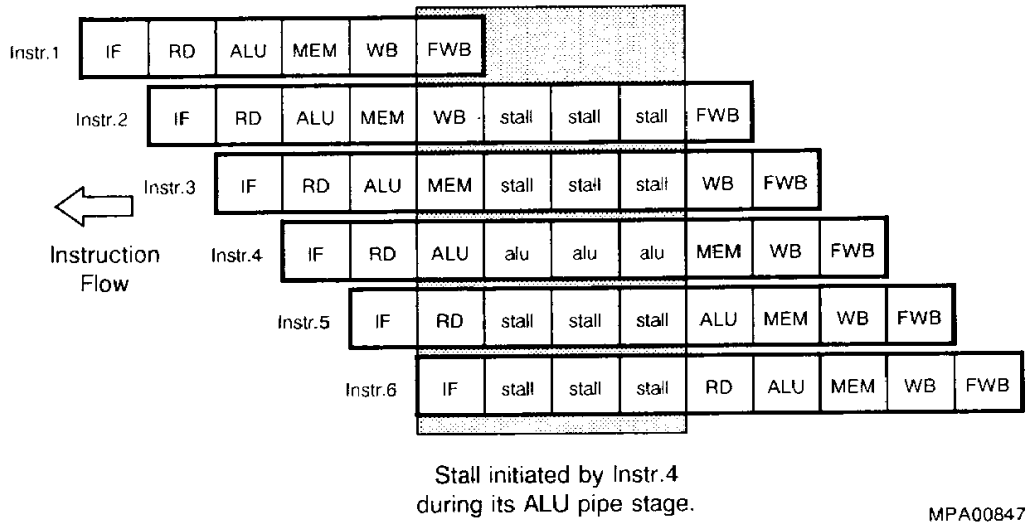
Figure 5
FPA Instruction Pipeline



This is a simplified view of the overlapped instruction execution of the SAB-R2010A because the figure assumes that each instruction can be completed in a single cycle. Most FPA instructions, however, require more than one cycle to be completed. Therefore, the pipeline must be stalled whenever register or resource conflicts occur. Figure 6 illustrates the effect of a three-cycle stall on the SAB-R2010A pipeline.

To alleviate the performance impact that would result from frequently stalling the pipeline, the SAB-R2010A overlaps instructions so that instruction execution can proceed so long as there are no resource conflicts, data dependencies or exceptional conditions.

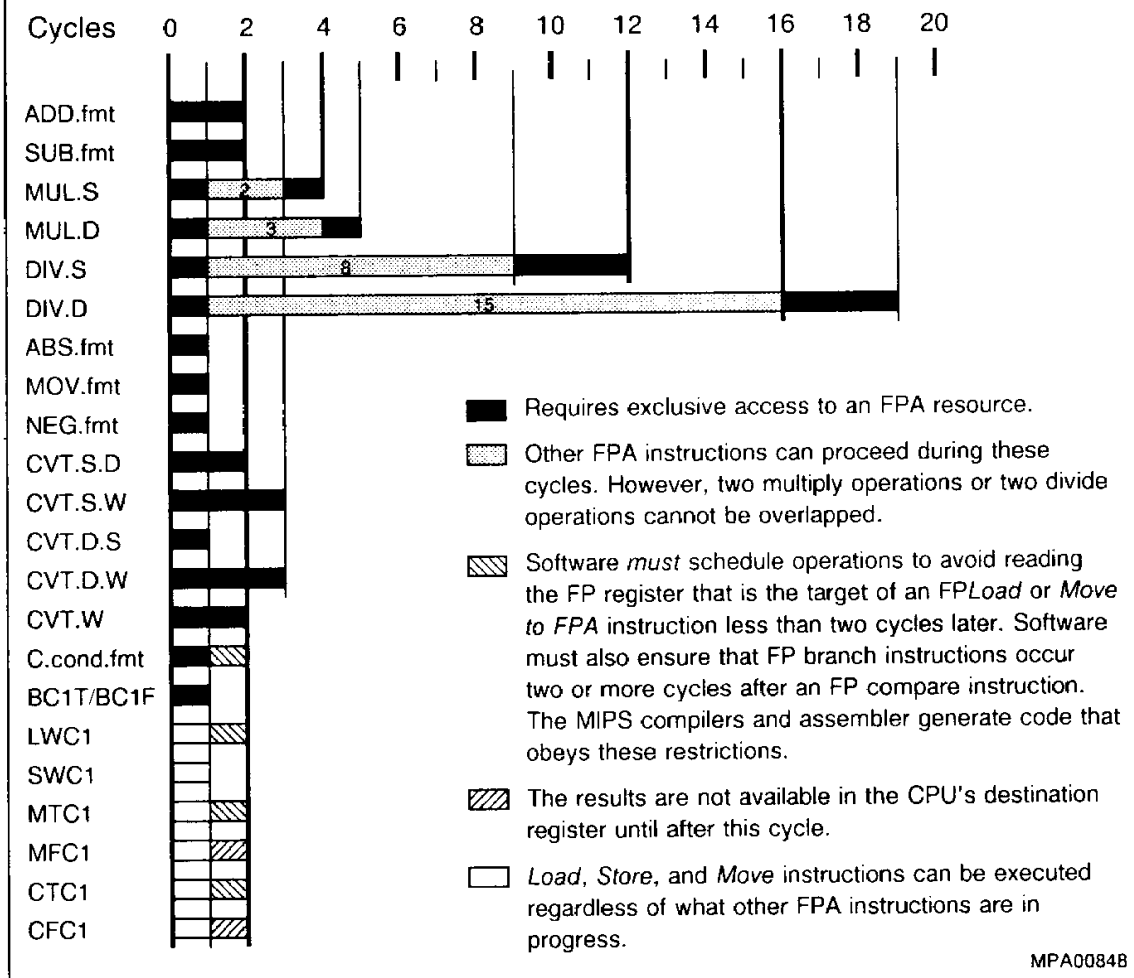
Figure 6
An FPA Pipeline Stall



As mentioned earlier the majority of SAB-R2010A instructions require more than one cycle to be completed. Figure 7 shows the number of cycles required to execute each of the FPA instructions, which varies from 1 to 19 cycles.

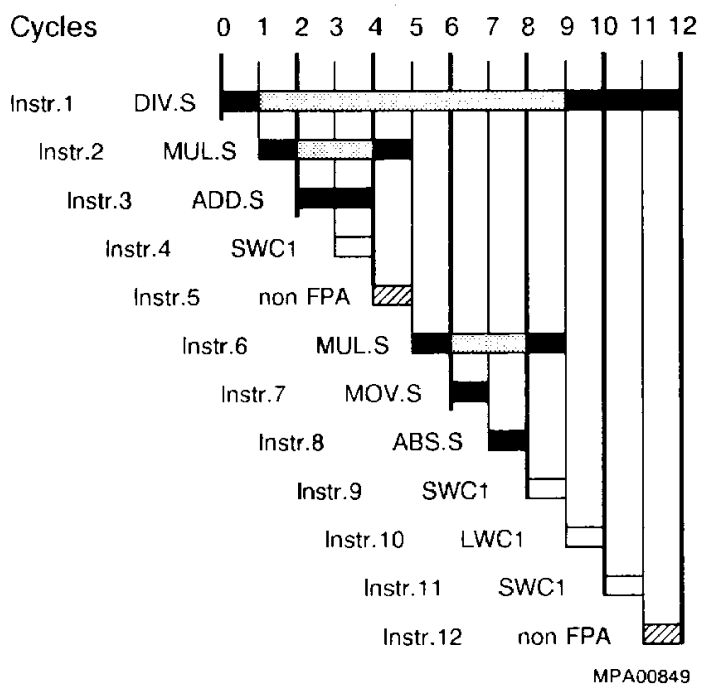
In figure 7 the cycles of an instruction's execution time which are shaded darkest (i.e. at the beginning and at the end of instruction execution time) require exclusive access to an FPA resource (such as the Add unit) that precludes the concurrent use by another instruction and therefore prohibits overlapping execution of another FPA instruction. However, Load and Store operations can be overlapped with these cycles because the SAB-R2010A's register unit can execute memory operations when the other arithmetic units are busy.

Figure 7
FPA Instruction Execution Times



Those cycles that are lightly shaded (i.e. in the middle of the Multiply and Divide instructions execution time) place minimal demands on SAB-R2010A resources (i.e. for a Multiply instruction only the Multiply unit is being used) and other instructions can be overlapped to obtain simultaneous execution of instructions without stalling the pipeline. However, two Multiply or two Divide operations cannot be overlapped. An example of overlapped FPA and non-FPA instructions is shown in figure 8.

Figure 8
Overlapping FPA Instructions



In this figure the first operation (DIV.S) requires a total of 12 cycles for execution. Only the first and last 3 cycles of this operation preclude the simultaneous execution of another FPA operation. Similarly, in the second operation (MUL.S) there are two cycles in the middle where an FPA operation can be overlapped. In this case the overlapping operation is ADD.S. Although the execution of an instruction requires 6 pipestages, the SAB-R2010A does not require that each instruction complete execution within 6 cycles to avoid stalling the instruction pipeline. If a subsequent instruction does not require the FPA resources being used by a preceding instruction and has no data dependencies with preceding uncompleted instructions, then execution continues. This can be seen clearly in figure 8.

This figure assumes that there are no data dependencies between the instructions that would stall the pipeline. For example, if any instruction before Instr.13 (not shown in figure 8) required the results of Instr.1 (DIV.S), then the pipeline would be stalled until the results are available.

Note: For a detailed discussion of the individual pipestages refer to the SAB-R2000A data sheet.

