3164/3364
64-BIT FLOATING-POINT DATA PATH UNITS

November 1989

1. Features

64-BIT FLOATING-POINT DATA PATH

64-bit and 32-bit floating-point and 32-bit integer multiplier

64-bit and 32-bit floating-point and 32-bit integer ALU

64-bit and 32-bit divide/square root unit

Six-port 32-word by 64-bit register file

Six 64-bit internal buses

HIGH PERFORMANCE

100 ns, 75 ns, 60 ns, 50 ns and 40 ns clock cycle

Single-cycle throughput up to 50 MFLOPS

Two-cycle register-to-register latency

Three-cycle memory-to-memory latency

Four-cycle chained operation latency

High I/O bandwidth: up to 600 Mbytes/sec

FLEXIBLE I/O CONFIGURATIONS

Three 32-bit buses: 1 input, 1 output, 1 input/output (3364)

Single 32-bit I/O bus (3164) or single 64-bit I/O bus (3364)

FULL FUNCTION

Divide and square root operations

Single-cycle pipeline throughput for the following operations:

Multiply/Add	$XiRaYi \rightarrow Zi$ $XiRa+ Rb \rightarrow Rc$
Concurrent Multiply and Add Operations	$XiYi \rightarrow Z$ and $Ra + Rb \rightarrow Rc$
Sum of products (3364)	$\sum XiYi$
Product of sums (3364)	$\Pi(Xi + Yi)$

Compare, absolute value, format conversion
Integer, logical, shift, and min/max

FULL IEEE COMPLIANCE

Conforms fully, in a pipelined environment, to the *IEEE Standard for Binary Floating-Point Arithmetic*

FULLY INTERRUPTIBLE

HIGH INTEGRATION CMOS TECHNOLOGY WITH TTL I/O

COMPATIBLE WITH WEITEK XL-SERIES PROCESSOR FAMILY

2. Description

The 3164 and 3364 are 64-bit floating-point data path units designed for high-speed operation in a pipelined environment, while making possible full compliance with the *IEEE Standard For Binary Floating-Point Arithmetic(Std 754-1985)*, as well as full interruptibility during pipelined register-to-register operations. For the rest of this document, when a reference is made to a feature or attribute common to both the 3164 and 3364, they will both be referred to as "3x64."

The 3x64 is fabricated in CMOS technology; it integrates on a single chip all necessary arithmetic units: the independently-controlled floating-point multiplier and ALU; divide/square root unit (DSR); 32-word by 64-bit six-port register file; extensive status and control logic. See figure 1.

Unlike previous IEEE-compatible floating-point processors, the 3x64 makes possible full conformance to the *IEEE Standard For Binary Floating-Point Arithmetic* in a *pipelined* environment.

The 3x64 has a built-in six-port 32-deep by 64-bits wide register file. It can be bypassed on loads, stores and during register-to-register operations. This saves one cycle of latency in each case. The register file reduces the need for external components and increases performance.

The 3364 has three 32-bit ports: the bidirectional X port; the Y input port; and the Z output port. It is intended for microprogrammable building-block applications where both "raw" floating-point performance and high I/O bandwidth are important. The 3364 may be used in either a three-32-bit-bus configuration or a single-64-bit-bus configuration. It is packaged in a 168-pin grid array. The 3364 is intended for scientific problem-solving in vector and array processors, supermini- and minisupercomputers, and high performance engineering workstations.

2. Description, continued

The 3164 has a single bidirectional 32-bit bus. It is intended for microprogrammable building-block applications which are compute-bound and cost-sensitive and in which high I/O bandwidth is of secondary importance. The on-chip register file of the 3164 minimizes I/O traffic for many applications. The 3164 is packaged in a 144-pin grid array. It is aimed primarily at numeric coprocessor and graphics applications.

The 3164 is used with the WEITEK XL-8137 32-bit integer processing unit (IPU) and the XL-8136 32-bit program sequencing unit (PSU) to create a fast general-purpose numeric processor, the XL-8164. The 3364 can also be used with the IPU and the PSU to create an even faster processor, the XL-8364, which has a 64-bit data bus. Full development system support is available for XL-Series processors, including C and FORTRAN 77 compilers. The 3164 and 3364 are functionally identical to the 3164 and 3364, respectively, but are used in modes specific to the XL-Series. Please see appendix A for more information on the 3x64 parts.

2.1. Functional Simulator

WEITEK offers a functional (pin-level) simulator for the 3x64. The simulator can aid the user in the understanding of the 3x64 and in developing code for it. The simulator allows the user to specify stimuli to the chip—input operands, control signals and status register mode bits—and to examine responses: results and status outputs, as well as the contents of register file registers and status registers.

The simulator was written (in C) to be event-driven; if needed, it can be linked into the user's own simulation environment.

A license agreement is required to purchase the simulator.

2. Description, continued

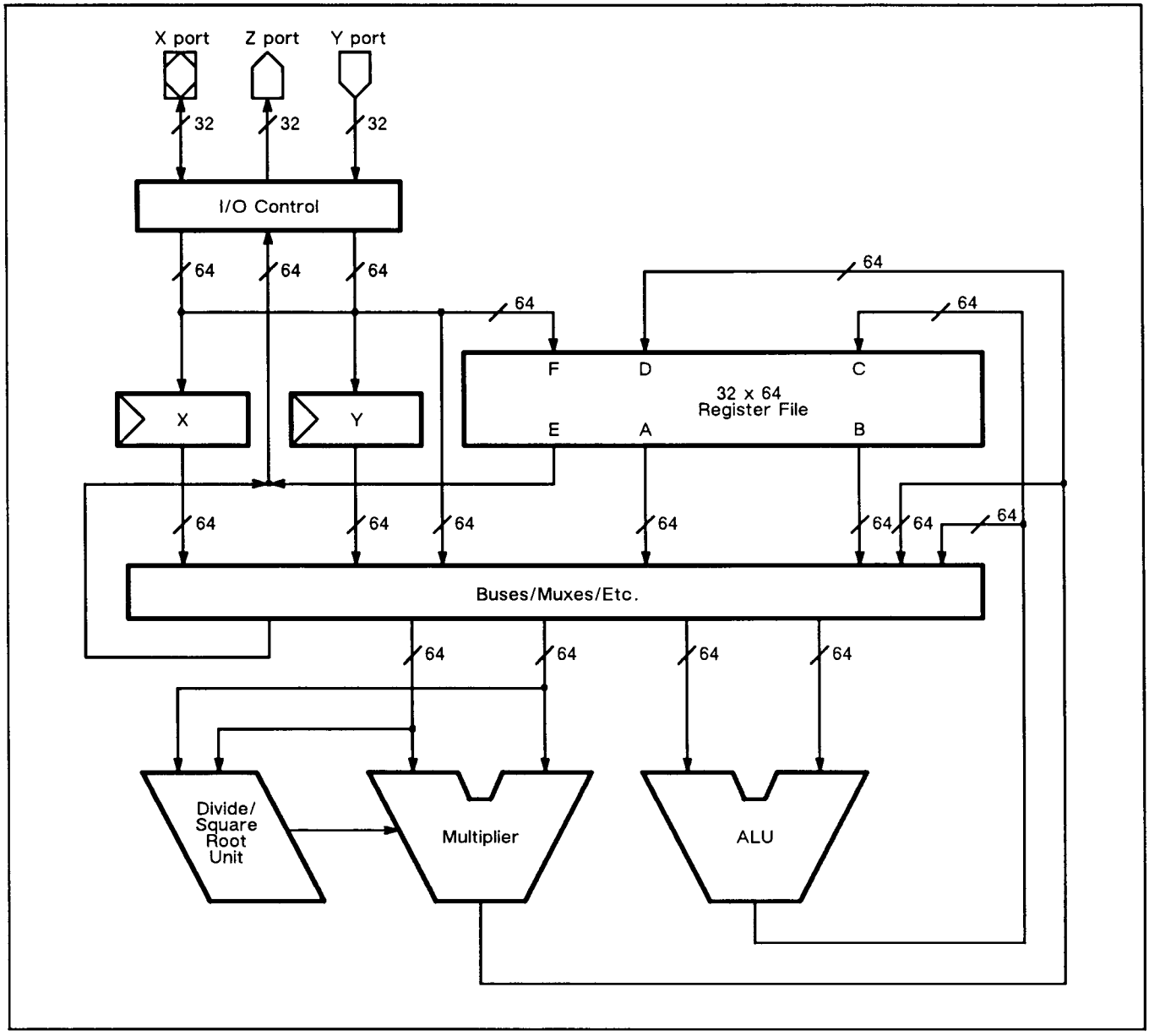


Figure 1. 3164/3364 conceptual block diagram

3. Architecture

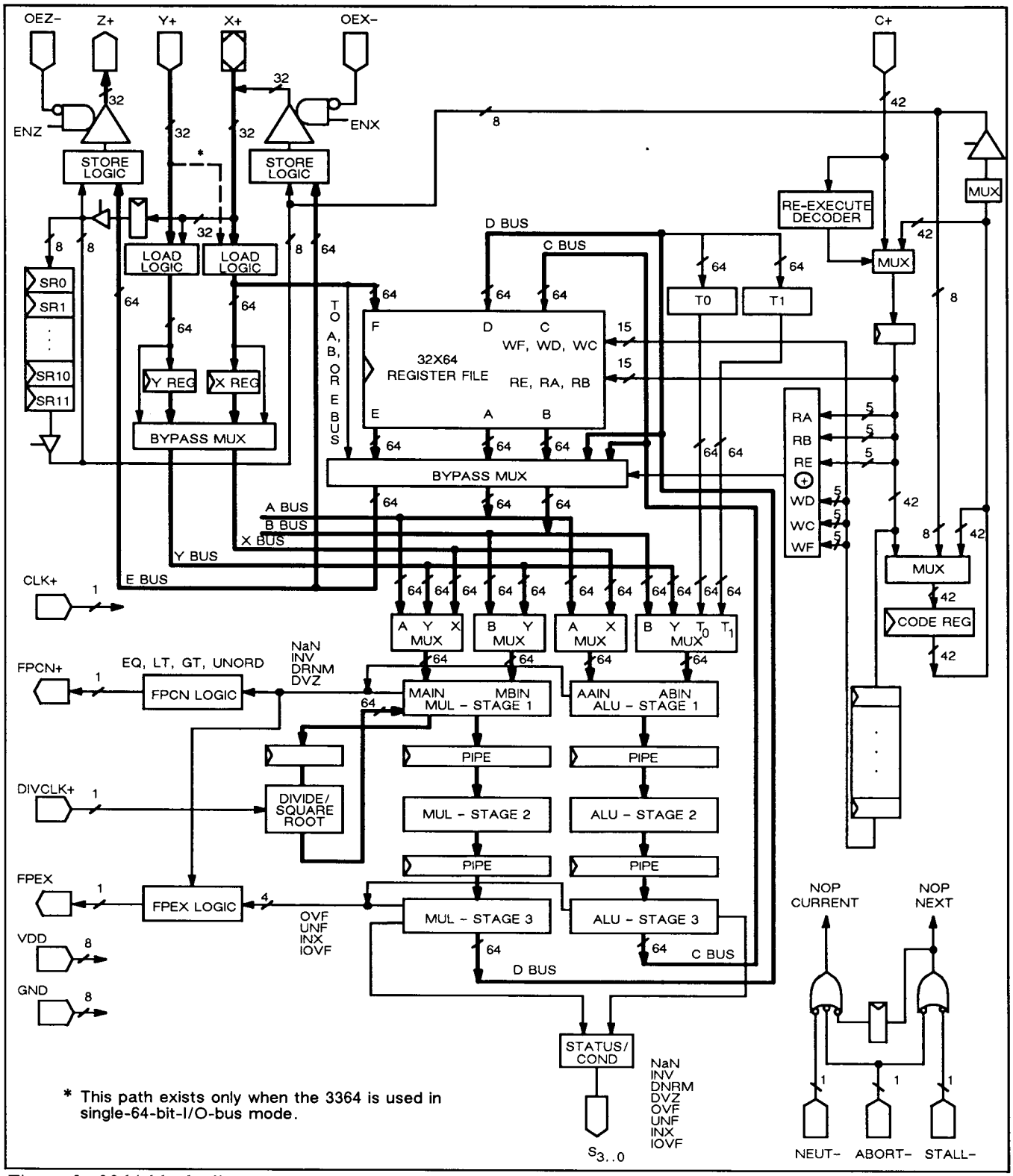


Figure 2. 3364 block diagram

3. Architecture, continued

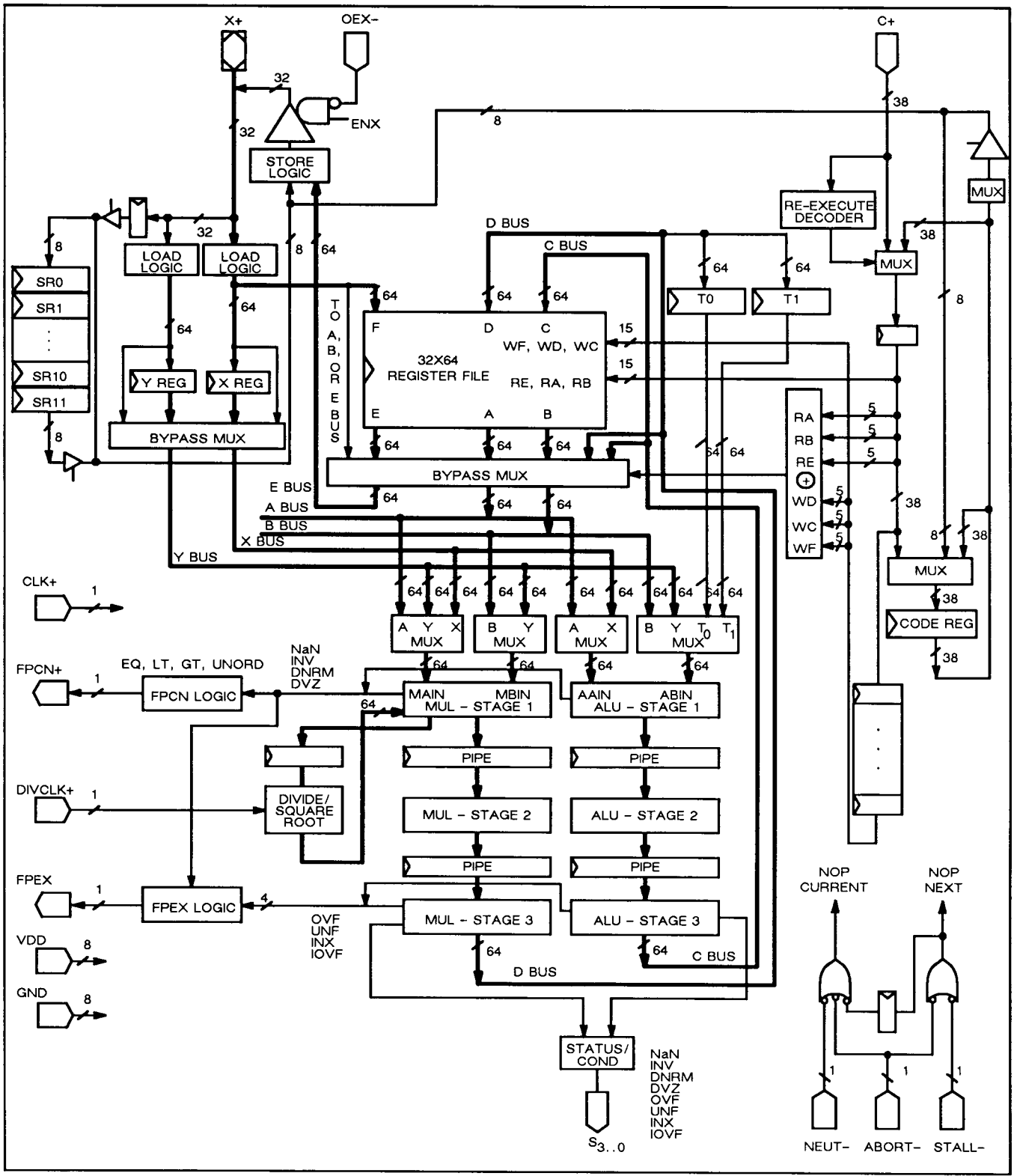


Figure 3. 3164 block diagram

4. Signal Description

Signals marked with a minus sign (–) after their names are active low. All other signals are active high.

X PORT INPUT/OUTPUT

The 32-bit $X_{31..0}$ port is a bidirectional data bus. In *single-pump mode*, input data is sampled on the rising edges of CLK; in *double-pump mode* it is sampled on both the rising and falling edges of CLK. In single-pump mode, output data is available after the rising edge of CLK; in double-pump mode, 32-bit halves of a 64-bit word are available after both the rising and falling edges of CLK.

Data transfers are controlled by the XCNT field of the code word, $C_{7..4}$; a nop (0000) in this field causes the device to ignore the X port input and tri-states the output. Single- or double-pump mode is controlled by the I/O mode field in the status register, $SR1_{4..0}$. Status and code registers are also loaded and stored through the X port. The X port may be set to a high-impedance state, independently of all other controls, by the asynchronous OEX– signal.

Y PORT INPUT

The 32-bit $Y_{31..0}$ port is a data input bus. It is available only on the 3364. In single-pump mode, input data is sampled on the rising edges of CLK; in double-pump mode on both the rising and falling edges of CLK. Single- or double-pump mode is controlled by the I/O mode field in the status register, $SR1_{4..0}$. Data transfers are controlled by the YCNT field of the code word, $C_{3..2}$; a nop (00) in this field causes the device to ignore the Y port input, except when in 64-bit I/O mode.

Z PORT OUTPUT

The 32-bit $Z_{31..0}$ port is a data output bus. It is available only on the 3364. In single-pump mode, output data is available after the rising edge of CLK; in double-pump mode, 32-bit halves of a 64-bit word are available after both the rising and falling edges of CLK. Single- or double-pump mode is controlled by the I/O mode field in the status register, $SR1_{4..0}$. Data transfers are controlled by the ZCNT field of the code word, $C_{1..0}$; a nop (00) in the ZCNT field tri-states the Z port, except when in 64-bit I/O mode. Stores of status and code registers are driven to the Z port as well. The Z port may be set to a high-impedance state, independently of all other controls, by the asynchronous OEZ– signal.

C PORT INPUT

The $C_{41..0}$ port is used as a code input bus. Instructions are latched on the rising edge of CLK. The 3364 has a 42-bit code port, $C_{41..0}$; the 3164 has a 38-bit code port $C_{41..4}$. The structure of the code word is diagrammed in figure 4 and repeated in figure 95. The mnemonics used in this diagram are explained in section 17.1. The code word consists of the I/O portion ($C_{12..0}$) and the operation portion ($C_{41..13}$), which are independent. Note that everything that needs to be known about a register-to-register operation can be specified at once in the operation portion of the instruction, at the time the instruction is issued.

OEX– INPUT

X port output enable input. OEX–, when high, asynchronously places the X port output in a high-impedance state. When low, the X port output enable is controlled by the XCNT field of the code word.

OEZ– INPUT

Z port output enable input. OEZ–, when high, asynchronously places the Z port in a high-impedance state. When low, the Z port output enable is controlled by the ZCNT field of the code word.

NEUT– INPUT

Neutralize input. Cancels the effect of the *current instruction*. Typically used for delayed branches and interrupts. Sampled on the rising edge of the cycle.

STALL– INPUT

Stall input. Cancels the effect of the *next instruction*. Typically used as a "not ready" line from the code memory. Sampled on the rising edge of the cycle.

ABORT– INPUT

Abort input. Cancels the effect of the current and next instructions. Typically used as a "not ready" line from the data store. Sampled on the rising edge of the cycle.

4. Signal Description, continued

FPEX OUTPUT

Floating-point exception output. FPEX signals the occurrence of an enabled exception. Whenever an enabled exception occurs, the FPEX output is asserted, and the FPEX TAKEN bit in the status register, $SR11_7$, is set. This bit stays set until explicitly cleared. Clearing it is the only way to reset the FPEX output, when in sticky mode (see below). FPEX has two modes associated with it: delayed/undelayed and sticky/pulsed.

FPEX delay is controlled by a mode bit in the status register, $SR1_5$. In undelayed mode, whenever an enabled exception occurs, FPEX is asserted at the end of the cycle in which the exception has occurred. In delayed mode, FPEX is asserted in the beginning of the cycle following the one in which the exception has occurred. The delayed FPEX mode eases system timing constraints.

FPEX sticky/pulsed mode is controlled by another status register mode bit, $SR0_6$. In sticky mode, whenever FPEX is asserted as a result of an enabled exception, it stays asserted until the FPEX TAKEN bit in the status register is explicitly cleared. In pulsed mode, FPEX stays asserted for one cycle only. It returns to de-asserted state in the beginning of the following cycle. In addition, when FPEX is in sticky mode, it has negative polarity (active low); when in pulsed mode, it has positive polarity (active high).

FPCN OUTPUT

Floating-point condition output. FPCN is asserted if the condition specified in a compare instruction is true. It is asserted on the cycle following the occurrence of the condition and will maintain state until changed by the result of the subsequent compare instruction.

S OUTPUT

$S_{3..0}$ is a four-bit status output. It provides encoded and prioritized status of multiplier, ALU and divide/square root operations. See section 11.

CLK INPUT

Clock input.

DIVCLK INPUT

Divide clock input. DIVCLK is used by the divide/square root unit. DIVCLK frequency can be the same as, or twice the CLK frequency. Divide and square root operations complete faster when DIVCLK frequency is twice the CLK frequency.

VCC

All VCC pins must be connected to the 5.0 V power supply.

GND

All ground pins must be connected to system ground.

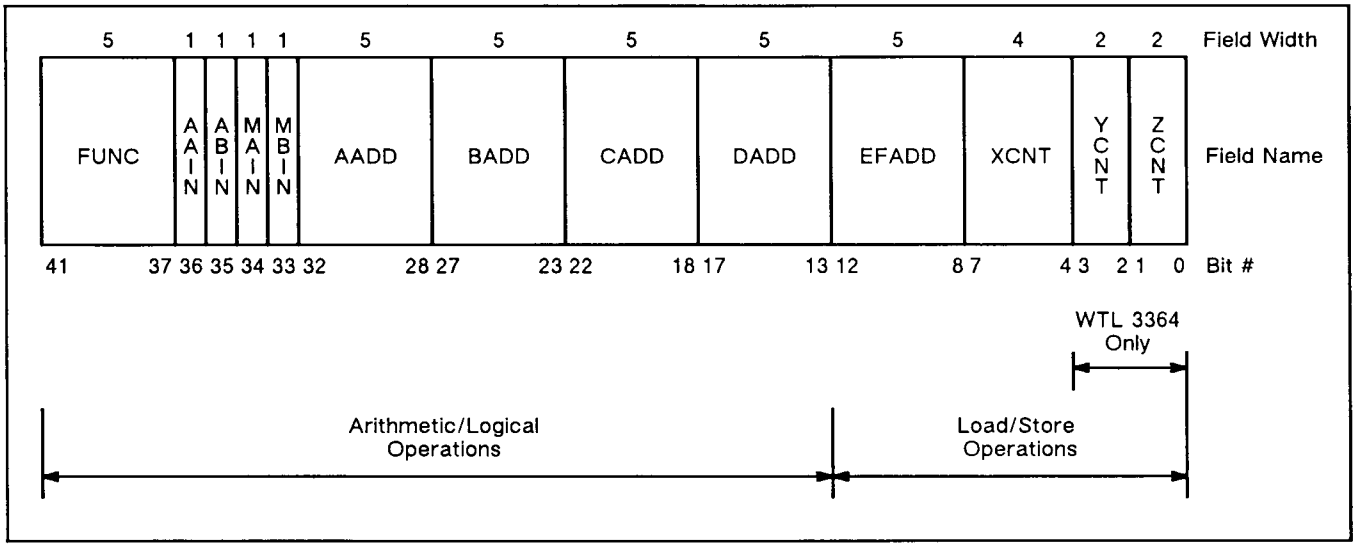


Figure 4. Code word format

5. Input/Output

The 3x64 provides a flexible interface for a variety of memory systems. The flexibility stems from the user's ability to select one of three I/O configurations in conjunction with several load/store alternatives. The available I/O configurations will be discussed first, followed by a discussion of load/store modes.

5.1. I/O Configurations

There are three basic I/O configurations: configurations A, B, and C (see figure 5).

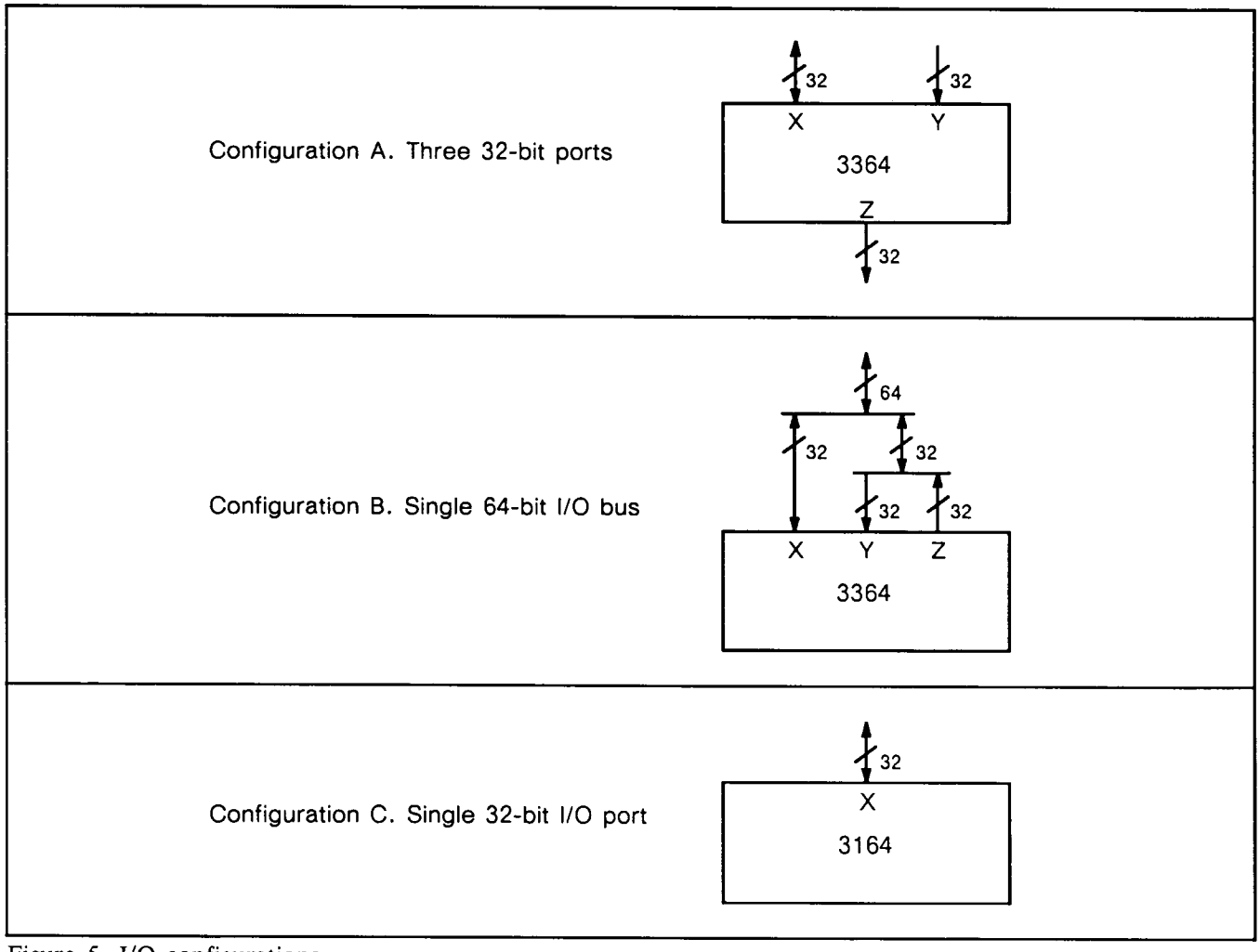


Figure 5. I/O configurations

5.1. I/O Configurations, continued

CONFIGURATION A

The 3364 used in this configuration has three 32-bit ports: one bidirectional port, X; one input port, Y; and one output port, Z. This configuration is applicable only to the 3364 and is used to maximize I/O bandwidth and throughput in vector processing applications.