Coprocessor Registers

Floating-Point Registers

The SAB-R2010A provides thirty-two 32-bit Floating-Point General Registers (FGR's). These are accessed through coprocessor Load/Store instructions and Move to/from coprocessor register instructions. There are two views of the thirty-two coprocessor FGR's. One is from the standpoint of the SAB-R2000A, which has no intrinsic representation of coprocessor registers. It regards these registers as simply thirty-two 32-bit registers. From the standpoint of the SAB-R2010A, pairs of these single word registers form Floating-Point Registers (FPR's), on which floating-point operations are performed. The SAB-R2010A contains 16 FPR's. Figure 9 shows the FGR's and the corresponding FPR's.

The FPR's provide a sufficient amount of registers to support the allocation of floating-point values in registers and to permit overlapping and scheduling of floating-point operations. Each FPR can hold one value of either a single- or double-precision format floating-point number. Only even numbers are used to address FPR's, odd FPR register numbers are invalid. During single-precision floating-point operations, only the even numbered (least) FGR's are used, and during double-precision operations, the FGR's are accessed in pairs. Thus, in double-precision operation, selecting FPR0 addresses FGR0 and FGR1. Table 1 shows the register addresses.

Figure 9
FPA Registers

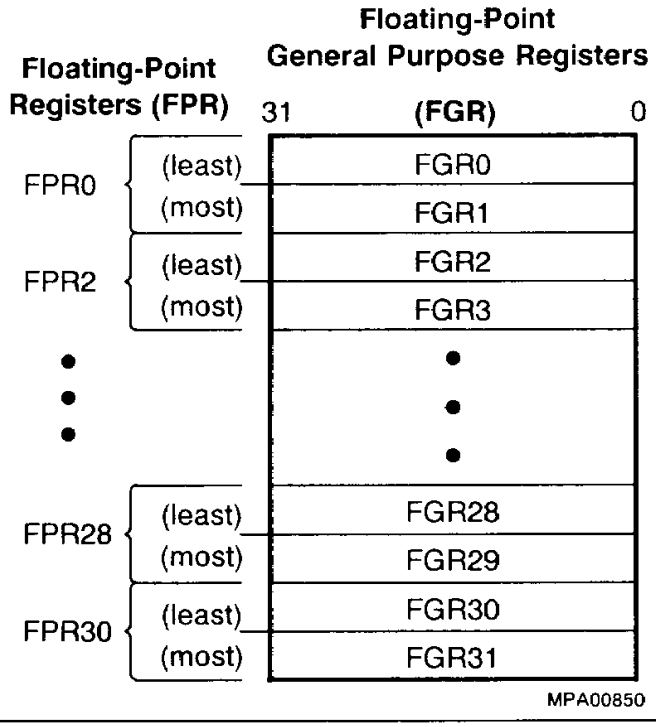


Table 1
Floating-Point General Registers

FGR Number	Usage		
0	FPR	0	(least)
1	FPR	0	(most)
2	FPR	2	(least)
3	FPR	2	(most)
•		•	
•		•	
•		•	
28	FPR	28	(least)
29	FPR	28	(most)
30	FPR	30	(least)
31	FPR	30	(most)

Floating-Point Control Registers

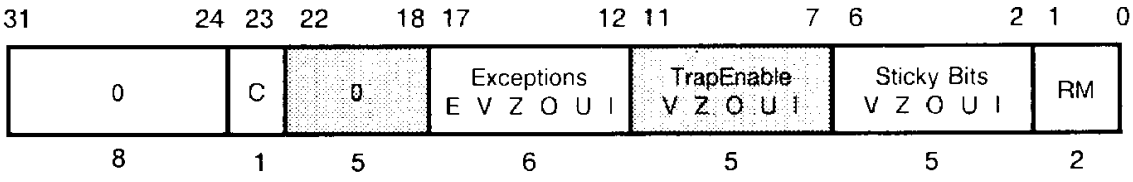
Coprocessors for the SAB-R2000A can have up to thirty-two 32-bit control registers. The SAB-R2010A implements two Floating-Point Control Registers (FCR's). These registers are the Control/Status register (FCR31) and the Implementation/Revision register (FCR0). These registers can only be accessed through Move to/from coprocessor register instructions which address floating-point control registers.

Control/Status Register:
contains control and status data and can be accessed by instructions running in either Kernel or User mode. It controls the arithmetic rounding mode and the enabling of exceptions. It also indicates the exceptions that occurred in the most recently executed instruction, and all exceptions that have occurred since the register was cleared.

Reading this register (using a Move Control From Coprocessor 1 instruction, CFC1), causes all unfinished instructions in the SAB-R2010A's pipeline to be completed before the contents of the register are transferred to the SAB-R2000A. If an exception occurs as the pipeline empties, the exception is taken and the Move instruction can be re-executed after the exception is serviced. Figure 10 illustrates the Control/Status register.

Figure 10
Control/Status Register Bit Assignments

The Control/Status Register



MPA00851

C : Condition bit. Set/cleared to reflect result of Compare instruction; drives the FPA's CpCond output signal.

Exceptions : These bits are set to indicate any exceptions that occurred during the most recent instruction.

TrapEnable : Trap Enables. These bits enable assertion of the CpInt# signal if the corresponding *Exception* bit is set during a floating-point operation.

Sticky bits : These bits are set if an exception occurs and are reset only by explicitly loading new settings into this register (with a Move instruction).

RM : Rounding Mode. These two bits specify which of the four rounding modes is to be used by the FPA.

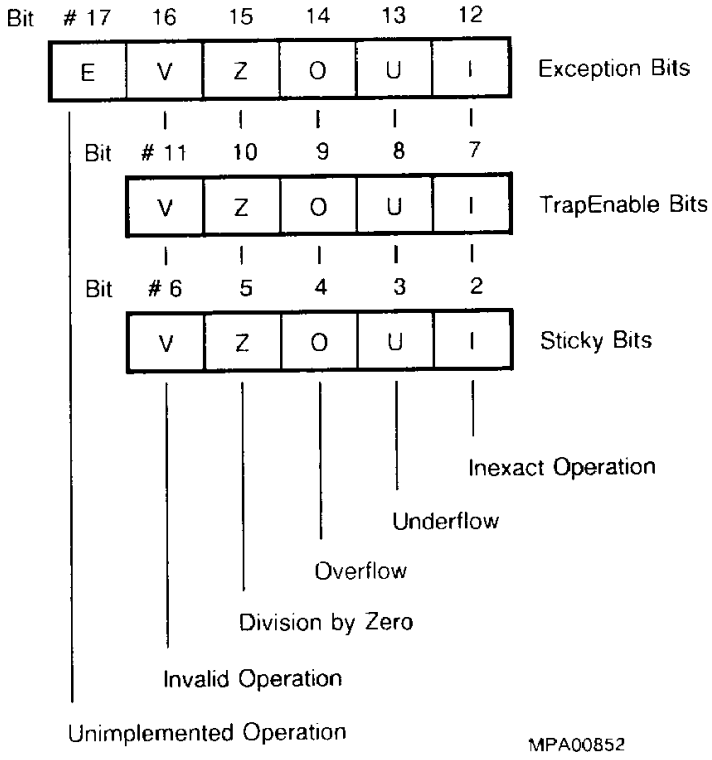
0 : Reserved. Currently ignores writes, undefined when read.

The bits in the Control/Status register can be set or cleared by writing to the register using a Move Control To Coprocessor 1 (CTC1) instruction. This register must only be written to when the FPA is not actually executing floating-point operations. This can be assured by first reading the contents of this register to empty the pipeline.

Condition bit:
When a floating-point Compare instruction takes place, the detected condition is placed at bit 23, the "C" (condition) bit, so that the state of the condition line may be saved or restored. If the condition is true it is set (1) and cleared (0) if it is false. This bit is only affected by Compare and Move To Control Register instructions.

Exception bits:
These are bits 17 through 12 in the Control/Status register, which are shown in figure 11, which indicates the meaning of each bit.

Figure 11
Control/Status Register Exception/Sticky/TrapEnable Bits



These bits are appropriately set or cleared after each floating-point instruction. This is a side effect of each floating-point operation (excluding Loads, Stores and unformatted Moves). The exceptions which were caused by the immediately previous floating-point operation can be determined by reading the exception field.

If two exceptions occur together in one instruction, both appropriate bits in the exception bit field will be set. When an exception occurs, both the corresponding exception and sticky bits are set. The exception bits cover the five IEEE standard exceptions and an extra unimplemented operation exception (E bit). The unimplemented operation exception is not one of the standard IEEE exceptions. It is provided to permit software implementation of IEEE standard operations and exceptions that are not fully supported by the FPA hardware. Trapping on this exception cannot be disabled – there is no TrapEnable bit for E.

Sticky bits:

Hold the accumulated or accured exception bits required by the IEEE standard for trap disabled operation. These bits are set whenever an FPA operation result causes one of the corresponding Exception bits to be set. However, unlike the Exception bits, the Sticky bits are never cleared as a side effect of floating-point operations; they can be cleared only by writing a new value into the Control/Status register.

TrapEnable bits:

Are used to enable a user trap when an exception occurs during a floating-point operation. If the TrapEnable bit corresponding to the exception is set (1) it causes the assertion of the FPA's FpInt# signal. The SAB-R2000A responds to the FpInt# signal by taking an interrupt exception which can be used to implement trap handling of the FPA exception.

Rounding Mode Control bits:

These bits specify the rounding mode the FPA will use for all floating-point operations as shown in table 2.

Table 2
Rounding Mode Bit Decoding

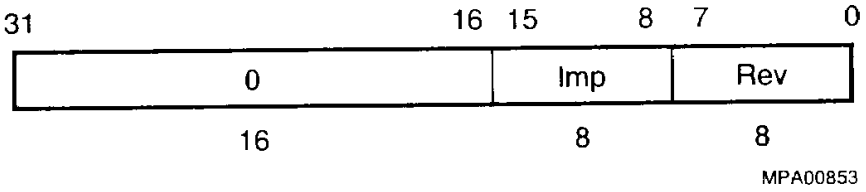
RM Bits	Mnemonic	Rounding Mode Description
00	RN	Rounds result to nearest representable value; rounds to value with least significant bit zero when the two nearest representable values are equally near.
01	RZ	Rounds result toward zero; rounds to value closest to and not greater in magnitude than the infinitely precise result.
10	RP	Rounds toward $+\infty$; rounds to value closest to and not less than the infinitely precise result.
11	RM	Rounds toward $-\infty$; rounds to value closest to and not greater than the infinitely precise result.

Implementation and Revision Register:

This read only register, FCR0, contains values that define the implementation and revision number of the SAB-R2010A. This information can be used to determine the co-processor revision and performance level and can also be used by diagnostic software. However, due to the variety of levels at which design changes may be implemented to the silicon, the revision information cannot be guaranteed with every revision of the device nor assured to follow a completely predictable numerical sequence. Siemens has complete discretion over defining these characteristics of the FPA.

Only the low-order bits of the implementation and revision register are defined. Bit 15 through 8 identify the implementation and bits 7 through 0 identify the revision number as shown in figure 12.

Figure 12
Implementation/Revision Register

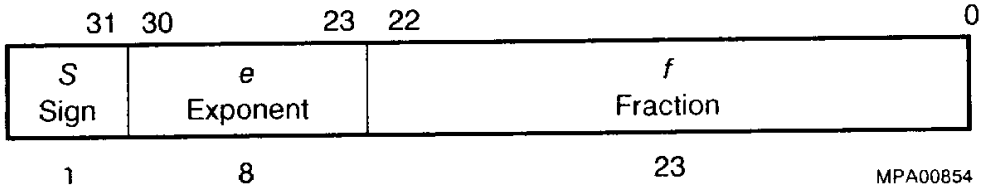


Imp : Implementation: 0 × 10 = R2010
Rev : Revision of FPA
0 : Unused; ignored on writes, zero when read.

Floating-Point Formats

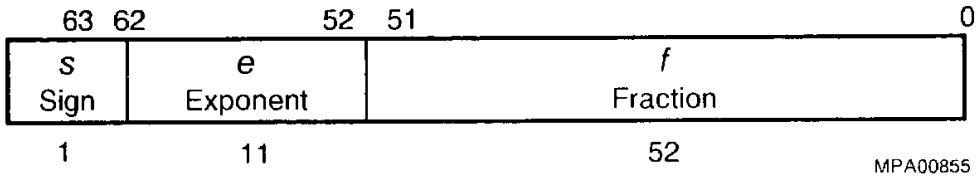
The SAB-R2000A performs both 32-bit (single-precision) and 64-bit (double-precision) IEEE standard floating-point operations. The 32-bit format ist divided into 3 fields: a single-bit sign, an 8-bit biased exponent and a 23-bit fraction, as shown in figure 13.

Figure 13
Single-Precision Floating-Point Format



The 64-bit format has a 1-bit sign, an 11-bit biased exponent and a 52-bit fraction field, as shown in figure 14.

Figure 14
Double-Precision Floating-Point Format



Numbers in the single- and double-precision floating-point formats are composed of three fields;

- A 1-bit sign: s
- A biased exponent: $e = E + \text{bias}$
- A fraction: $f = .b_1b_2...b_{p-1}$

The range of the unbiased exponent "E" includes every integer between and including two values "E_{min}" and "E_{max}", and also two other reserved values: "E_{min} - 1" to encode +/- 0 and denormalized numbers, and "E_{max} + 1" to encode +/- ∞ and NaNs (Not a Number). For single- and double-precision each representable non-zero numerical value has just one encoding.

For single- and double-precision formats, the value of a number, "v", is determined by the equations shown in table 3.

Table 3
Equations for Calculating Values in Floating-Point Format

(1)	if $E = E_{max} + 1$ and $f \neq 0$, then v is NAN, regardless of s.
(2)	if $E = E_{max} + 1$ and $f = 0$, then $v = (-1)^S \infty$.
(3)	if $E_{min} \leq E \leq E_{max}$, then $v = (-1)^S 2^E (1.f)$.
(4)	if $E = E_{min} - 1$ and $f \neq 0$, then $v = (-1)^S 2^{E_{min}} (0.f)$.
(5)	if $E = E_{min} - 1$ and $f = 0$, then $v = (-1)^S 0$.

For all floating-point formats, if "v" is NaN, the most significant bit of "f" determines whether the value is a signaling or quiet NaN. "v" is a signaling NaN if the most significant bit of "f" is set; otherwise, "v" is a quiet NaN. Signaling NaNs indicate uninitialized variables or variables for implementing user-designed extensions to the operations provided by the IEEE standard. Quiet NaNs are generated for invalid operations. Table 4 defines the values for the format parameters in the preceding description.

Table 4
Floating-Point Format Parameter Values

Parameter	Single	Double
P	24	53
E_{max}	+127	+1023
E_{min}	−126	−1022
exponent *bias*	+127	+1023
exponent width in bits	8	11
integer bit	hidden	hidden
fraction width in bits	23	52
format width in bits	32	64

Number Definitions

This subsection contains a definition of the following number types specified in the IEEE 754 standard:

- Normalized Numbers
- Denormalized Numbers
- Infinity
- Zero

Normalized Numbers:

The majority of floating-point calculations are performed on normalized numbers. Normalized numbers have a biased exponent "e" and a normalized fraction field "f" – which means that the leftmost (i.e. the one to the immediate left of the binary point), or hidden, bit is one.

Denormalized Numbers:

Have a zero exponent and a denormalized (hidden bit equal to zero) non-zero fraction field.

Infinity:

Has an exponent of all ones and a fraction field equal to zero. Both positive and negative infinity are supported.

Zero:

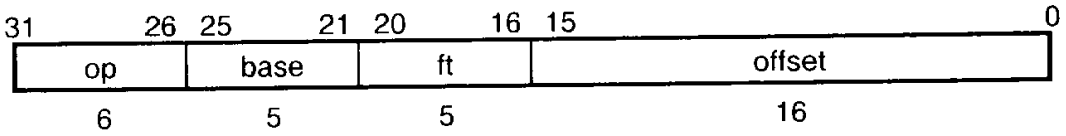
Has an exponent of zero, a hidden bit equal to zero and a value of zero in the fraction field. Both positive and negative zero are supported.

Instruction Set Overview

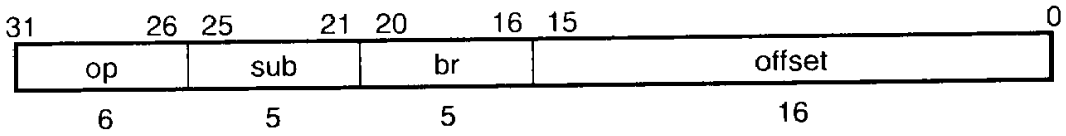
All SAB-R2010A instructions are 32-bits long. There are four basic instruction format types as shown in figure 15.

Figure 15
Instruction Formats

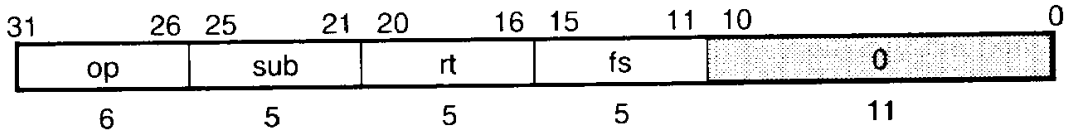
I-type (Immediate)



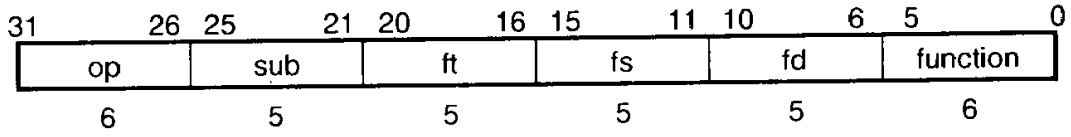
B-type (Branch)



M-type (Move)



R-type (Register)



MPA00856

where

op	:	is a 6-bit operation code
sub	:	is a 5-bit sub-operation code
br	:	is a 5-bit branch code
rt	:	is a 5-bit source/destination general register specifier
ft	:	is a 5-bit source/destination float register specifier
fs	:	is a 5-bit source register specifier
fd	:	is a 5-bit destination register specifier
offset	:	is a 16-bit address/branch displacement
function	:	is a 6-bit function code
0	:	result of operation undefined if non-zero

The single instruction length simplifies instruction fetch and decode and eliminates the overhead for instructions crossing word and page boundaries within the memory hierarchy, thereby simplifying the interaction of instruction fetch with the virtual memory management unit. The four instruction formats ensure that opcodes and register descriptors are always found in the same bit locations. This enables register fetch to proceed in parallel with instruction decode on all instructions.

The SAB-R2010A instruction set can be divided into the following groups:

- **Load/Store and Move** instructions move data between memory, the main processor and the FPA general registers.
- **Computational** instructions perform arithmetic operations on floating-point values in the FPA registers.
- **Conversion** instructions perform conversion operations between the various data formats, e.g. floating-point to fixed-point format.
- **Compare** instructions perform comparisons of the contents of registers and set the condition bit based on the results.

Table 5 lists the instruction set of the SAB-R2010A FPA. A more detailed summary is contained in the Instruction Set Summary section.