CONFIGURATION B

The 3364 used in this configuration has a single 64-bit bidirectional port. This configuration is applicable only to the 3364; it may be used in coprocessor applications. To use the 3364 in Configuration B, the X, Y, and Z ports must be connected as shown in figure 5; and the YCNT and ZCNT fields must be set to zero. Then the XCNT field specifies load/store operations, as shown in figure 9.

CONFIGURATION C

This configuration is applicable only to the the 3164: it has a single 32-bit bidirectional port, X. The 3164 is typically used in compute-bound coprocessor applications.

5.2. Single-Pumping and Double-Pumping

Each of the ports is capable of transferring 32-bit data in either one of two modes: single- or double-pump. In single-pump mode, a 32-bit data word is transferred on the rising edge of CLK. If greater I/O bandwidth is needed than is achievable in single-pump mode, double-pump mode may be used. In double-pump mode, two 32-bit data words are transferred on each CLK cycle: one on each edge of the clock.

Single- or double-pump mode is selected, along with other specifications for load and store operations, via a 5-bit field in the status register, $SR1_{4..0}$. These controls are discussed in section 5.5. Note that single-pump mode on loads implies single-pump mode on stores; likewise, double-pump mode on loads implies double-pump mode on stores. In configurations A and C, both single- and double-pump modes are supported. In Configuration B, only single-pumping is supported.

5.3. Data Types

When performing I/O operations, the user must distinguish integers from other data types.

The following four data types are supported for all I/O operations:

- 32-bit floating-point, conforms to the IEEE standard
- 64-bit floating-point, conforms to the IEEE standard
- 32-bit two's complement integer
- 64-bit logical

More information on data types is found in section 6.1.

5.4. I/O Ports

X PORT

The 32-bit X port is the only bidirectional data bus in the device. As such, this port is used in both the 3164 and in 3364. Data transfers through this port are controlled by the 4-bit XCNT field in the code word, $C_{7..4}$, and, in some operations, by the function select fields: FUNC, MAIN, MBIN, AAIN, ABIN, and BADD.

Loads through the X port may be routed to several destinations:

- The X register
- The Y register
- To a register in the register file (address specified by the EFADD field of the code word)
- To both the register file and to the A bus or B bus (through bypassing). Register file bypassing is discussed in section 6.5.
- To the status register (one byte at a time)
- To the code register (one byte at a time)

5.4. I/O Ports, continued

When data is loaded into the X or Y registers it is automatically available on the X or Y bus, respectively, as an operand. The X and Y registers are static in that their values do not change unless explicitly loaded with new values. It is possible to load the X or Y register once and then re-use the loaded data as an operand in subsequent instructions.

Stores to the X port can come from the following sources:

- The register file (address specified by the EFADD field of the code word)
- The result of an ALU operation
- The result of a multiplier or a DSR operation
- The status register (one byte at a time)
- The code register (one byte at a time)

Storing a multiplier or ALU result involves register file bypassing which is discussed more fully in section 6.5.1.

Y PORT

The 32-bit Y port is a data input bus. It is available only on the 3364. Loads through the Y port are controlled by the YCNT field of the code word, $C_{3..2}$, or when the device is used in Configuration B, by the XCNT field.

Loads through the Y port can be routed to one of three destinations:

- The Y register
- The X register (Configuration B only)
- The register file (Configuration B only. Address is specified by the EFADD field of the code word)

When data is loaded into the Y register, it is automatically available on the Y bus as an operand.

Z PORT

The 32-bit Z port is the data output bus. It is available only on the 3364. Stores to the Z port are controlled through the ZCNT field of the code word, $C_{1..0}$, and, in some operations, by the function select fields: FUNC, MAIN, MBIN, AAIN, ABIN, and BADD.

Sources of stores to the Z port are identical to those for the X port, when the X port is used as an output:

- The register file (address specified by the EFADD field of the code word)
- The result of an ALU operation
- The result of a multiplier or a DSR operation
- The status register (one byte at a time)
- The code register (one byte at a time)

5.5. Load/Store Operations and Their Control

Figures 7–11 explain load and store operations for each configuration, in both single- and double-pump modes, as appropriate. Figure 12 provides a summary for all three configurations. Figure 6 explains mnemonics used in these figures.

XR	X register
YR	Y register
ER	Register in the register file, addressed by the EFADD field of the code word. This is the register that is stored through the E read port of the register file.
FR	Register in the register file, addressed by the EFADD field of the code word. This is the register that is loaded through the F write port of the register file.
LS	Least-significant half of a register
MS	Most-significant half of a register. Used for most-significant half of a double-precision floating-point number or for single-precision floating-point number.
INT	Integer data type
, (comma)	Separates two operations taking place at one I/O port on two successive clock edges as a result of one code word, when double-pump mode is used.
; (semicolon)	In configuration B, separates simultaneous loads through the X and Y ports or simultaneous stores through the X and Z ports.
/ (slash)	Separates operations involving two registers taking place simultaneously; for example, it is possible to simultaneously load the same data into two registers: the X register and a register in the register file, addressed by the EFADD field of the code word.

Figure 6. Load/store mnemonics

5.5. Load/Store Operations and Their Control, continued

Control Field		Configuration A Three 32-bit buses, single-pump	
		I/O operation	Comments
XCNT			
0	0000	NOP	Ignore X port input and tri-state its output
1	0001	ER LS → X PORT	Store LS of ER register
2	0010	ER MS → X PORT	Store MS of ER register
3	0011	ER INT → X PORT	Store an integer from ER register
4	0100	X PORT → YR LS	Load LS of the Y Register
5	0101	X PORT → XR/FR LS	Load LS of both the X Register and FR register
6	0110	X PORT → XR/FR MS	Load MS of both the X Register and FR register
7	0111	X PORT → XR/FR INT	Load an integer into both the X Register and FR register
8	1000	X PORT → YR MS	Load MS of the Y Register
9	1001	X PORT → FR LS	Load LS of FR register
10	1010	X PORT → FR MS	Load MS of FR register
11	1011	X PORT → FR INT	Load an integer into FR register
12	1100	X PORT → YR INT	Load an integer into the Y Register
13	1101	X PORT → XR LS	Load LS of the X Register
14	1110	X PORT → XR MS	Load MS of the X Register
15	1111	X PORT → XR INT	Load an integer into the X Register
YCNT			
0	00	NOP	Ignore Y port input
1	01	Y PORT → YR LS	Load LS of the Y Register
2	10	Y PORT → YR MS	Load MS of the Y Register
3	11	Y PORT → YR INT	Load an integer into the Y Register
ZCNT			
0	00	NOP	Tri-state Z port
1	01	ER LS → Z PORT	Store LS of ER register
2	10	ER MS → Z PORT	Store MS of ER register
3	11	ER INT → Z PORT	Store an integer from ER register

Figure 7. Load/store operations and their control for Configuration A

5.5. Load/Store Operations and Their Control, continued

Control Field		Configuration A Three 32-bit buses, double-pump	
		I/O operation	Comments
XCNT			
0	0000	NOP	Ignore X port input and tri-state its output
1	0001	ER LS, MS → X PORT	Store LS and MS of ER register
2	0010	ER MS, MS → X PORT	Store MS of ER register
3	0011	ER INT, INT → X PORT	Store an integer from ER register
4	0100	X PORT → YR LS, MS	Load both halves of the Y Register
5	0101	X PORT → XR/FR LS, MS	Load both halves of both the X Register and FR register
6	0110	X PORT → XR/FR NOP, MS	Load MS of both the X Register and FR register
7	0111	X PORT → XR/FR INT, NOP*	Load an integer into both the X Register and FR register
8	1000	X PORT → YR NOP, MS	Load MS of the Y Register
9	1001	X PORT → FR LS, MS	Load both halves of FR register
10	1010	X PORT → FR NOP, MS	Load MS of FR a register
11	1011	X PORT → FR INT, NOP*	Load an integer into FR register
12	1100	X PORT → YR INT, NOP*	Load an integer into the Y Register
13	1101	X PORT → XR LS, MS	Load both halves of the X Register
14	1110	X PORT → XR NOP, MS	Load MS of the X Register
15	1111	X PORT → XR INT, NOP*	Load an integer into the X Register
YCNT			
0	00	NOP	Ignore Y port input
1	01	Y PORT → YR LS, MS	Load both halves of the Y Register
2	10	Y PORT → YR NOP, MS	Load MS of the Y Register
3	11	Y PORT → YR INT, NOP	Load an integer into the Y Register
ZCNT			
0	00	NOP	Tri-state Z port
1	01	ER LS, MS → Z PORT	Store both halves of ER register
2	10	ER MS, MS → Z PORT	Store MS of ER register
3	11	ER INT, INT → Z PORT	Store an integer from ER register

* Integers must always be loaded on the rising edge of the clock. This table assumes undelayed load mode. In delayed load mode, the entries in the I/O operation column should read "NOP, INT".

Figure 8. Load/store operations and their control for Configuration A

5.5. Load/Store Operations and Their Control, continued

Control Field		Configuration B Single 64-bit I/O bus, single-pump only	
		I/O operation	Comments
XCNT			
0	0000	NOP	Ignore inputs and tri-state outputs
1	0001	ER LS; MS → X PORT; Z PORT	Store both halves of ER register
2	0010	ER MS; MS → X PORT; Z PORT	Store MS of ER register to X and Z ports
3	0011	ER INT; INT → X PORT; Z PORT	Store an integer from ER register to both X and Z ports
4	0100	X PORT → YR LS; Y PORT → YR MS	Load LS of the Y Register from X port, MS of the Y Register from Y port
5	0101	X PORT → XR/FR LS; Y PORT → XR/FR MS	Load LS of the X Register and FR register from X port, Load MS of the X Register and FR register from Y port
6	0110	Y PORT → XR/FR MS	Load MS of the X Register and FR register from Y port
7	0111	X PORT → XR/FR INT	Load an integer from X port into the X Register and FR register
8	1000	Y PORT → YR MS	Load MS of the Y Register from Y port
9	1001	X PORT → FR LS; Y PORT → FR MS	Load LS of FR register from X port, MS from Y port
10	1010	Y PORT → FR MS	Load MS of FR register from Y port
11	1011	X PORT → FR INT	Load an integer into FR register from X port
12	1100	X PORT → YR INT	Load an integer into the Y Register from X port
13	1101	X PORT → XR LS; Y PORT → XR MS	Load LS of the X Register from X port, MS from Y port
14	1110	Y PORT → XR MS	Load MS of the X Register from Y port
15	1111	X PORT → XR INT	Load an integer into the X Register from X port
YCNT			
0	00		Must be set to zero
ZCNT			
0	00		Must be set to zero

Figure 9. Load/store operations and their control for Configuration B

5.5. Load/Store Operations and Their Control, continued

Control Field		Configuration C Single 32-bit I/O bus, single-pump	
		I/O operation	Comments
XCNT			
0	0000	NOP	Ignore X port input and tri-state its output
1	0001	ER LS → X PORT	Store LS of ER register
2	0010	ER MS → X PORT	Store MS of ER register
3	0011	ER INT → X PORT	Store an integer from ER register
4	0100	X PORT → YR LS	Load LS of the Y Register
5	0101	X PORT → XR/FR LS	Load LS of both the X Register and FR register
6	0110	X PORT → XR/FR MS	Load MS of both the X Register and FR register
7	0111	X PORT → XR/FR INT	Load an integer into both the X Register and FR register
8	1000	X PORT → YR MS	Load MS of the Y Register
9	1001	X PORT → FR LS	Load LS of FR register
10	1010	X PORT → FR MS	Load MS of FR register
11	1011	X PORT → FR INT	Load an integer into FR register
12	1100	X PORT → YR INT	Load an integer into the Y Register
13	1101	X PORT → XR LS	Load LS of the X Register
14	1110	X PORT → XR MS	Load MS of the X Register
15	1111	X PORT → XR INT	Load an integer into the X register

Figure 10. Load/store operations and their control for Configuration C

5.5. Load/Store Operations and Their Control, continued

Control Field		Configuration C Single 32-bit I/O bus, double-pump	
		I/O operation	Comments
XCNT			
0	0000	NOP	Ignore X port input and tri-state its output
1	0001	ER LS, MS → X PORT	Store LS and MS of ER register
2	0010	ER MS, MS → X PORT	Store MS of ER register
3	0011	ER INT, INT → X PORT	Store an integer from ER register
4	0100	X PORT → YR LS, MS	Load both halves of the Y Register
5	0101	X PORT → XR/FR LS, MS	Load both halves of both the X Register and FR register
6	0110	X PORT → XR/FR NOP, MS	Load MS of both the X Register and FR register
7	0111	X PORT → XR/FR INT, NOP*	Load an integer into both the X Register and FR register
8	1000	X PORT → YR NOP, MS	Load MS of the Y Register
9	1001	X PORT → FR LS, MS	Load both halves of FR register
10	1010	X PORT → FR NOP, MS	Load MS of FR a register
11	1011	X PORT → FR INT, NOP*	Load an integer into FR register
12	1100	X PORT → YR INT, NOP*	Load an integer into the Y Register
13	1101	X PORT → XR LS, MS	Load both halves of the X Register
14	1110	X PORT → XR NOP, MS	Load MS of the X Register
15	1111	X PORT → XR INT, NOP*	Load an integer into the X Register

* Integers must always be loaded on the rising edge of the clock. This table assumes undelayed load mode. In delayed load mode, the entries in the I/O operation column should read "NOP, INT".