Table 5
Instruction Set Summary

OP	Description
Load/Store/Move Instructions	
LWC1	Load word to FPA
SWC1	Store word from FPA
MTC1	Move word to FPA
MFC1	Move word from FPA
CTC1	Move control word to FPA
CFC1	Move control word from FPA
Conversion Instructions	
CVT.S.fmt	Floating-point convert to single FP
CVT.D.fmt	Floating-point convert to double FP
CVT.W.fmt	Floating-point convert to fixed-point

OP	Description
Computational Instructions	
ADD.fmt	Floating-point add
SUB.fmt	Floating-point subtract
MUL.fmt	Floating-point multiply
DIV.fmt	Floating-point divide
ABS.fmt	Floating-point absolute value
MOV.fmt	Floating-point move
NEG.fmt	Floating-point negate
Compare Instructions	
C.cond.fmt	Floating-point compare

Exception Handling

This section describes how the SAB-R2010A FPA handles floating-point exceptions. The term exception is used for any infrequent or exceptional event that causes the SAB-R2010A to make a temporary transfer of control from its current process to another process that services the event. A floating-point exception occurs whenever the FPA cannot handle the operands or results of a floating-point operation in the normal way. On the occurrence of an exception the FPA either generates an interrupt (by asserting the signal FpInt#) to initiate a software trap, or sets a flag. If the trap is taken, the FPA remains in the state found at the beginning of the operation (i.e. execution is suspended) and a software exception handling routine is executed. If no trap is taken (i.e. a flag is set), an appropriate value is written into the SAB-R2010A destination register (of the exceptional instruction) and execution continues (see table 6).

The five IEEE standard exceptions are supported with exception bits, trap enables and sticky bits (status flags). Refer to the Control/Status register in the Coprocessor's Registers section. The SAB-R2010A has an additional exception type, unimplemented operation exception (E). This is used in cases where the FPA itself cannot implement the floating-point architecture specification, including cases where the FPA cannot determine the correct exception behaviour. The unimplemented operation exception has no trap enable or sticky bits; whenever this exception occurs, an unimplemented exception trap is taken (if the FPA's interrupt input to the SAB-R2000A is enabled). It is impossible to disable this exception, there is no trap enable bit.

Each of the five IEEE exceptions (Invalid Operation, Division by Zero, Overflow Exception, Underflow Exception and Inexact Operation) is associated with a trap under user control which is enabled by setting one of the five TrapEnable bits. When an exception occurs, both the corresponding Exception and Sticky bits are set. If the corresponding TrapEnable bit is set, the FPA generates an interrupt to the SAB-R2000A and the subsequent exception processing allows a trap to be taken.

Exception Processing

When a floating-point exception trap is taken, the SAB-R2000A processor's Cause register (refer to the SAB-R2000A data sheet) indicates that an external interrupt from the FPA is the cause of the exception and the SAB-R2000A's EPC (Exception Program Counter) contains the address of the instruction that caused the exception trap.

When no exception trap is signalled, a default action is taken, which provides a substitute value for the original, exceptional result of the floating-point operation. The default action taken depends on the type of exception and, in the case of the Overflow exception, the current rounding mode. Table 6 lists the default action taken by the FPA for each of the IEEE exceptions.

Table 6
FPA Exception Default Actions

Exception		Rounding Mode	Default Action (no exception trap signaled)
V	Invalid operation	– –	Supply a quiet NaN.
Z	Division by zero	– –	Supply a properly signed ∞.
O	Overflow	RN	Modify overflow values to ∞ with the sign of the intermediate result.
		RZ	Modify overflow values to the format's largest finite number with the sign of the intermediate result.
		RP	Modify negative overflows to the format's most negative finite number. Modify positive overflows to + ∞.
		RM	Modify positive overflows to the format's largest finite number. Modify negative overflow to – ∞.
U	Underflow	– –	Generate an unimplemented exception.
I	Inexact	– –	Supply a rounded result.

Internally the SAB-R2010A detects eight different conditions that can cause exceptions. When it encounters one of these unusual situations, it will cause either an IEEE exception or an Unimplemented Operation exception. Table 7 lists the exception-causing situations and contrasts the behaviour of the SAB-R2010A with the IEEE standard's requirements.

Table 7
FPA Exception-causing Conditions

FPA internal result	IEEE Stndrd	Trap Enab.	Trap Disab.	Note
Inexact result	I	I	I	loss of accuracy
Exponent overflow	O I*)	O I	O I	normalized exponent > E_{max}
Divide by zero	Z	Z	Z	zero is (exponent = E_{min}-1, mantissa = 0)
Overflow on convert	V	V	E	source out of integer range
Signaling NaN source	V	V	E	quiet NaN source produces quiet NaN result
Invalid operation	V	V	E	0/0 etc.
Exponent underflow	U	E	E	normalized exponent < E_{min}
Denormalized source	none	E	E	exponent = E_{min}-1 and mantissa < >0

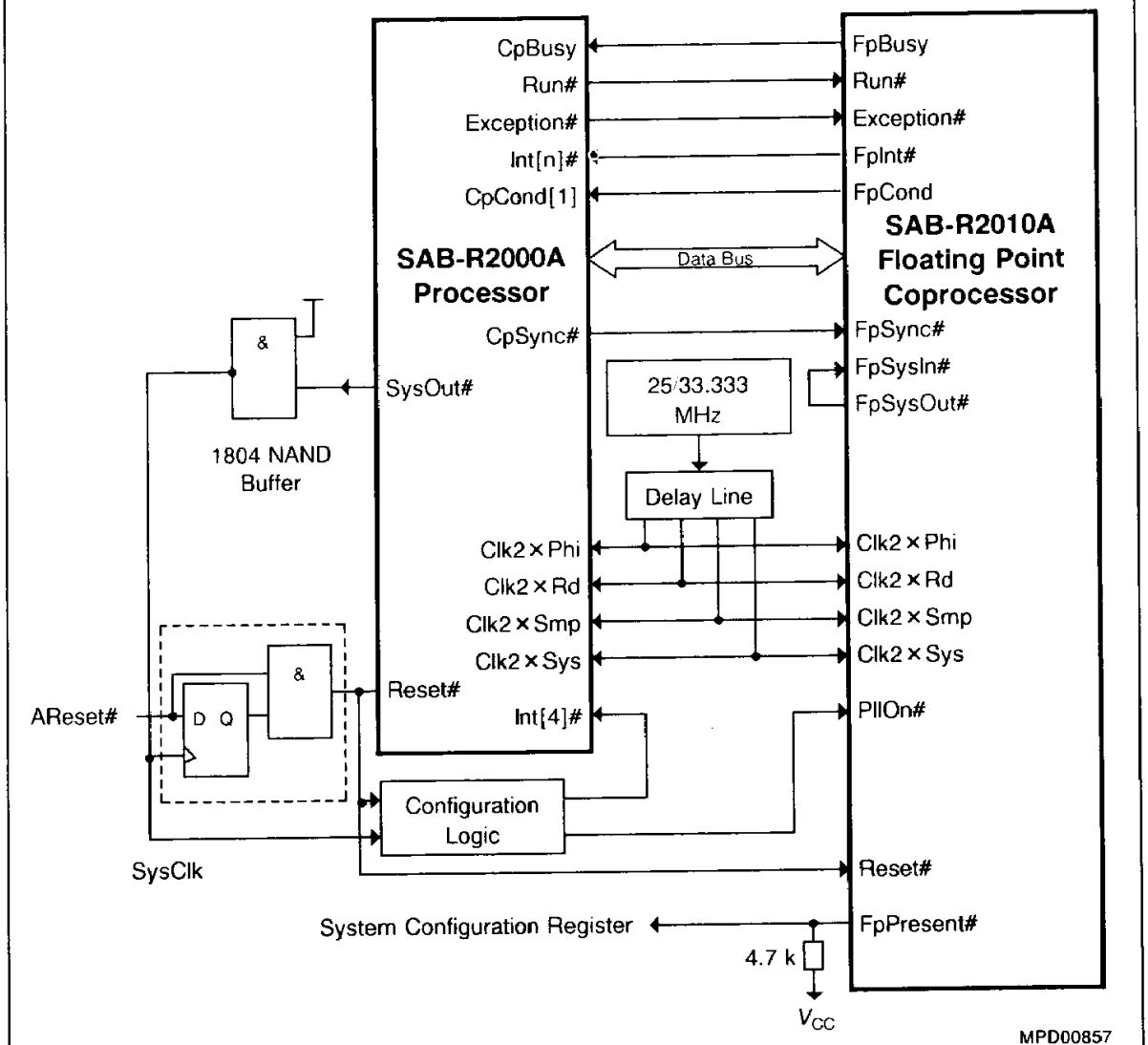
*) Standard specifies inexact exception on overflow only if overflow trap is disabled.

Note: *A detailed description of the "exception handling" system for a SAB-R2000A is contained in the SAB-R2000A data sheet.*

Processor Interface

Figure 16 illustrates the tightly coupled coprocessor interface between the SAB-R2000A and the SAB-R2010A.

Figure 16
SAB-R2000 and SAB-R2010 Coprocessor Interface



AReset = Asynchronous Reset

This external coprocessor interface of the SAB-R2000A is designed to support the SAB-R2010A floating point accelerator, in what is called a tightly coupled interface, and up to two additional coprocessors. The SAB-R2010A is connected to the DATA bus only. During each cycle in which a valid Instruction-Data pair is on the bus, the FPA accepts an Instruction. The coprocessor decodes the Instruction in parallel with the main processor and if it is a floating-point Instruction it will proceed to execute the Instruction. The coprocessor condition (CpCond(1) – FpCond) signal allows the main processor to branch on a coprocessor condition set up by a previous operation. The SAB-R2010A can assert FpBusy to stall the main CPU when a floating-point instruction is issued while the FPA still has the required functional unit busy with an earlier operation. The SAB-R2000A asserts Run# to advance operations in the SAB-R2010A. When Run# is deasserted in the n th cycle the FPA disregards the instruction-data pair presented in the n-1 th cycle. The assertion of Exception# indicates that the SAB-R2000A is taking an exception. FpSync# is used for timing synchronization between the SAB-R2000A and the coprocessor.

Instruction Set Summary

The following section is a table of the instructions available in the SAB-R2010A. The instructions are listed in alphabetical order. For a more detailed description of the operation of each instruction refer to the "SAB-R2010A Users Manual". A chart at the end of this section lists the bit encoding for the constant fields of each instruction.

Instruction Notation Convention

The table that follows is split up into three columns: Instruction, format and operation. The instruction column contains the mnemonic name of the instruction and its meaning. The instruction format (refer to figure 15) and assembly language notation for each instruction are listed in the format column. The operation column describes the operation performed by each instruction using a high level language notation. Special symbols used in the notation are described in table 8.

Table 8
FPA Instruction Operation Notations

Symbol	Meaning
←	Assignment
‖	Bit string concatenation
x^y	Replication of bit value *x* into a *y*-bit string. Note that *x* is always a single-bit value.
$x_{y..z}$	Selection of bits *y* through *z* of bit string *x*. Little-endian bit notation is always used. If *y* is less than *z*, this expression is an empty (zero length) bit string.
+	Two's complement or floating-point addition
–	Two's complement or floating-point subtraction
*	Two's complement or floating-point multiplication
div	Two's complement integer division
mod	Two's complement modulo
<	Two's complement less than comparison
and	Bitwise logic AND
or	Bitwise logic OR
xor	Bitwise logic XOR
nor	Bitwise logic NOR
GPR[*x*]	SAB-R2000A General Register *x*. Note that the contents of GPR[0] are always zero: attempts to alter GPR[0] contents have no effect.
FGR[*x*]	FPA General Register *x*. As viewed by the R2000A processor.
FPR[*x*]	FPA Floating-Point register *x*. Each FPR is assembled from two FGRs.
FCR[*x*]	FPA Control Register *x*.
T + *i*	Indicates the time steps (CPU cycles) between operations. Thus, operations identified as occurring at T + 1 are performed during the cycle following the one where the instruction was initiated. This type of operation occurs with loads, stores, jumps, branches and coprocessor instructions.
virtualAddress	Virtual address
physicalAddress	Physical address

In the Load/Store operation descriptions, the functions listed in table 9 are used to summarize the handling of virtual addresses and physical memory.

Table 9
Load/Store Common Functions

Function	Description
Addr Translation	Uses the TLB to find the physical address given the virtual address. The function fails and an exception is taken if the entry for the page containing the virtual address is not present in the TLB (Translation Lookaside Buffer).
Load Memory	Uses the cache and main memory to find the contents of the word containing the specified physical address. The low-order two bits of the address and the access type field indicate which of each of the four bytes within the data word need to be returned. If the cache is enabled for this access, the entire word is returned and loaded into the cache.
Store Memory	Uses the cache, write buffer, and main memory to store the word or part of word specified as data into the word containing the specified physical address. The low-order two bits of the address and the access type field indicate which of the four bytes within the data word should be stored.

The mnemonics of the floating-point instructions contain a ".fmt" field. This means "format" and table 10 shows the three formats available.

Table 10
".fmt" field encoding

Mnemonic	Size	Format
S	single	binary floating-point
D	double	binary floating-point
W	single	binary fixed-point

Instruction Set Summary

Instruction	Format	Operation
ABS.fmt: Floating-Point Absolute Value	R-Type; ABS.fmt fd, fs	T: StoreFPR (fd, fmt, AbsoluteValue(Value FPR (fs.fmt)));
ADD.fmt: Floating-Point Add	R-Type; ADD.fmt fd, fs, ft	T: StoreFPR (fd, fmt, ValueFPR(fs.fmt) + ValueFPR (ft.fmt));
BC1F: Branch On FPA False (Coprocessor 1)	B-Type; BC1F offset	T: target ← (offset $_{15}$) ^{14} ‖ offset ‖ 0^2 condition ← not CpCond(1) T + 1: If condition then PC ← PC + target endif
BC1T: Branch On FPA True (Coprocessor 1)	B-Type; BC1T offset	T: target ← (offset $_{15}$) ^{14} ‖ offset ‖ 0^2 condition ← CpCond(1) T + 1: If condition then PC ← PC + target endif

C.cond.fmt:
Floating-Point Compare
	R-Type;
	C.cond.fmt fs, ft
		T: if NaN(ValueFPR(fs, fmt)) or NaN(ValueFPR(ft, fmt)) then
		less ← false
		equal ← false
		unordered ← true
		if cond$_3$ then
		signal InvalidOperationException
		endif
		else
		less ← ValueFPR(fs.rmt) < ValueFPR(ft.rmt)
		equal ← ValueFPR(fs.rmt) = ValueFPR(ft.rmt)
		unordered ← false
		endif
		T + 1: condition ← (cond$_2$ and less) or
		(cond$_1$ and equal) or
		(cond$_0$ and unordered)

Instruction	Format	Operation
CFC1: Move Control word from FPA (Coprocessor1)	M-Type; CFC1 rt, ts	T: temp ← FCR[fs]; T + 1: GPR[rt] ← temp;
CTC1: Move Control word to FPA (Coprocessor1)	M-Type; CTC1 rt, fs	T: temp ← GPR[rt]; T + 1: FCR[fs] ← temp
CVT.D.fmt: Floating-Point Convert to Double FloatingPoint Format	R-Type; CVT.D.fmt fd, fs	T: StoreFPR (fd, D, ConvertFmt(ValueFPR(fs, fmt), fmt, D))
CVT.S.fmt: FloatingPoint Convert to Single FloatingPoint Format	R-Type; CVT.S.fmt fd, fs	T: StoreFPR (fd, S, ConvertFmt(ValueFPR(fs, fmt), fmt, S))
CVT.W.fmt: FloatingPoint Convert to FixedPoint Format	R-Type; CVT.W.fmt fd, fs	T: StoreFPR (fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
DIV.fmt: Floating-Point Divide	R-Type; DIV.fmt fd, fs, ft	T: StoreFPR (fd, fmt, ValueFPR(fs, fmt) / ValueFPR (ft, fmt));
LWC1: Load Word to FPA (Coprocessor 1)	I-Type; LWC1 ft, offset (base)	T: virtualAddress ← (offset $_{15}$) ^{16} ‖ offset $_{15..0}$ + GPR[base]; physicalAddress ← AddressTranslation (virtualAddress); mem ← LoadMemory (WORD, physicalAddress); byte ← virtualAddress $_{1..0}$; T + 1: FGR[ft] ← mem
MFC1: Move from FPA (Coprocessor 1)	M-Type; MFC1 rt, fs	T: temp ← FGR[fs]; T + 1: GPR[rt] ← temp
MOV.fmt: Floating-Point Move	R-Type; MOV.fmt fd, fs	T: StoreFPR (fd, fmt, ValueFPR (fs, fmt));

Instruction	Format	Operation
MTC1: Move to FPA (Coprocessor1)	M-Type; MTC1 rt, fs	T: temp ← GPR[rt]; T + 1: FGR[fs] ← data;
MUL.fmt: Floating-Point Multiply	R-Type; MUL.fmt fd, fs, ft	T: StoreFPR (fd, fmt, ValueFPR(fs, fmt) * ValueFPR (ft, fmt));
NEG.fmt: Floating-Point Negate	R-Type; NEG.fmt fd, fs	T: StoreFPR (fd, fmt, Negate(ValueFPR(fs, fmt)));
SUB.fmt: Floating-Point Subtract	R-Type; SUB.fmt fd, fs, ft	T: StoreFPR (fd, fmt, ValueFPR(fs, fmt) – ValueFPR (ft, fmt));
SWC1: Store Word from FPA (Coprocessor 1)	I-Type; SWC1 ft, offset (base)	T: virtualAddress ← (offset $_{15}$) ^{16} ‖ offset $_{15..0}$ + GPR[base] physicalAddress ← AddressTranslation (virtualAddress); data ← FGR[ft] T + 1: StoreMemory (WORD, data, physicalAddress)

Instruction Encoding

Opcode

31..29 \ 28..26	0	1	2	3	4	5	6	7
0	~	~	~	~	~	~	~	~
1	~	~	~	~	~	~	~	~
2	~	COP1	~	~	~	~	~	~
3	~	~	~	~	~	~	~	~
4	~	~	~	~	~	~	~	~
5	~	~	~	~	~	~	~	~
6	~	LWC1	~	~	~	~	~	~
7	~	SWC1	~	~	~	~	~	~

sub

25..24 \ 23..21	0	1	2	3	4	5	6	7
0	MF	~	CF	~	MT	~	CF	~
1	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗
2	Single	Double	⊗	⊗	⊗	⊗	⊗	⊗
3	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗

br

20..19 \ 18..16	0	1	2	3	4	5	6	7
0	BCF	BCT	~	~	~	~	~	~
1	~	~	~	~	~	~	~	~
2	~	~	~	~	~	~	~	~
3	~	~	~	~	~	~	~	~

function

5..3 \ 2..0	0	1	2	3	4	5	6	7
0	ADD.fmt	SUB.fmt	MUL.fmt	DIV.fmt	⊗	ABS.fmt	MOV.fmt	NEG.fmt
1	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗
2	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗
3	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗
4	CVT.S	CVT.D	⊗	⊗	CVT.W	⊗	⊗	⊗
5	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗
6	C.F	C.UN	C.EQ	C.UEQ	C.OLT	C.ULT	C.OLE	C.ULE
7	C.SF	C.NGLE	C.SEQ	C.NGL	C.LT	C.NGE	C.LE	C.NGT

⊗ Codes marked with a '⊗' cause unimplemented operation exceptions and are reserved for future versions of the architecture.

~ Codes marked with a '~' are not valid and are reserved for future versions of the architecture. The results of such an encoding are undefined

Timing Specifications

Absolute Maximum Ratings

Ambient temperature under bias (T_A)	0 to +70 °C
Storage temperature (T_{ST})	– 65 to +150 °C
Supply Voltage (V_{CC})	– 0.5 to +7.0 V
Input voltage (V_{IN})	– 0.5 to +7.0 V

Note: *Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.*
Not more than one output should be shorted at a time. Duration of the short should not exceed 30 seconds.