Figure 11. Load/store operations and their control for Configuration C

5.5. Load/Store Operations and Their Control, continued

Control Field		Configurations A and C Three 32-bit ports (WTL 3364) and single 32-bit I/O port (WTL 3164)		Configuration B Single 64-bit I/O port(3364)
		Single-pump	Double-pump	Single-pump only
XCNT				
0	0000	NOP	NOP	NOP
1	0001	ER LS → X PORT	ER LS, MS → X PORT	ER LS; MS → X PORT; Z PORT
2	0010	ER MS → X PORT	ER MS, MS → X PORT	ER MS; MS → X PORT; Z PORT
3	0011	ER INT → X PORT	ER INT, INT → X PORT	ER INT; INT → X PORT; Z PORT
4	0100	X PORT → YR LS	X PORT → YR LS, MS	X PORT → YR LS; Y PORT → YR MS
5	0101	X PORT → XR/FR LS	X PORT → XR/FR LS, MS	X PORT → XR/FR LS; Y PORT → XR/FR MS
6	0110	X PORT → XR/FR MS	X PORT → XR/FR NOP, MS	Y PORT → XR/FR MS
7	0111	X PORT → XR/FR INT	X PORT → XR/FR INT, NOP*	X PORT → XR/FR INT
8	1000	X PORT → YR MS	X PORT → YR NOP, MS	Y PORT → YR MS
9	1001	X PORT → FR LS	X PORT → FR LS, MS	X PORT → FR LS; Y PORT → FR MS
10	1010	X PORT → FR MS	X PORT → FR NOP, MS	Y PORT → FR MS
11	1011	X PORT → FR INT	X PORT → FR INT, NOP*	X PORT → FR INT
12	1100	X PORT → YR INT	X PORT → YR INT, NOP*	X PORT → YR INT
13	1101	X PORT → XR LS	X PORT → XR LS, MS	X PORT → XR LS; Y PORT → XR MS
14	1110	X PORT → XR MS	X PORT → XR NOP, MS	Y PORT → XR MS
15	1111	X PORT → XR INT	X PORT → XR INT, NOP*	X PORT → XR INT
YCNT				
0	00	NOP	NOP	Must be set to zero
1	01	Y PORT → YR LS	Y PORT → YR LS, MS	
2	10	Y PORT → YR MS	Y PORT → YR NOP, MS	
3	11	Y PORT → YR INT	Y PORT → YR INT, NOP	
ZCNT				
0	00	NOP	NOP	Must be set to zero
1	01	ER LS → Z PORT	ER LS, MS → Z PORT	
2	10	ER MS → Z PORT	ER MS, MS → Z PORT	
3	11	ER INT → Z PORT	ER INT, INT → Z PORT	

* Integers must always be loaded on the rising edge of the clock. This table assumes undelayed load mode. In delayed load mode, the entries in the I/O operation column should read "NOP, INT".

Figure 12. Summary of load/store operations and their control

5.5. Load/Store Operations and Their Control, continued

5.5.1. LOAD/STORE MODE CONTROL

The 3x64 has been designed for flexible memory interface in a variety of system configurations. Both load and store modes are selected through the $SR1_{4..0}$ field in the status register. The allowable load/store modes are listed below, followed by a timing diagram of each mode.

$SR1_{4..0}$ Decimal	Binary	Load Mode	Store Mode	Comment
0	00000	SP-U	SP-U	Single-pump undelayed load and store
2	00010	SP-U/CB	SP-U/CB	Single-pump undelayed load and store, Configuration B
4	00100	SP-D	SP-U	Single-pump delayed load and single-pump undelayed store
6	00110	SP-D/CB	SP-U/CB	Single-pump delayed load and single-pump undelayed store, Configuration B
8	01000	SP-U	SP-DD	Single-pump undelayed load and single-pump delayed-data store
9	01001	DP-U	DP-DD	Double-pump undelayed load and double-pump delayed-data store
10	01010	SP-U/CB	SP-DD/CB	Single-pump undelayed load and single-pump delayed-data store, Configuration B
12	01100	SP-D	SP-DD	Single-pump delayed load and single-pump delayed-data store
13	01101	DP-D	DP-DD	Double-pump delayed load and double-pump delayed-data store
14	01110	SP-D/CB	SP-DD/CB	Single-pump delayed load and single-pump delayed-data store, Configuration B
16	10000	SP-U	SP-DS	Single-pump undelayed load and single-pump delayed store
18	10010	SP-U/CB	SP-DS/CB	Single-pump undelayed load and single-pump delayed store, Configuration B
20	10100	SP-D	SP-DS	Single-pump delayed load and single-pump delayed store
22	10110	SP-D/CB	SP-DS/CB	Single-pump delayed load and single-pump delayed store, Configuration B

The codes that are not listed are reserved and must not be used.

Figure 13. Load and store modes

5.5. Load/Store Operations and Their Control, continued

5.5.2. NOTES ON LOAD/STORE TIMING DIAGRAMS

Figures 14 through 32 contain timing diagrams for each load and store mode. By examining the figures, the reader will gain sufficient understanding to be able to create similar diagrams under different assumptions. The assumptions and notation used in these figures are listed below.

- Two-cycle-latency mode is used ($SR0_7 = 0$). This mode is explained in sections 7.3 and 12.3.
- Bypassing is enabled ($SR1_6 = 1$). Bypassing is explained in sections 6.5 and 12.3.
- Two sequential operations are shown in either the multiplier or ALU, identified as OP1 and OP2. They are shown to demonstrate how to load operands for immediate use by the multiplier or ALU and how to store results immediately after they are available at the output of the multiplier or ALU.
- Operations OP1 and OP2 are dyadic, of the form

  X1 OP1 Y1 → R1
  X2 OP2 Y2 → R2

  where the values X1 and X2 come from the X register and values Y1 and Y2 from the Y register.
- Both the operands (X1, Y1) for OP1 and (X2, Y2) for OP2 — as well as the results R1 and R2 — are double-precision. The least-significant half of the operands and the results is labeled LS; the most-significant half MS. The double-precision data type was chosen as it is the most complex one, and because operations with other data types can be understood and derived from it.

Single-precision floating-point data type is handled identically with the most-significant half of the double-precision data type. Integer data must always be clocked on the *rising* edge of CLK. Refer to the corresponding double-precision figure; the timing for the integer data type corresponds to whatever half (LS or MS) of the double-precision word is aligned with the rising edge of the clock. Logical data is handled exactly the same as double-precision floating-point.

Register file writes are shown for completeness. They are shown on the loads to demonstrate the earliest time that the result of the operation on loaded operands can be written into the register file. On stores, they are shown to demonstrate when the result of an operation is written into the register file and may be bypassed to the output for the store operation. The registers being written are identified as RN, where N may range from 0 to 31. This notation is also used to identify the result being stored at the outputs.

The code bus is shown split into two parts. The first one, $C_{12..0}$, consists of the EFADD, XCNT, YCNT, and ZCNT fields (see figures 7–12), and controls I/O (load/store) operations. The second one, $C_{41..13}$, consists of the FUNC, AAIN, ABIN, MAIN, MBIN, and the AADD, BADD, CADD, DADD register file addresses, and specifies the operation itself. See figure 4. It can be seen from the timing diagrams that these two parts of the code bus are independent: the loading of the operands and/or storing of the results need not be a part of the same code word that specifies the operation which uses these operands and/or produces the results. The operands may be loaded well in advance, and the results can be stored well after the operation.

5.5.3. LOAD MODES

From a timing standpoint, four load modes are possible. None of them affects the operation's latency. Figure 14 provides a conceptual comparison among them. Figures 15–24 describe each mode under assumptions listed in the previous section.

SINGLE-PUMP UNDELAYED LOAD (FIGURES 15–17)

In single-pump undelayed load, the load and the operation portions of the instruction can be specified in the same code word. This code word and the corresponding data are clocked into their respective inputs simultaneously. The specified operation begins execution in the same cycle in which it was clocked in.

SINGLE-PUMP DELAYED LOAD (FIGURES 18–20)

Delayed load means that the actual loading of data is delayed relative to the clocking-in of the load instruction on $C_{12..0}$ inputs. Since stage 1 begins execution in the same cycle in which the operation portion of the instruction is clocked in, and since both double-precision operands are necessary for the operation to take place, the operation portion of the instruction may be clocked in on the same clock edge that loads the second half of the double-precision operands.

Note that in Configuration B (single 64-bit I/O port), only single-pump mode is supported.

5.5. Load/Store Operations and Their Control, continued

DOUBLE-PUMP UNDELAYED LOAD (FIGURES 21-22)

The double-pump mode is similar to its single-pump counterpart in that the load and the operation portions of the instruction can be loaded on the same rising clock edge. The first halves (LS) of data operands are also loaded on this rising edge. The difference between the two modes is that the second halves of the data operands (MS) are loaded on the falling edge of the same cycle. Because both double-precision operands are available in the second half of the cycle in which they are loaded, the arithmetic units (stage 1) can begin execution on the same cycle when the operands are loaded thus reducing the total latency by one cycle. This load mode facilitates single-cycle throughput in double-precision vector operations.

This mode may not be used with undelayed FPEX.

DOUBLE-PUMP DELAYED LOAD (FIGURES 23-24)

The double-pump delayed load is similar to its single-pump counterpart in that the loading of the operands is delayed relative to the clocking-in of the load portion of the instruction on $C_{12..0}$ inputs. However, in the double-pump mode the delay is shorter. When the load instruction is clocked in on the rising edge of cycle 1, the first halves (LS) of the double-precision operands are loaded on the falling edge of cycle 1, and the second halves (MS) on the rising edge of cycle 2. Thus both operands are available in cycle 2, and the operation portion of the instruction can be clocked in on the rising edge of clock 2. This mode, like the double-pump undelayed load, facilitates single-cycle throughput in vector operations.

EFFECTS OF NEUT-, STALL-, ABORT- ON LOAD OPERATIONS

A NEUT- or ABORT- stops delayed loads but cannot stop an undelayed load that was clocked in on the current cycle, because the data has already been written into the register file or the X or Y register. STALL-, however, eliminates the effect of a load in any mode.

5.5.4. STORE MODES

From a timing standpoint, four store modes are possible. Figure 25 provides a conceptual comparison among them. Figures 26-32 describe each mode under assumptions listed in the section 5.5.2.

Store modes may be classified into single-pump and double-pump. There are three single-pump modes: undelayed, delayed-data and delayed. There is one double-pump store mode: delayed-data. The name of the store mode is assigned depending on when the store instruction is clocked in on $C_{12..0}$ pins and when the result becomes available at the outputs $X_{31..0}$ and/or $Z_{31..0}$ relative to the completion of the operation in the last stage (stage 3) of the multiplier or ALU. See figure 25.

SINGLE-PUMP UNDELAYED STORE (FIGURES 26-27)

If the operation is clocked in on the rising edge of cycle 1, the result of the operation will be available at the output of stage 3 in the first half of cycle 3. In order to store this result in the second half of this cycle, the store instruction must be clocked in on the rising edge of cycle 3. The result is output via the C or D buses and the E port bypass mux. This mode has the lowest latency.

SINGLE-PUMP DELAYED-DATA STORE (FIGURES 28-29)

The timing relationship between the completion of an operation in stage 3 and the clocking-in of the store instruction on $C_{12..0}$ inputs is the same as in the undelayed store case; however, the actual storing of the result at the outputs is delayed by one cycle, thus increasing the latency by one cycle. On the positive side, the output delay is shorter.

SINGLE-PUMP DELAYED STORE (FIGURES 30-31)

In this mode bypassing is automatically disabled on stores. Therefore, delayed stores are possible only from the register file. The timing relationship between the clocking-in of the store instruction on the $C_{12..0}$ inputs and the actual store taking place in the same cycle is the same as in the undelayed case; however the store instruction is clocked in on the rising edge following the availability of the result at the output of stage 3.

5.5. Load/Store Operations and Their Control, continued

DOUBLE-PUMP DELAYED-DATA STORE (FIGURE 32)

If stage 3 produces the result in the first half of cycle 3, the store operation must be clocked in on the rising edge of cycle 3. The store of the least-significant half of the result will take place in the second half of cycle 3 and the store of the most-significant half in the first half of cycle 4.

Note that double-pumping is supported only for configurations A and C.

EFFECTS OF NEUT–, STALL–, AND ABORT– ON STORE OPERATIONS

If a store instruction is clocked in on the rising edge of cycle N, then:

STALL– or ABORT–, asserted during cycle N – 1, will eliminate the effects of any store operation.

NEUT– or ABORT–, asserted during cycle N, will eliminate the effects of a store instruction only if it is single-pump delayed-data store.