DC Characteristics

T_A = 0 to +70 °C; V_{CC} = 5 V ±5%

Parameter	Symbol	Limit values				Unit	Test condition
		12.5 MHz		16.67 MHz			
		min.	max.	min.	max.		
Operating Parameters							
Output HIGH voltage	V_{OH}	3.5	–	3.5	–	V	V_{CC} = min. I_{OH} = – 4mA
Output LOW voltage	V_{OL}	–	0.4	–	0.4	V	V_{CC} = min. I_{OL} = 4mA
Input HIGH voltage	V_{IH}	2	V_{CC} + 0.25	2	V_{CC} + 0.25	V	
Input LOW voltage	V_{IL}	– 0.5 1)	0.8	– 0.5 1)	0.8	V	
Input HIGH voltage	V_{IHS} 2)	2.5	V_{CC} + 0.25	3.0	V_{CC} + 0.25	V	
Input LOW voltage	V_{ILS} 2)	– 0.5 1)	0.4	– 0.5 1)	0.4	V	
Input HIGH voltage	V_{IHC} 3)	4.0	V_{CC} + 0.25	4.0	V_{CC} + 0.25	V	
Input LOW voltage	V_{IL} 3)	– 0.5 1)	0.4	– 0.5 1)	0.4	V	
Input capacitance	C_{In}	–	10	–	10	pF	
Output capacitance	C_{Out}	–	10	–	10	pF	
Operating current	I_{CC}	–	550	–	650	mA	V_{CC} = 5.25 V

1) V_{IL} min. = –3.0 V for pulse width less than 15 ns
2) V_{IHS} and V_{ILS} apply to Clk2 × Sys, Clk2 × Smp, Clk2 × Rd, Clk2 × Phi, FpSysIn#, FpSync# and Reset#.
3) V_{IHC} and V_{ILC} apply to Run# and Exception#.

AC Characteristics

T_A = 0 to 70 °C; V_{CC} = 5 V ±5%

Notes: *All output timings are given assuming 25 pf of capacitive load. Output timings should be derated where appropriate as per the table below.*
All timings referenced to 1.5 V.

Parameter	Symbol	Limit values				Unit	Test condition
		12.5 MHz		16.67 MHz			
		min.	max.	min.	max.		
Clock Parameters [4]							
Input clock high	$t_{ClkHigh}$	16	–	12	–	ns	Transition ≤ 5 ns
Input clock low	t_{ClkLow}	16	–	12	–	ns	Transition ≤ 5 ns
Input clock period	t_{ClkP}	40	1000	30	1000	ns	
Clk2×Sys to Clk2×Smp		0	$t_{\frac{Cyc}{4}}$	0	$t_{\frac{Cyc}{4}}$	ns	
Clk2×Smp to Clk2×Rd		0	$t_{\frac{Cyc}{4}}$	0	$t_{\frac{Cyc}{4}}$	ns	
Clk2×Smp to Clk2×Phi		11	$t_{\frac{Cyc}{4}}$	9	$t_{\frac{Cyc}{4}}$	ns	
Run Operation Parameters							
Data enable	t_{DEn}	– 1	– 2.5	– 1	– 2	ns	–
Data disable	t_{DDis}	0	– 1	0	– 1	ns	–
Data valid	t_{DVal}	–	3.5	–	3	ns	25 pF load
Data setup	t_{DS}	11.5	–	9	–	ns	–
Data hold	t_{DH}	– 2.5	–	– 2.5	–	ns	–
FpCondition	t_{FpCond}	0	45	0	35	ns	–
FpBusy	t_{FpBusy}	0	20	0	15	ns	–
FpInterrupt	t_{FpInt}	0	55	0	40	ns	–
FpMove To	t_{FpMov}	0	45	0	35	ns	–
Exception setup	t_{ExS}	15	–	10	–	ns	–
Exception hold	t_{ExH}	0	–	0	–	ns	–
Run setup	t_{RunS}	15	–	10	–	ns	–
Run hold	t_{RunH}	– 2	–	– 2	–	ns	–
Capacitive Load Deration							
Load derate	C_{LD}	0.5	2.5	0.5	2	ns/ 25pF	

4) The clock parameters apply to all four 2xClocks: Clk2×Sys, Clk2×Smp, Clk2×Rd, and Clk2×Phi.

Operation Fundamentals

A "cycle" is the basic instruction processing unit of the SAB-R2010A processor. Cycles in which forward progress is made, i.e. an instruction is retired, are called "run" cycles. An instruction is retired either by its completion or, in the presence of an exception, its abortion. Cycles in which no forward progress is made are called "stall" cycles. Stall cycles are used for resolving urgent situations such as cache misses on loads, write system busy during stores, and coprocessor interlocks. All cycles can be classified as either run cycles or stall cycles. "Fixup" stall cycles occur during the final cycle of the stall and are used in general to fix up the conditions which caused the stall. Processor transactions which occur during the first half of the cycle are called phase 1 transactions while those which occur during the second half of the cycle are called phase 2 transactions.

As described earlier Run# is asserted by the SAB-R2000A during run cycles and deasserted during stall cycles. When Run# is deasserted during the n th cycle, the SAB-R2000A disregards the instruction-data pair presented during the n-1 th cycle. When Run# is reasserted during the m th cycle, the SAB-R2010A takes, as are placement for the instruction-data pair which was disregarded, the instruction-data pair presented during the m-1 th cycle – which was the final fixup cycle for whatever stalls equence was occurring.

Exception# is used by the FPA to track exception related information during run cycles and stall related information during stall cycles.

- During phase 1 of run cycles Exception# indicates whether an exception has occurred for the instruction which is currently in its "write back" pipestage. Unless the exception is occurring as a result of an interrupt request by the SAB-R2010A, the assertion of Exception# prevents any state from being committed in the FPA.
- During phase 2 of run cycles Exception# indicates whether an interrupt request is being granted for the instruction which is currently in its "memory access" pipestage.
- During phase 1 of stall cycles Exception# indicates whether the current stall cycle is a fixup cycle. When a fixup cycle is occurring, it is guaranteed that the data present on the data bus is valid. The FPA uses the fixup indication to qualify the use of data sampled from the bus during the stall.
- During phase 2 of stall cycles Exception# indicates whether the current stall is a Coprocessor Busy stall. The FPA does not use this information.

The use of the Exception# signal is summarized below.

Table 11
Exception#

Cycle	Phase 1	Phase 2
Run	Exc1W#	IntGr2M#
Stall	Fixup1#	CPBusy2#

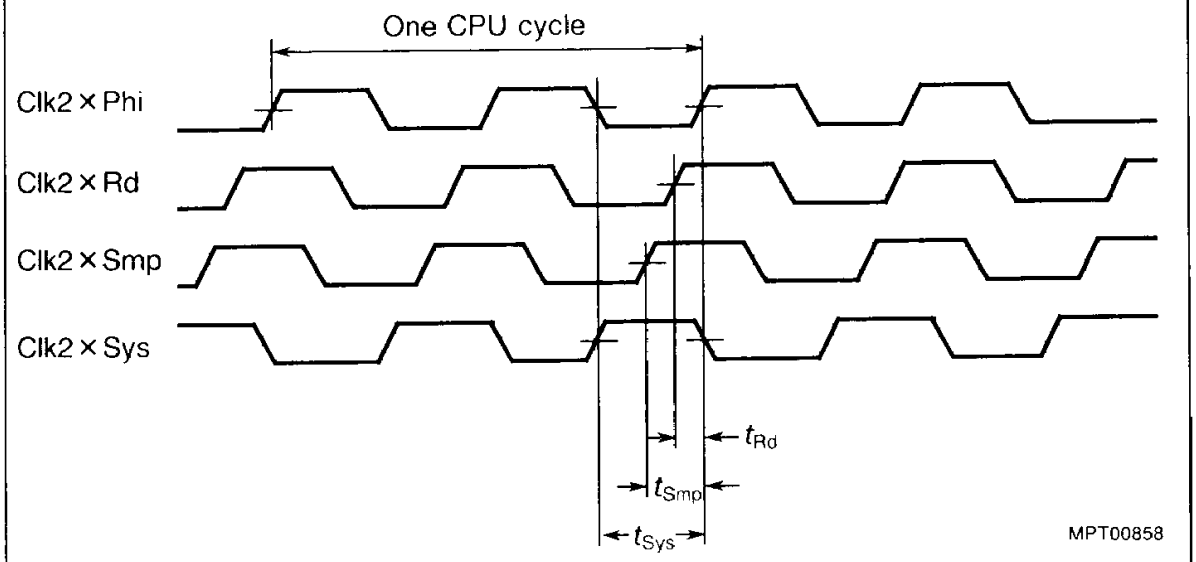
Processor Input Clocks

The SAB-R2010A has the same four separate double-frequency (i.e. in a 16.67 MHz system these clocks are 33.33 MHz) input clocks as the SAB-R2000A. They can be adjusted to obtain optimum positioning of cache interface signals. The absolute timing of these clocks with respect to the processor outputs is undefined, only the differences are important. A short description of these four clocks follows. Refer to figure 16 for the various signal names.

- Clk2 × Sys:
  is the master clock and must lead all others. It determines the position of SysOut# (the processors output clock) with respect to data, Tag and Address buses.
- Clk2 × Smp:
  determines the sample point for data coming into the SAB-R2000A on all its inputs except those coming directly from coprocessors.
- Clk2 × Rd:
  controls output enable time and provides sufficient address access to sample address hold from end of write, and data hold from end of write.
- Clk2 × Phi:
  determines the position of the internal phases 1 and 2.

Figure 17 shows the four 2 × input clocks.

Figure 17
2 × Input Clocks



In the timing diagrams which follow, timing specifications are given relative to a shifted version of the FPA output clock, FpSysOut#. The clock is called FpPhiOut# and is a virtual clock, i.e. the processor does not actually produce this output (FpSysOut# and FpPhiOut# are equivalent to SysOut# and PhiOut# for the SAB-R2000A). It is shown in the timing diagrams for reasons of clarity, because its period is synchronous with a machine cycle. The shift amount is equal to the difference between Clk2 × Sys to Clk2 × Phi and, as is shown in figure 17, is t_{Sys}. Also in the timing diagrams Clk2 × Sys and FpSysOut# are shown to clarify the relationship between these signals.

In reality FpSysOut# is produced rather than FpPhiOut# since this provides a signal with timing appropriate for synchronizing system transactions to the processor. Timings are given relative to FpPhiOut# since this makes determining the position of the input clocks the most straightforward. The timing of any output with respect to FpSysOut# can be determined from its timing with respect to FpPhiOut# by adding t_{Sys}.

Timing Diagram Notation

The following timing diagrams describe various transactions of the processor. Table 12 illustrates the notational conventions used in these diagrams.

Table 12
Notational Conventions for Timing Diagramms

Character	Meaning
I	Instruction
D	Data
#	Active low
%	An incorrect datum
!	An unused datum
Z	The high impedance state
Ad	Address
in	into coprocessor
out	out of coprocessor
/////	not valid or Don't Care

Load/Store and Processor Transfer Timings

During run cycles, the operation and timing of FPA loads and stores are identical to that of the SAB-R2000A. In the case of a load the SAB-R2010A accepts data from the data bus and on stores it drives data on the data bus. Both of these data bus transactions occur during the MEM pipestage of each instruction (refer to figure 4 in the pipeline architecture section). On FPA loads, the SAB-R2000A reads in the data and Tag buses for purposes of miss detection. The Tag bus is the cache Tag bus and is only connected to the CPU – for more information refer to the SAB-R2000A data sheet. For miss detection the SAB-R2000A checks the valid bit, does the Tag comparison and checks parity on the Tag and data buses. On Stores the SAB-R2010A generates data parity. All address generation, cache and memory control functions are provided by the SAB-R2000A.

During all stall and fixup cycles, the FPA is passive; if an FPA Store is blocked by a write busy stall or if the cycle in which the FPA Store occurs is redone due to any other stall, the SAB-R2000A will re-present the FPA data during the stall's fixup cycle. Timing of FPA Loads/Stores is illustrated in figure 19.

Transfers between the SAB-R2010A and the SAB-R2000A have identical input and output characteristics as Loads and Stores. That is, for a Move to FPA transfer (MTC1), the SAB-R2000A drives the data bus and the SAB-R2010A reads it, as for an FPA Load. For a Move from FPA instruction (MFC1) the roles are reversed, as for an FPA Store. Parity is not checked for either direction of transfer. The timings for these transfers are also illustrated in figure 19.

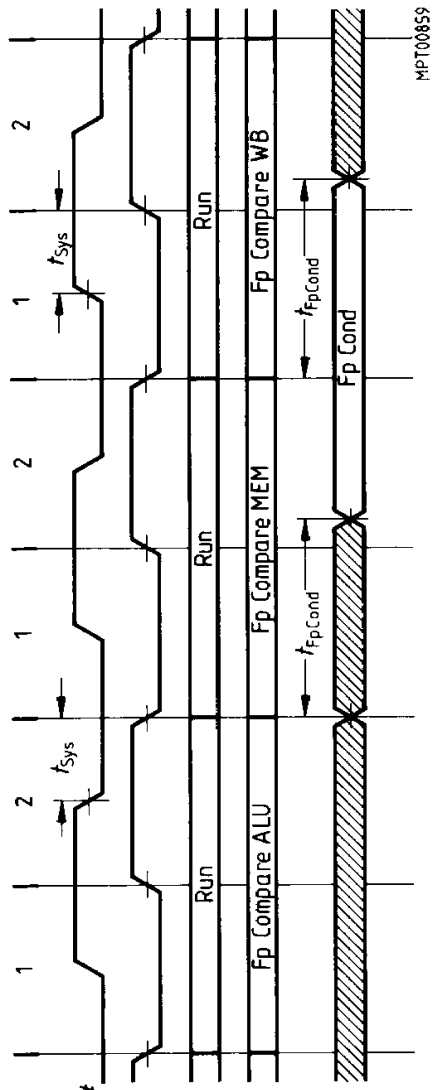
Floating-Point Condition Timing

Floating-point operations occur within the SAB-R2010A and only affect the interface when they change the Floating-Point Condition output or cause stalls or exceptions. Floating-point conditions are described here, stalls and exceptions are described in subsequent sections.

The SAB-R2010A has a Floating-Point Condition output called FpCond. The FpCond output is connected directly to the CpCond(1) input of the SAB-R2000A, refer to figure 16. The FpCond signal is sampled by the SAB-R2000A during phase 2 of every run cycle. If the SAB-R2000A executes an FPA branch instruction, the state of the FpCond signal determines the direction of the branch.

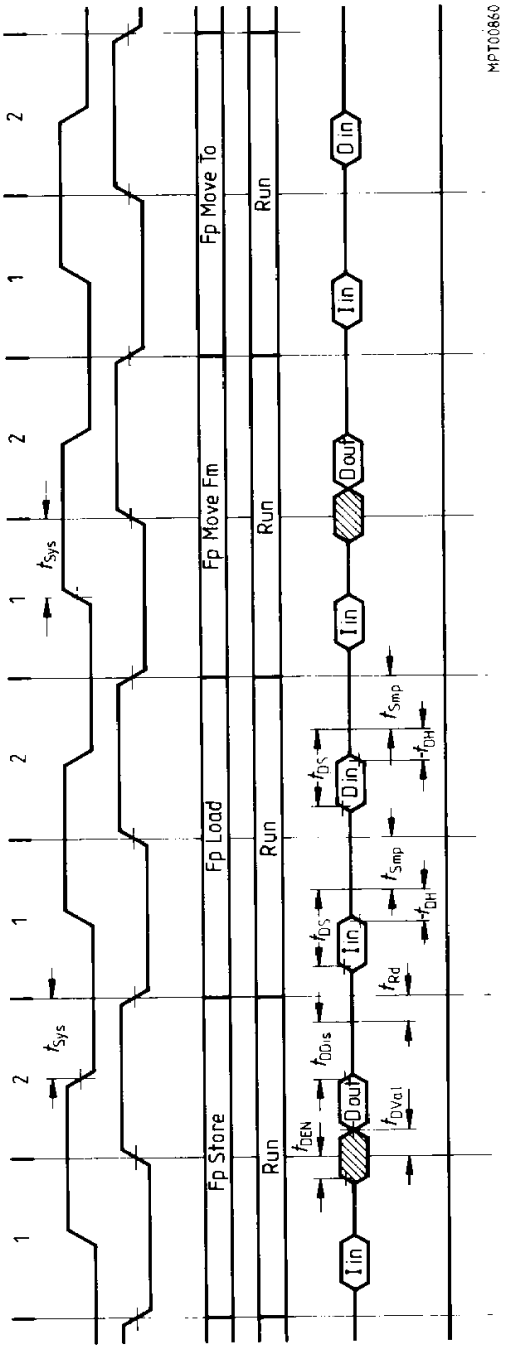
Floating-point instructions which affect the FpCond signal are two-cycle operations (e.g. Floating-Point Compare). This can be seen externally by the invalidity of the FpCond output during the entire ALU pipestage of Instruction Execution, refer to figure 18 which illustrates the FpCond timing. The FpCond output becomes valid during the MEM pipestage of instruction execution, which is too late to be used by the succeeding instruction, therefore the operation requires two cycles. Refer to the Pipeline Architecture section and the SAB-R2000A data sheet for more details on two cycle instructions.

Figure 18
FPCond Timing



Note: The Instruction used here to illustrate the FpCond timing is a Floating-Point Compare Instruction, where FpCompare ALU/MEM/WB are the Floating-Point Compare ALU/MEMORY/Write Back pipestages respectively.

Figure 19
Load/Store and Transfer Timing



FpMoveFm (FpMoveTo) has the same timing as FpStore (FpLoad).

Floating-Point Coprocessor Stall Timing

As described earlier, to maintain synchronization with the SAB-R2000A the FPA requests "Coprocessor Busy" stalls as required. To initiate such a stall the SAB-R2010A asserts FpBusy during phase 2 of the ALU pipestage of the stalling FPA instruction. To terminate the stall FpBusy is deasserted during phase 1 of the stall cycle in which it will complete the operation whose incompletion required the stall. In the absence of other stall requests, the cycle following that in which FpBusy is deasserted will be the fixup cycle. Figure 20 illustrates the FPA busy timing.

For all stalls, whether FPA-initiated or not, the indication of the stall condition is signalled by the SAB-R2000A via the deassertion of the Run# signal. If Run# is not deasserted following the assertion of FpBusy, then the FPA stall request has been ignored by the SAB-R2000A due to the occurrence of some exceptional event.