5.5. Load/Store Operations and Their Control, continued

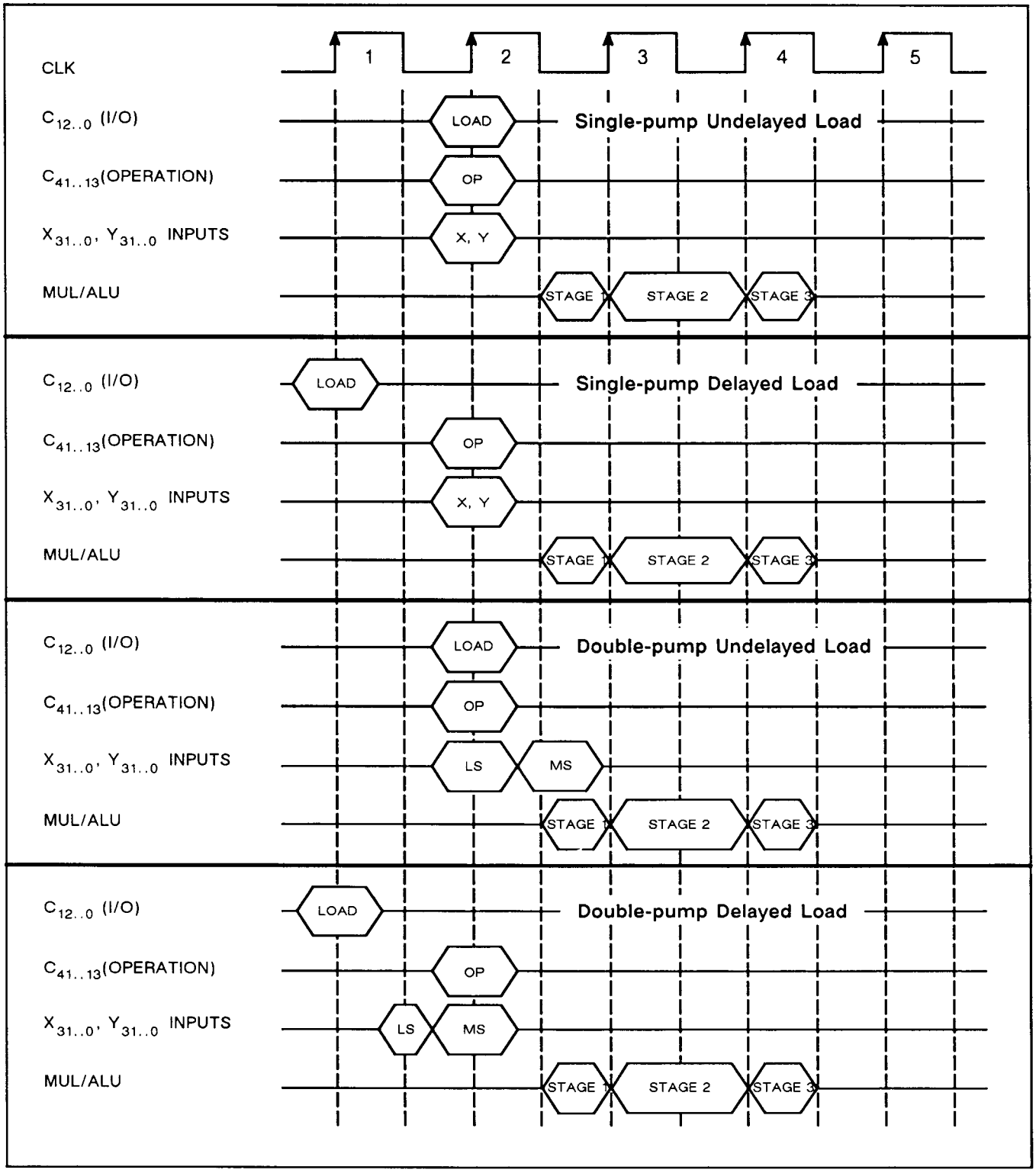


Figure 14. Load timing: conceptual diagram for comparison of load modes

5.5. Load/Store Operations and Their Control, continued

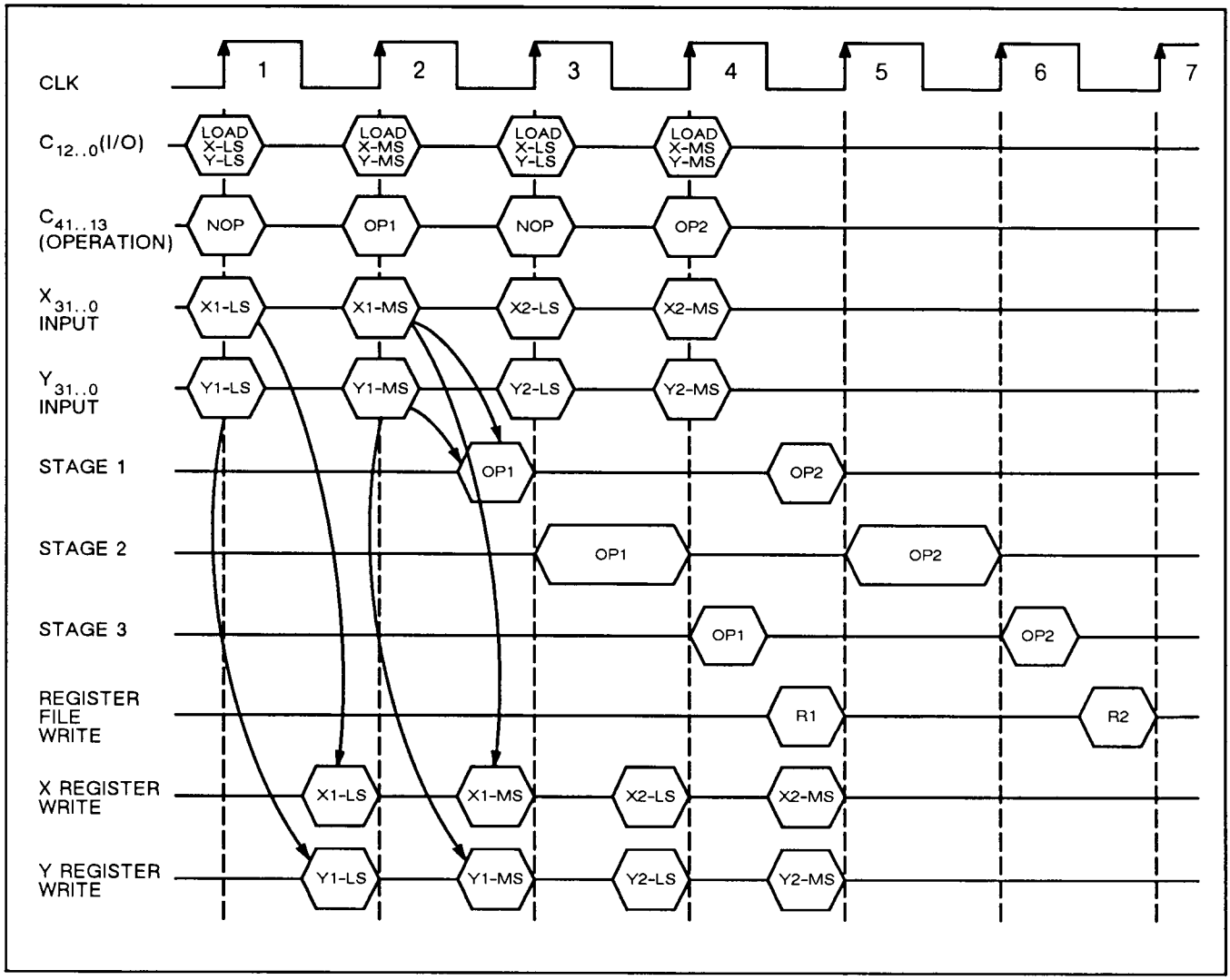


Figure 15. Single-pump undelayed load—Configuration A

5.5. Load/Store Operations and Their Control, continued

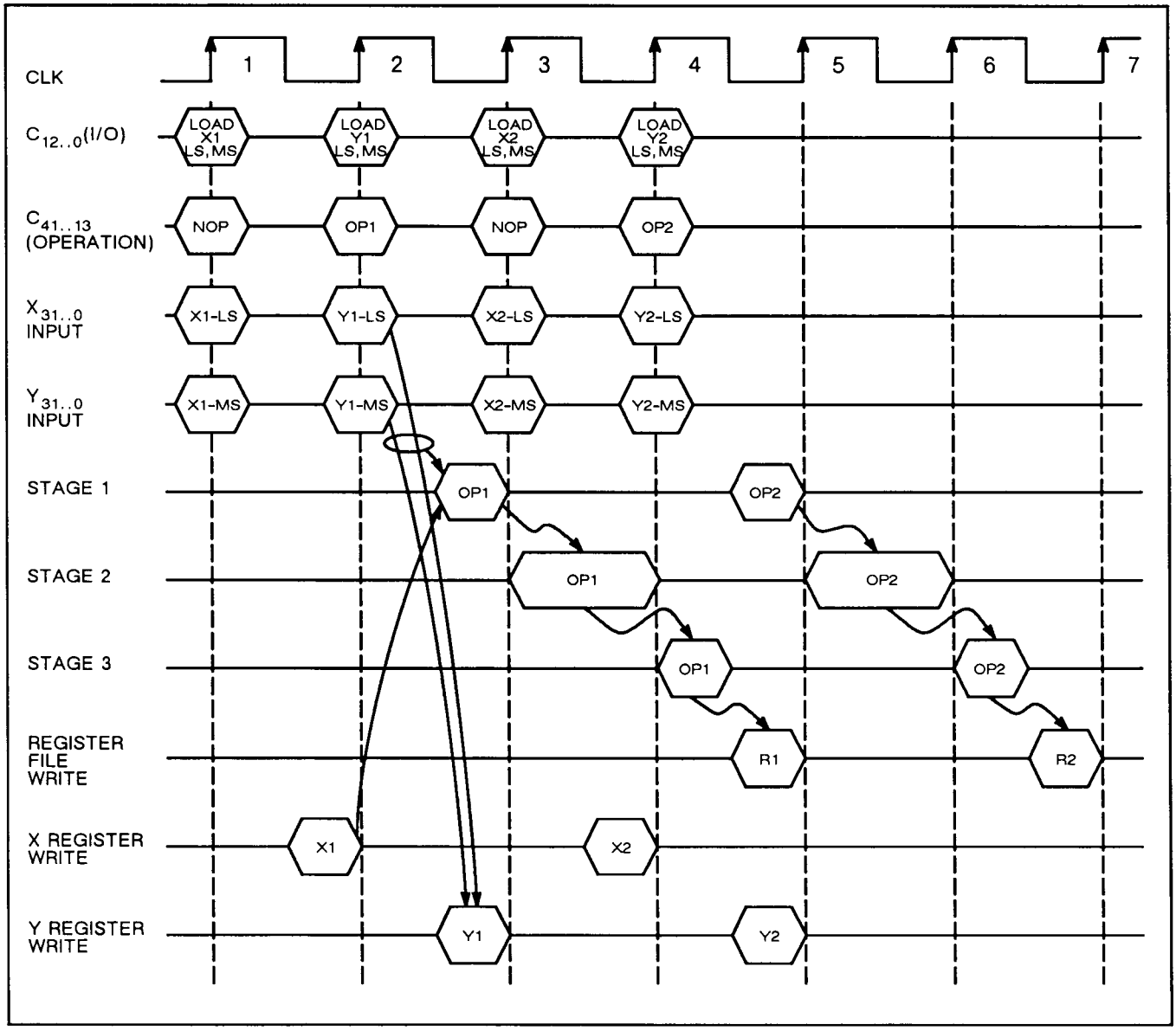


Figure 16. Single-pump undelayed load—Configuration B

5.5. Load/Store Operations and Their Control, continued

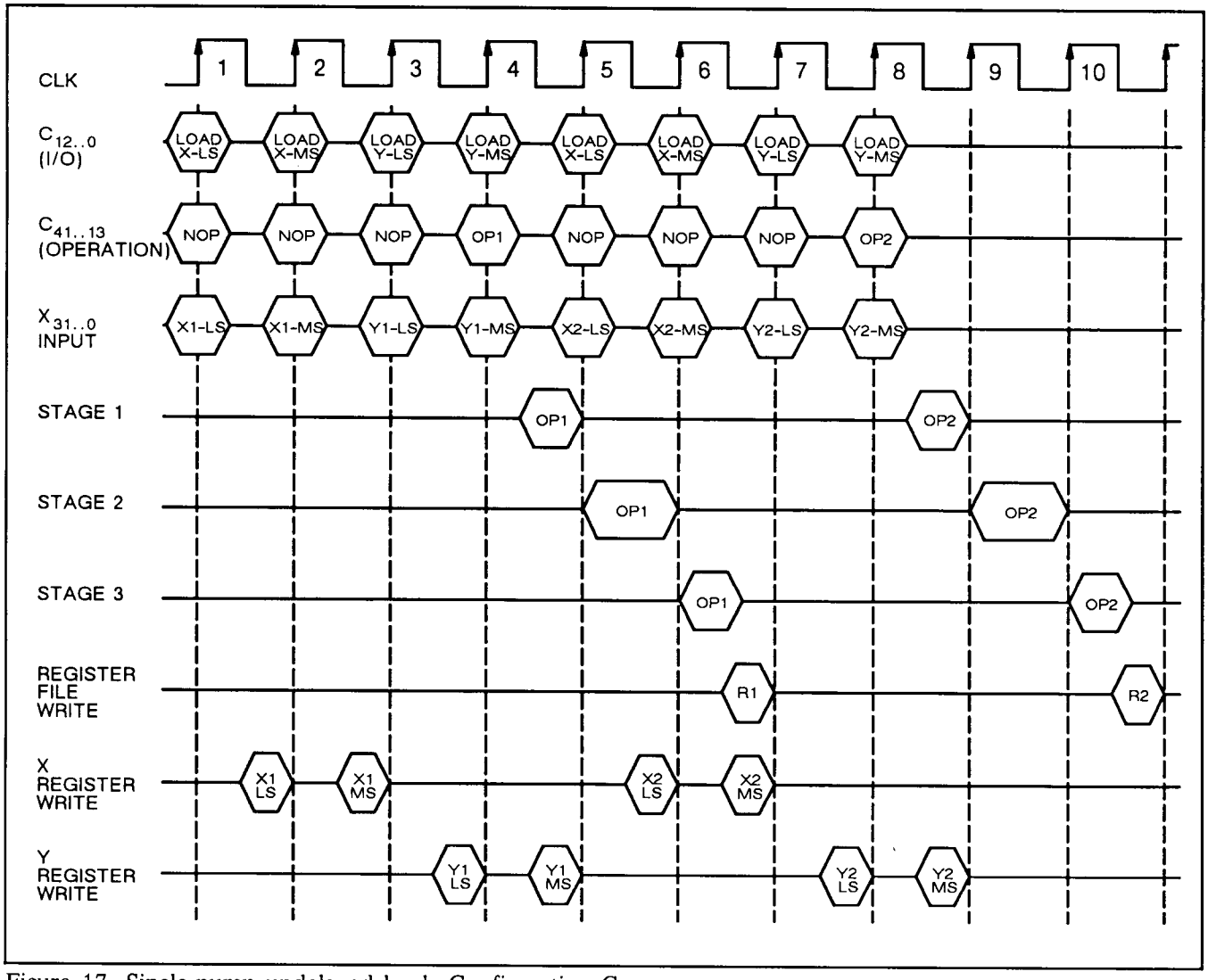


Figure 17. Single-pump undelayed load—Configuration C

5.5. Load/Store Operations and Their Control, continued

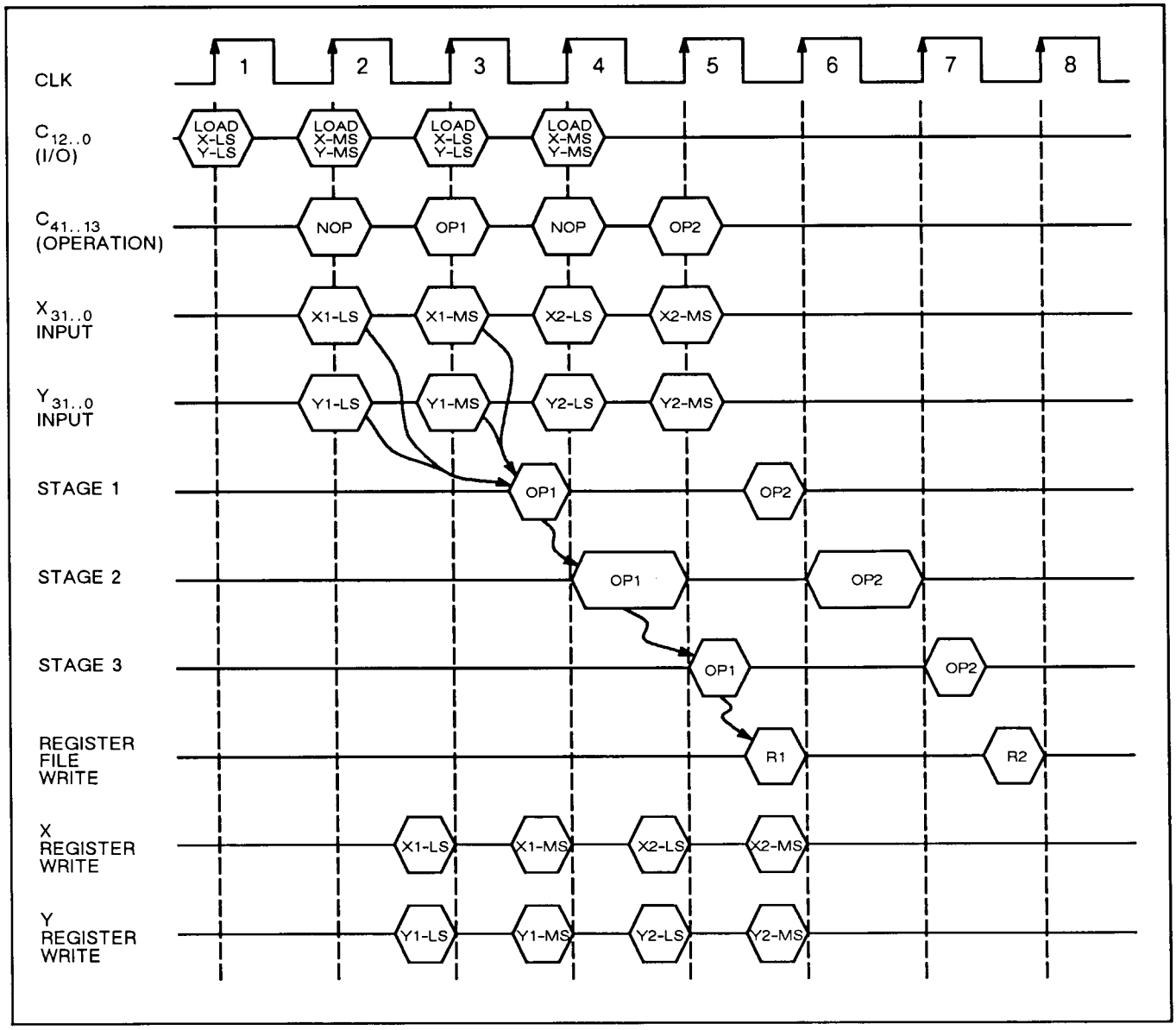


Figure 18. Single-pump delayed load—Configuration A

5.5. Load/Store Operations and Their Control, continued

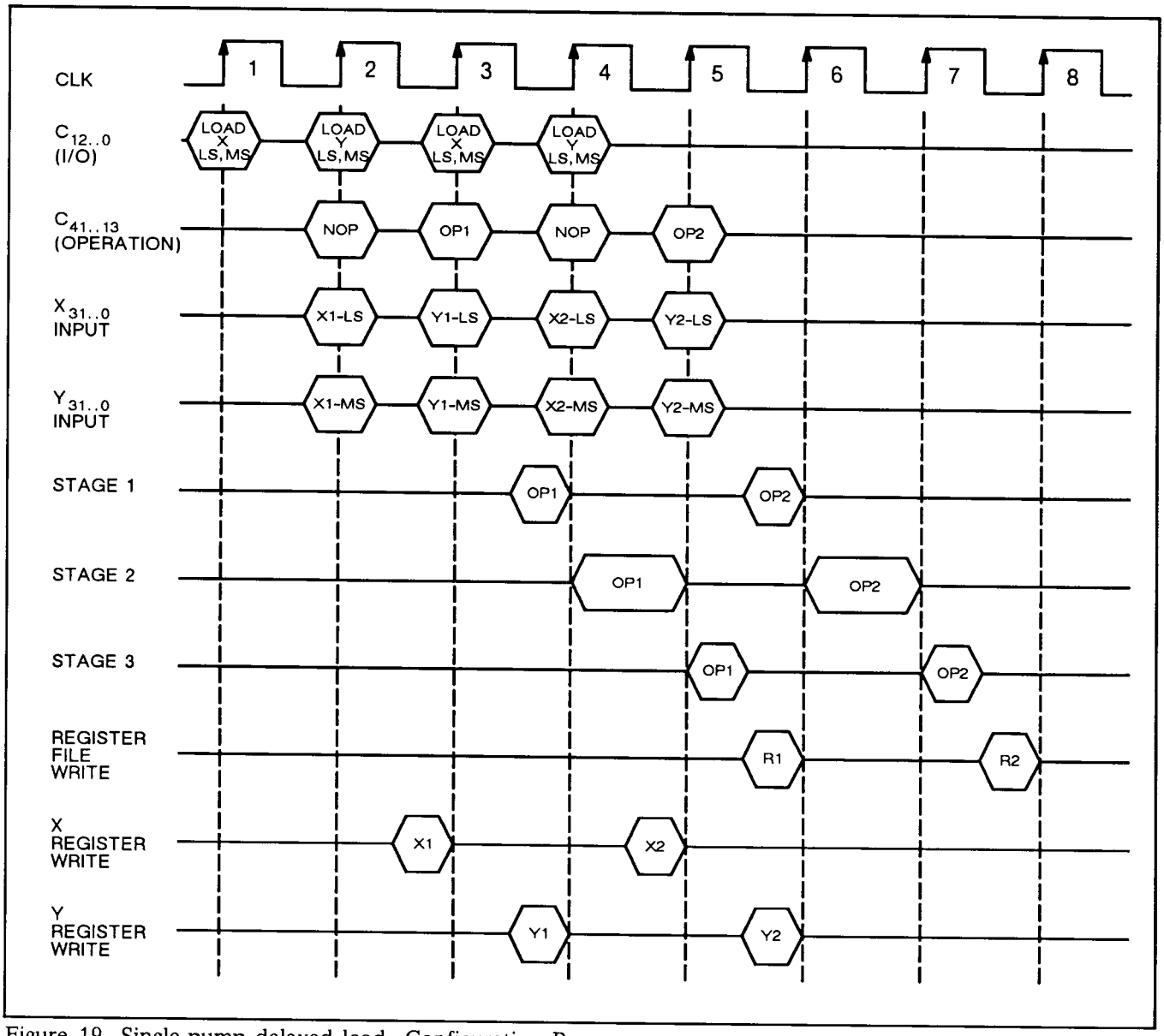


Figure 19. Single-pump delayed load—Configuration B

5.5. Load/Store Operations and Their Control, continued

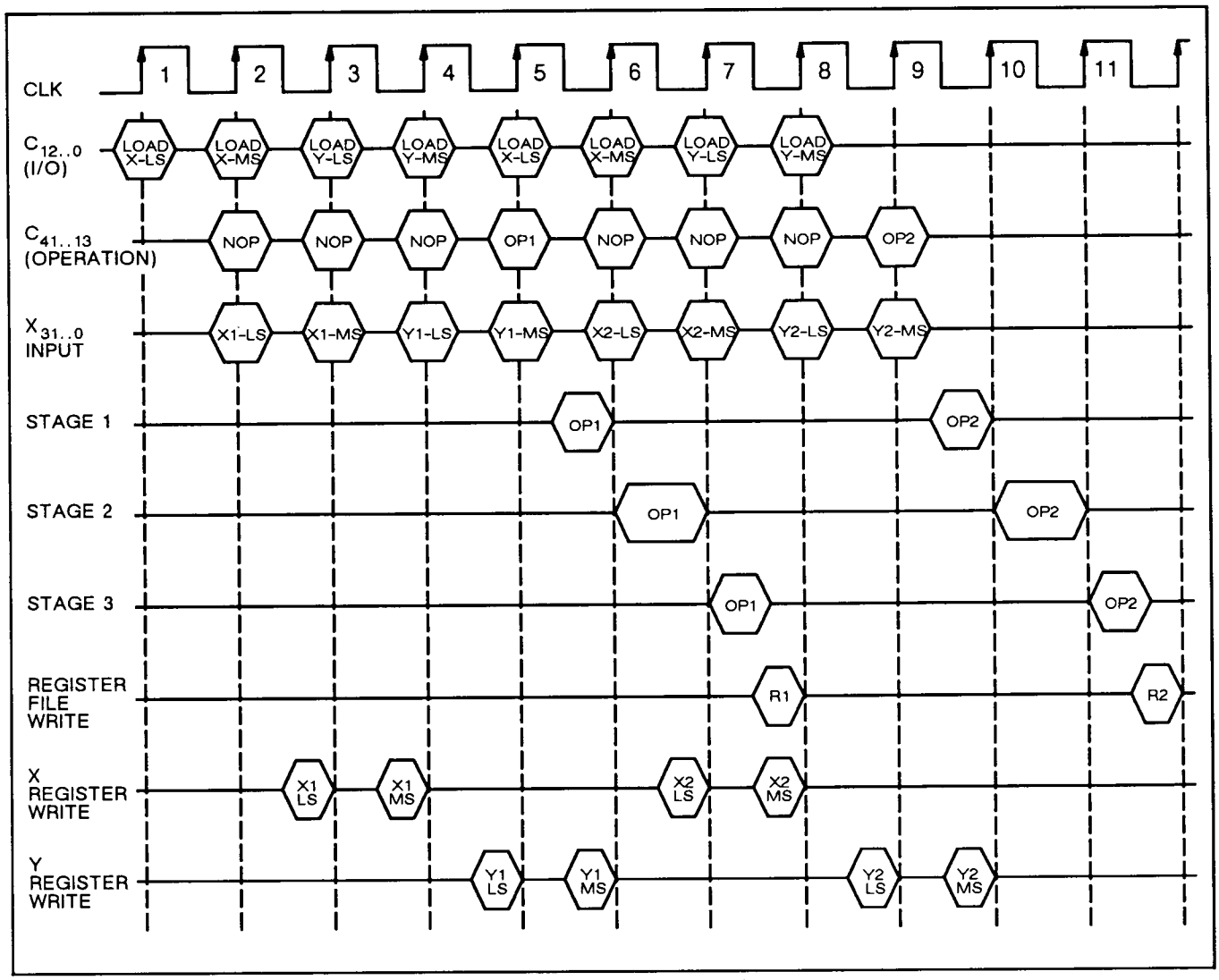
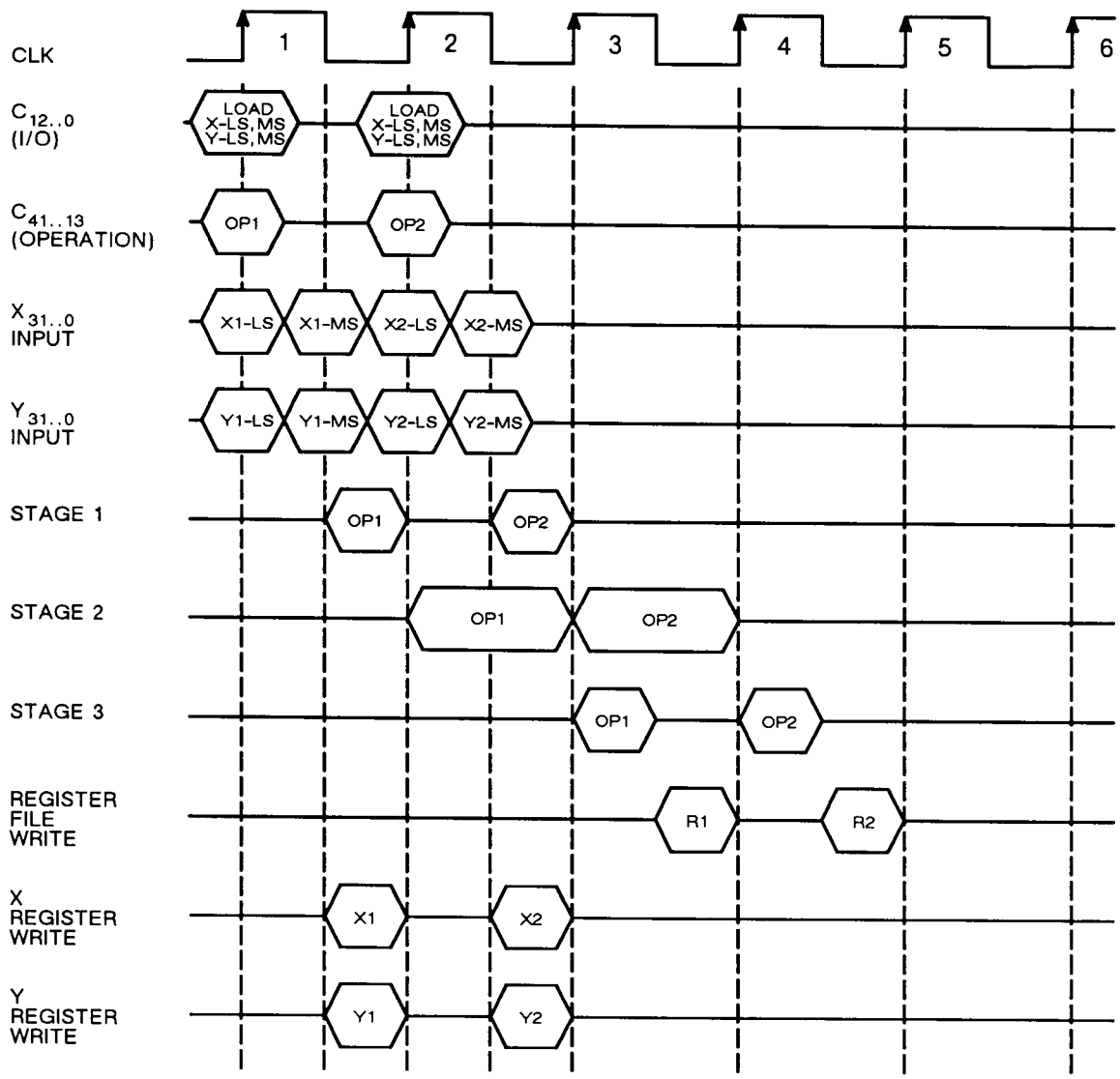