Exception/Interrupt Timing

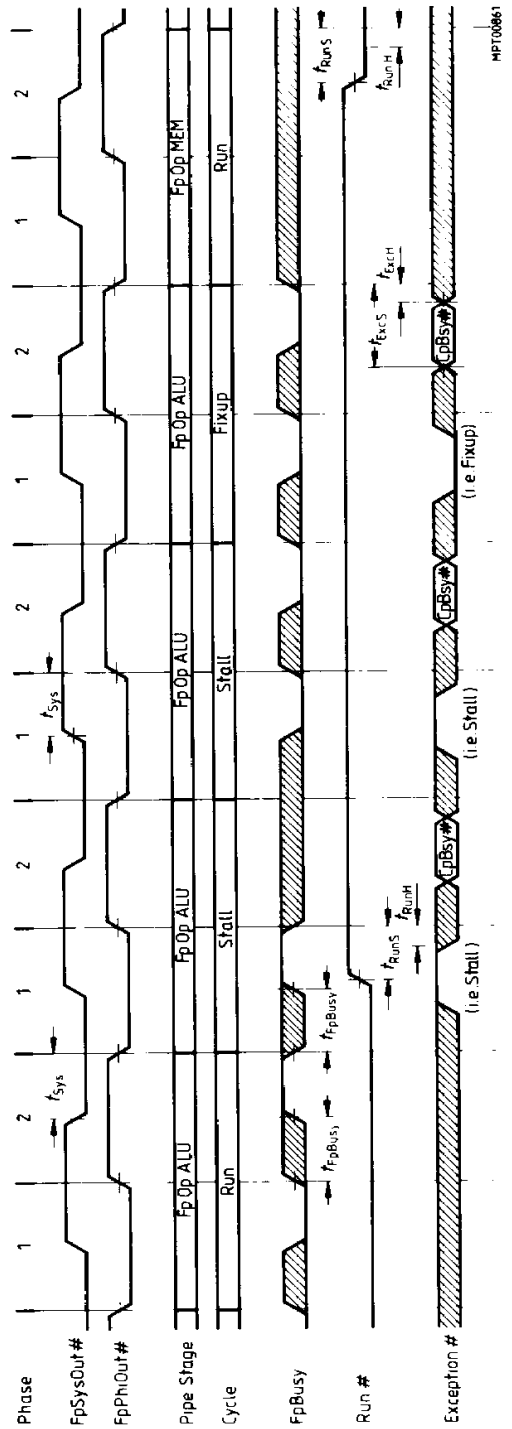
The SAB-R2010A signals exceptions to the SAB-R2000A through one of its interrupt inputs using the FpInt# output, refer to figure 16. The SAB-R2000A samples the interrupt inputs during phase 2 of every run cycle and final fixup cycle of a stall sequence. The FPA signals exceptions by asserting FpInt# during the ALU pipestage of the instruction causing the exception. If the SAB-R2000A takes the interrupt during that cycle, it signals interrupt grant (IntGr2M#) back to the FPA (via the Exception# signal) during the MEM pipestage of the exceptional instruction. Interrupt grant is signalled to the FPA on its Exception# input during phase 2 of the MEM pipestage.

The occurrence of any exception, including those caused by the FPA, is signalled to the FPA during the WB pipestage of the exception-causing instruction. The occurrence of an exception prevents any non-exception-related state from being committed within the FPA. This means that when an exception occurs which is not FPA related, execution in the FPA is suspended until the exception is resolved. The occurrence of an exception is signalled to the FPA on its Exception# input during phase 1 of the WB pipestage of the exceptional instruction. Figure 21 illustrates an Interrupt timing sequence.

Special Case

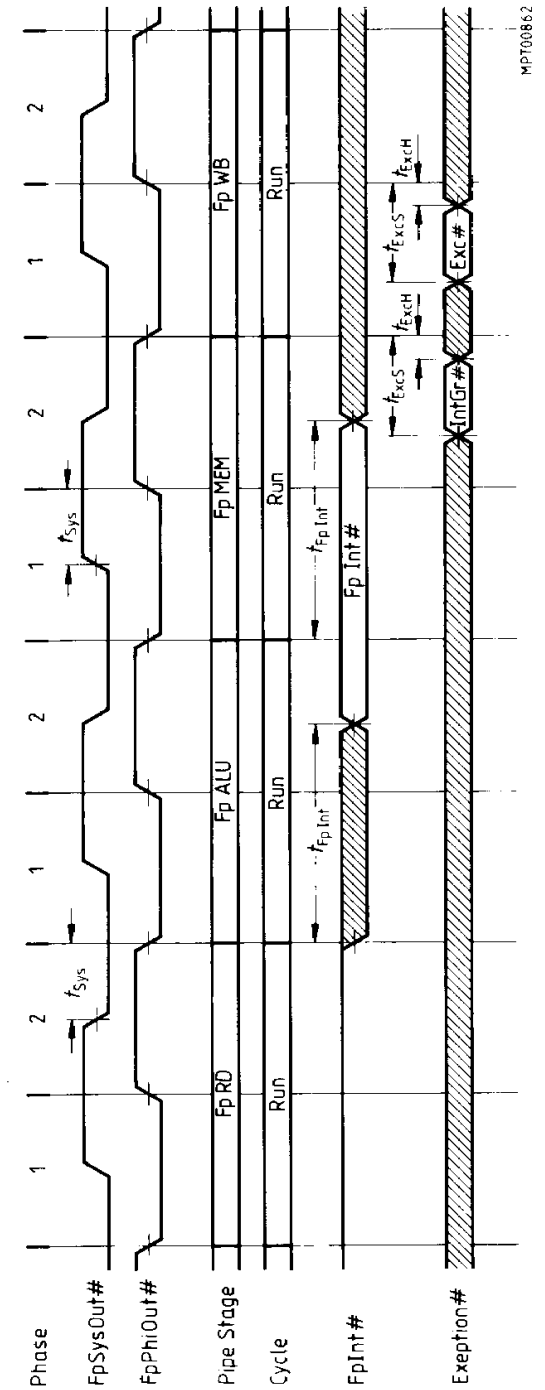
A special case of FPA – SAB-R2000A transfers is the MoveTo FPA Status register. When moves to this register occur the interface can be further affected via a change in the FpInt# or FpCond outputs. Figure 22 illustrates the timing of the FpCond and FpInt# outputs in conjunction with an FPA MoveTo instruction.

Figure 20
Busy Timing



Note: CpBusy# is the same as CpBusy2# in table 12.

Figure 21
Interrupt and Exception Timing



***Where:** IntGr# = IntGr2M# and Exc# = Exc1W# in table 12.*

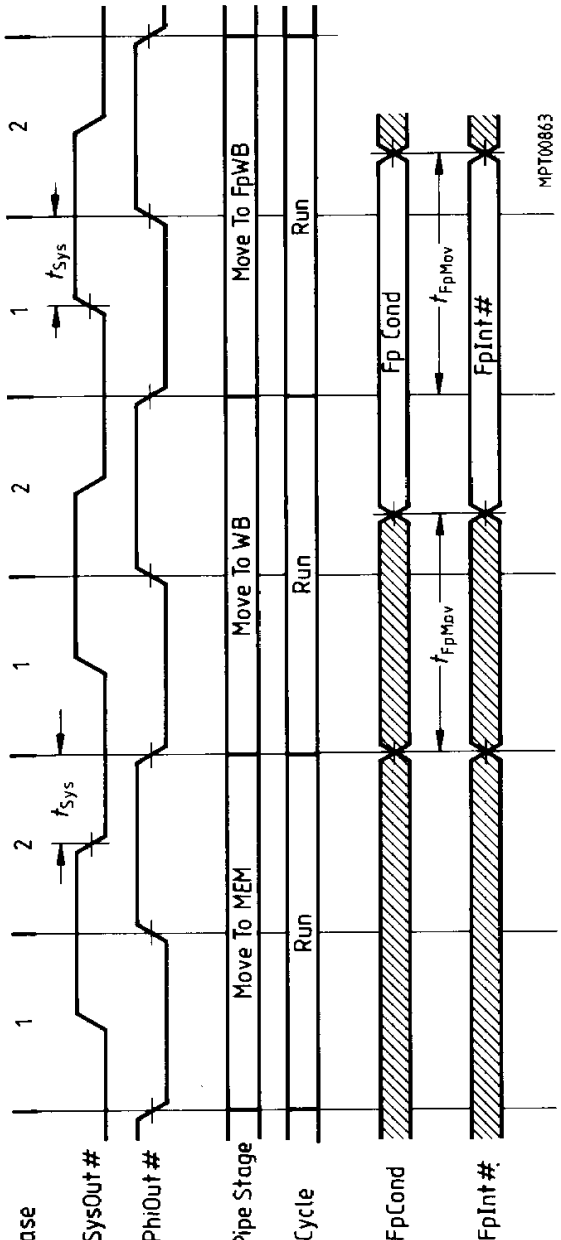


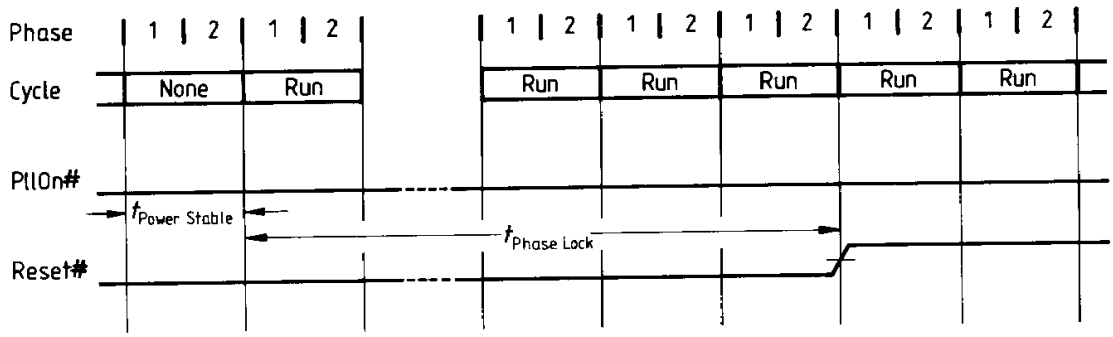
Figure 22
Move to FPA Status Timing

SAB-R2000A – FPA Timing Synchronization

The processor – coprocessor (SAB-R2000A – SAB-R2010A) system requires that there be minimum timing skew between the SAB-R2000A and FPA to operate at maximum speed. To facilitate deskewing, a phase lock loop is provided on the FPA.

This synchronization must be achieved during the reset period to ensure that clock skew is acceptably small when the first instruction is fetched. During the reset period, the SAB-R2010A's phase lock circuitry acquires and locks to the SAB-R2000A's output clock. For correct operation, the CPU – FPA system must remain in reset for 3000 clocks or 200 microseconds after power is stable, whichever is longer. If the phase lock mechanism is not enabled, the reset period can be shortened to 128 clock cycles after power is stable. Figure 23 illustrates the required reset sequence for the case where the phase lock mechanism is enabled.

Figure 23
Reset Sequence



MPT00864

Note: PllOn# must be asserted continuously after Reset# is deasserted for correct operation of the SAB-R2010A – SAB-R2000A system.