Figure 20. Single-pump delayed load—Configuration C (3164 mode)

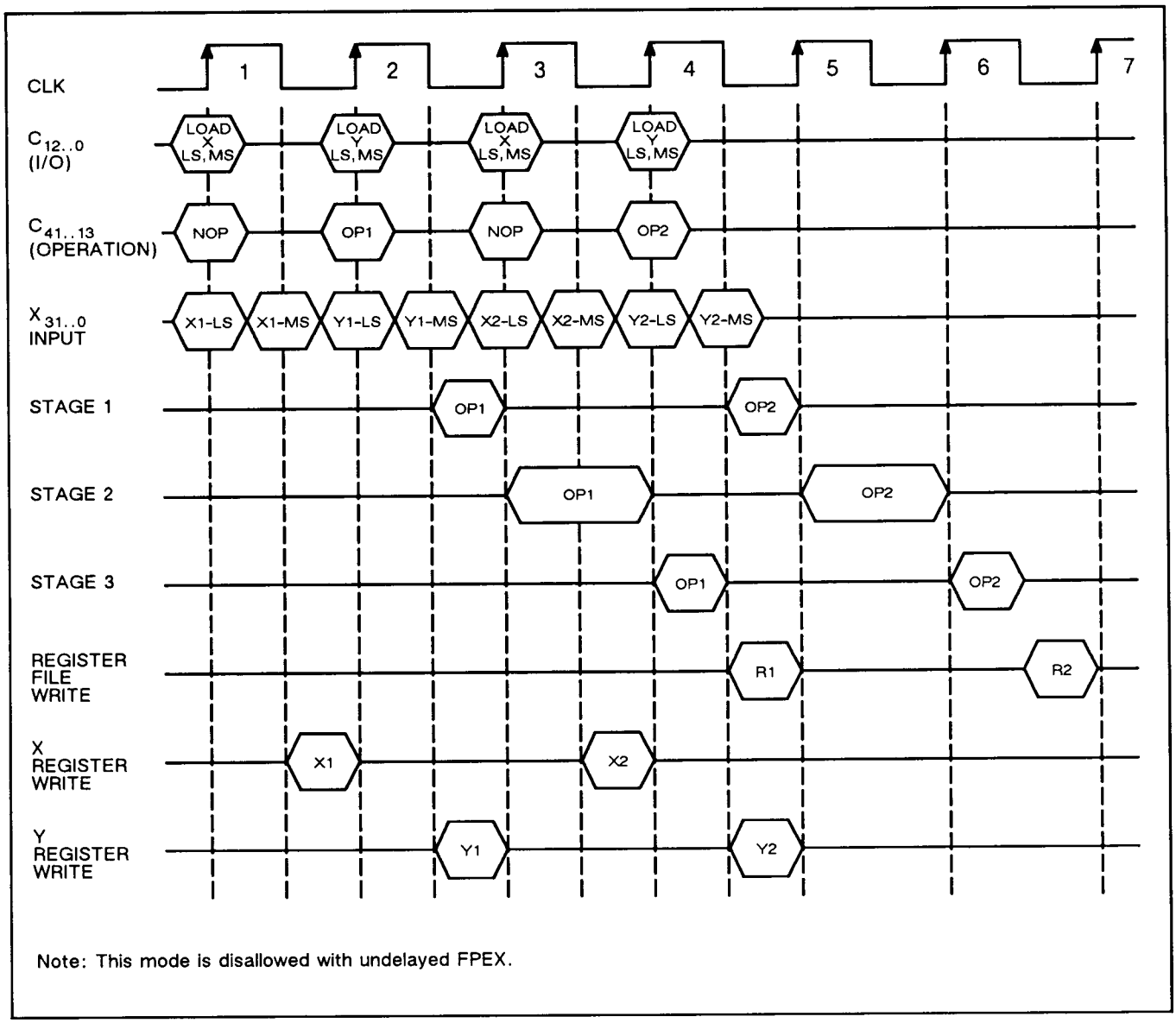
5.5. Load/Store Operations and Their Control, continued



Note: This mode is disallowed with undelayed FPEX.

Figure 21. Double-pump undelayed load—Configuration A

5.5. Load/Store Operations and Their Control, continued



Note: This mode is disallowed with undelayed FPEX.

Figure 22. Double-pump undelayed load—Configuration C

5.5. Load/Store Operations and Their Control, continued

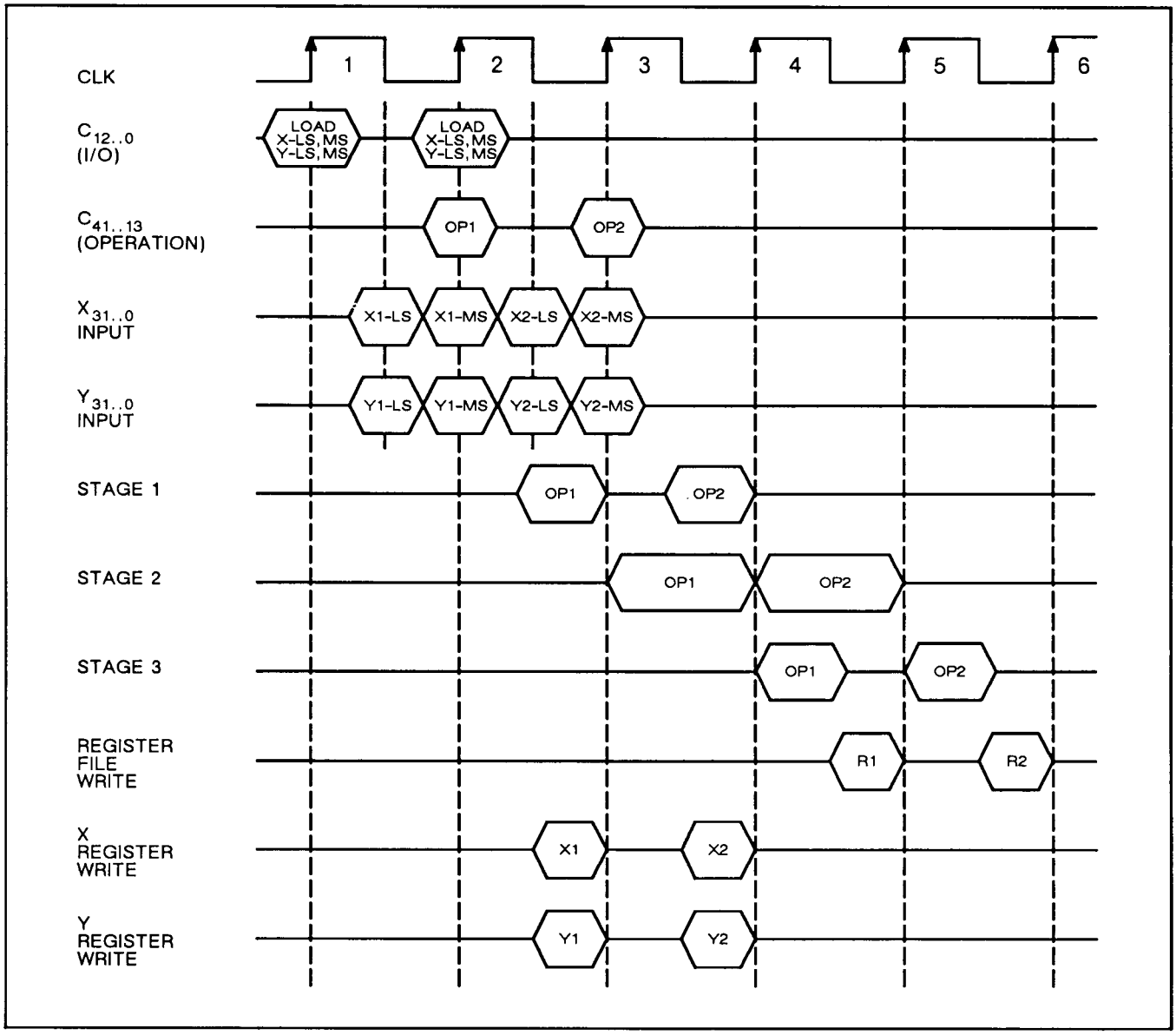


Figure 23. Double-pump delayed load—Configuration A

5.5. Load/Store Operations and Their Control, continued

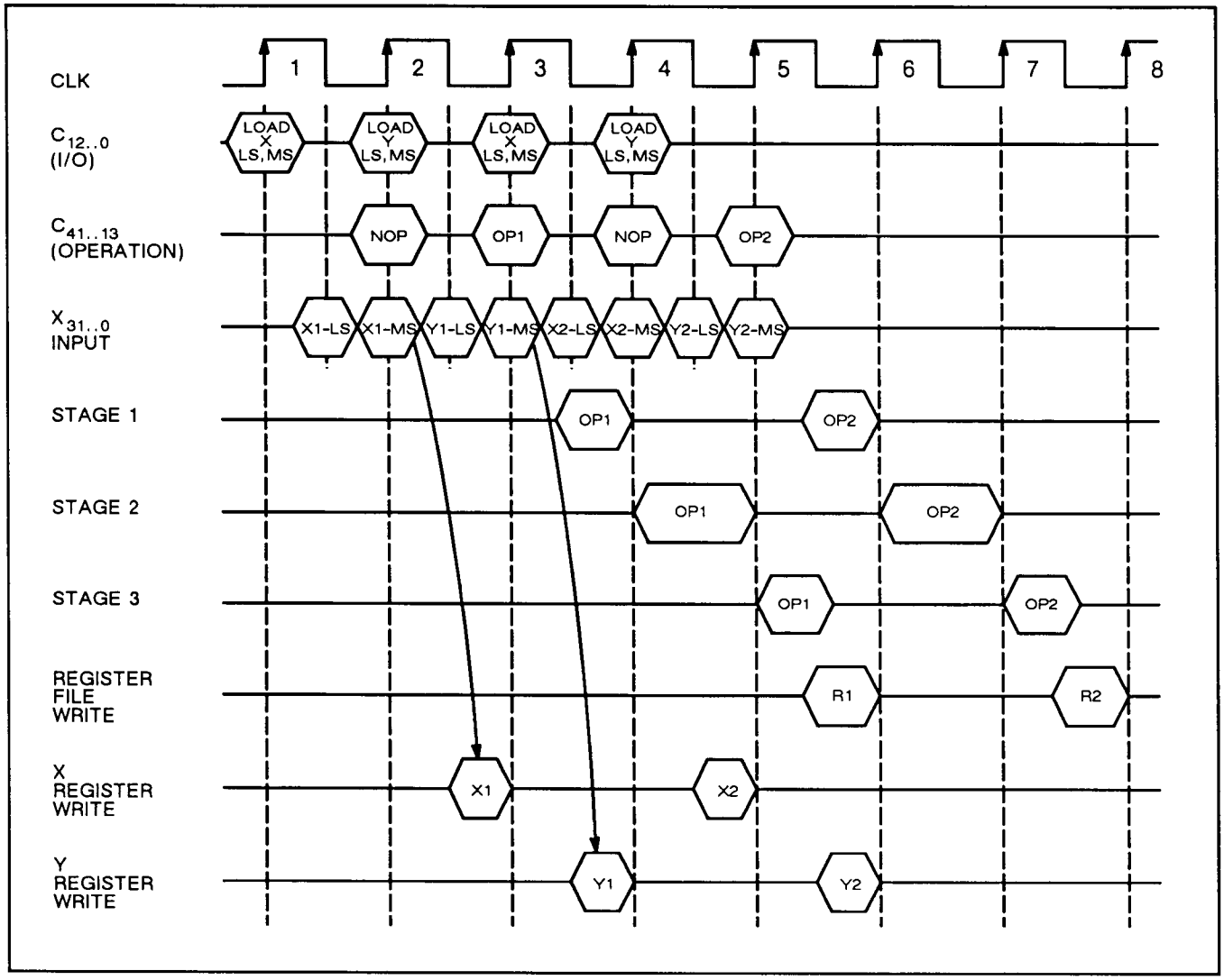


Figure 24. Double-pump delayed load—Configuration C

5.5. Load/Store Operations and Their Control, continued

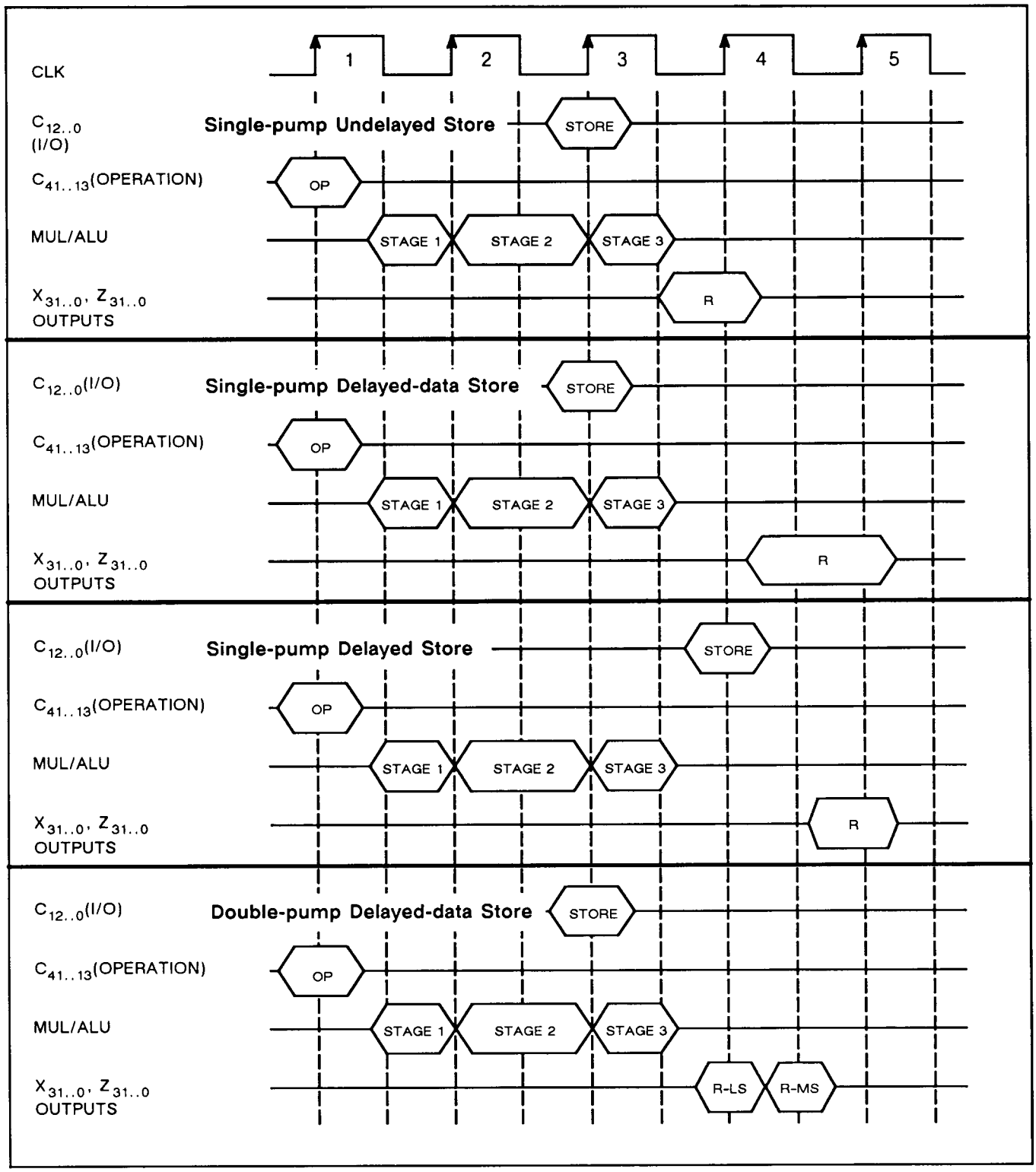


Figure 25. Store timing: conceptual diagram for comparison of store modes

5.5. Load/Store Operations and Their Control, continued

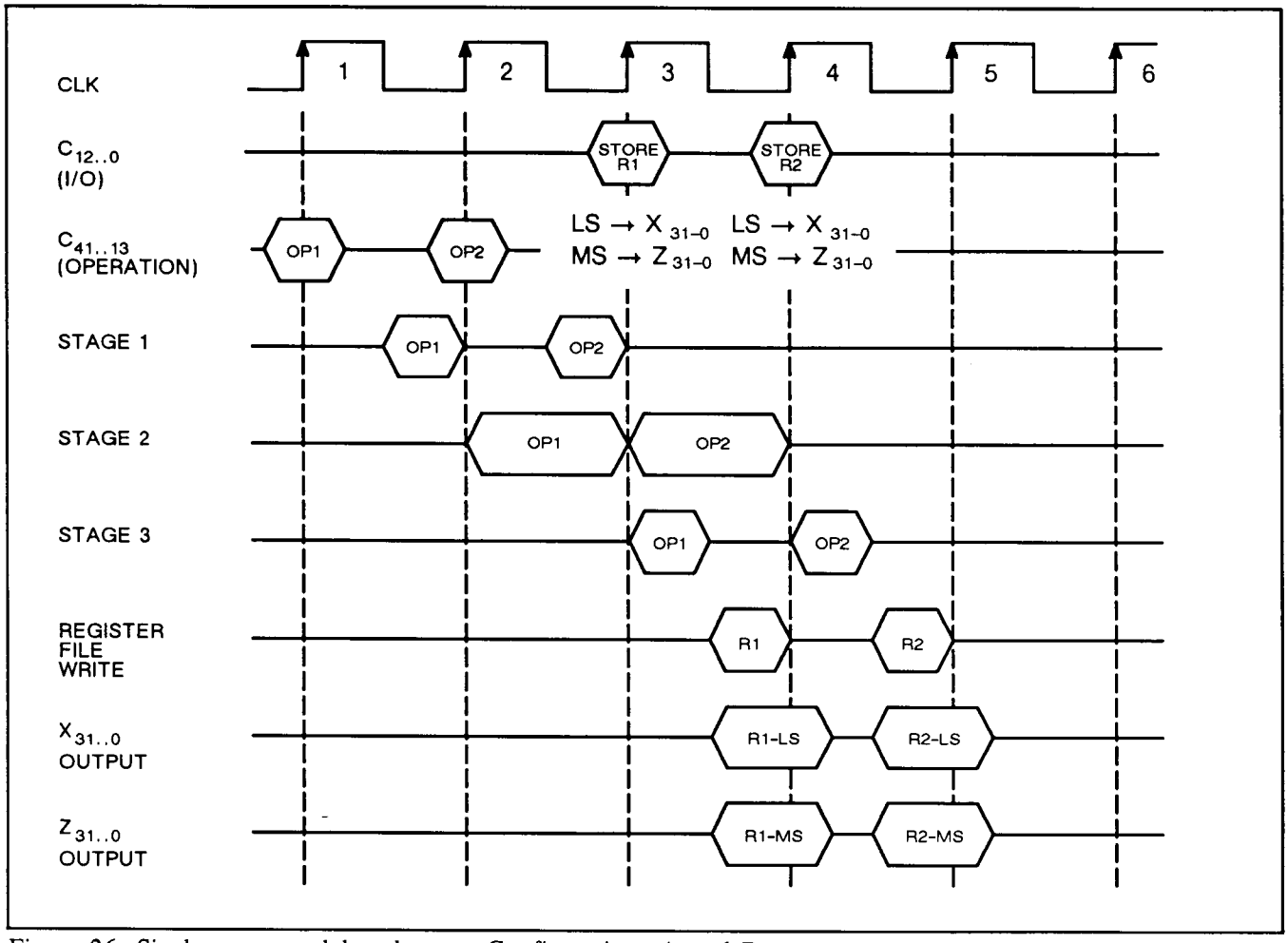


Figure 26. Single-pump undelayed store—Configurations A and B

5.5. Load/Store Operations and Their Control, continued

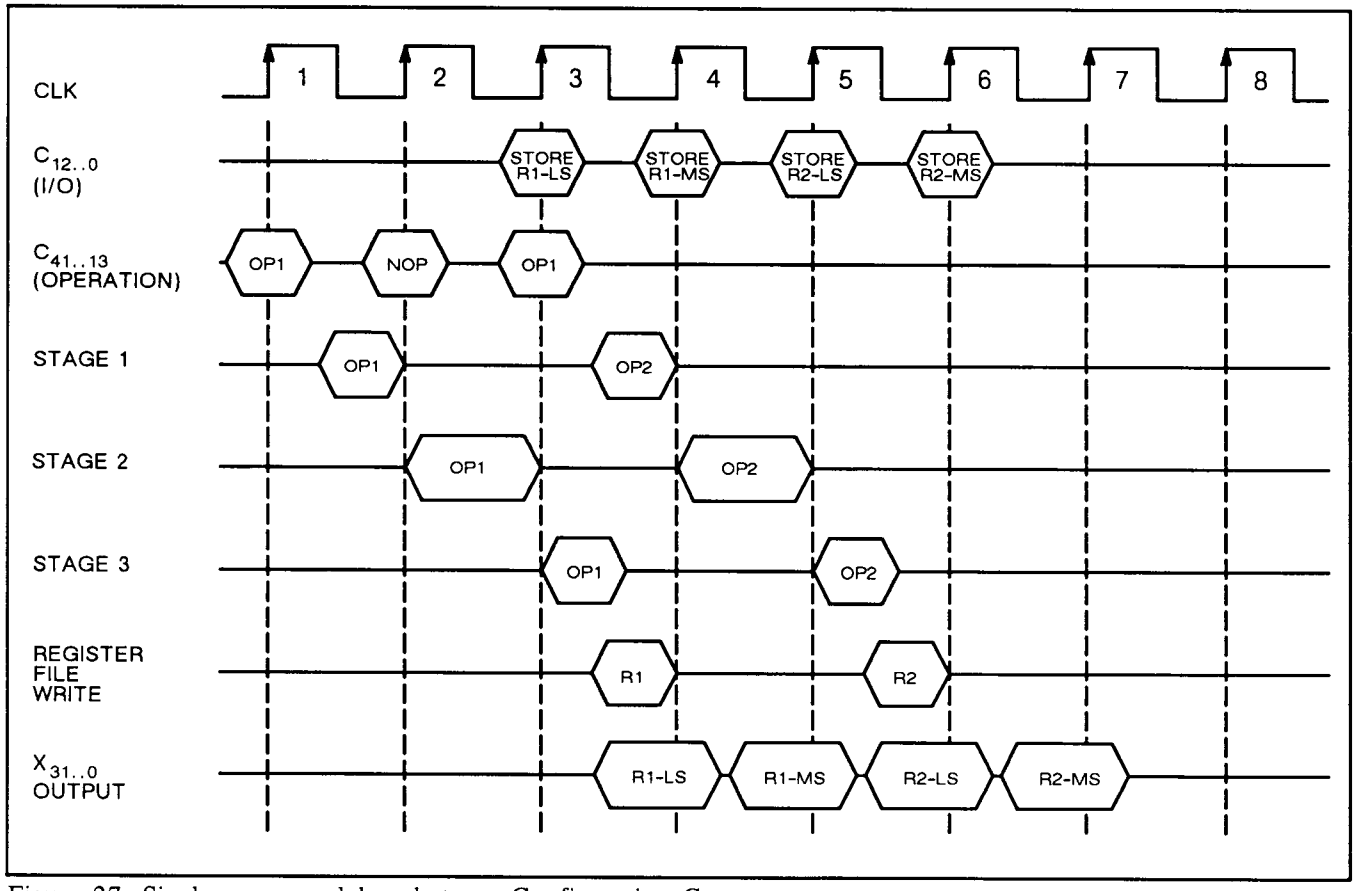


Figure 27. Single-pump undelayed store—Configuration C

5.5. Load/Store Operations and Their Control, continued

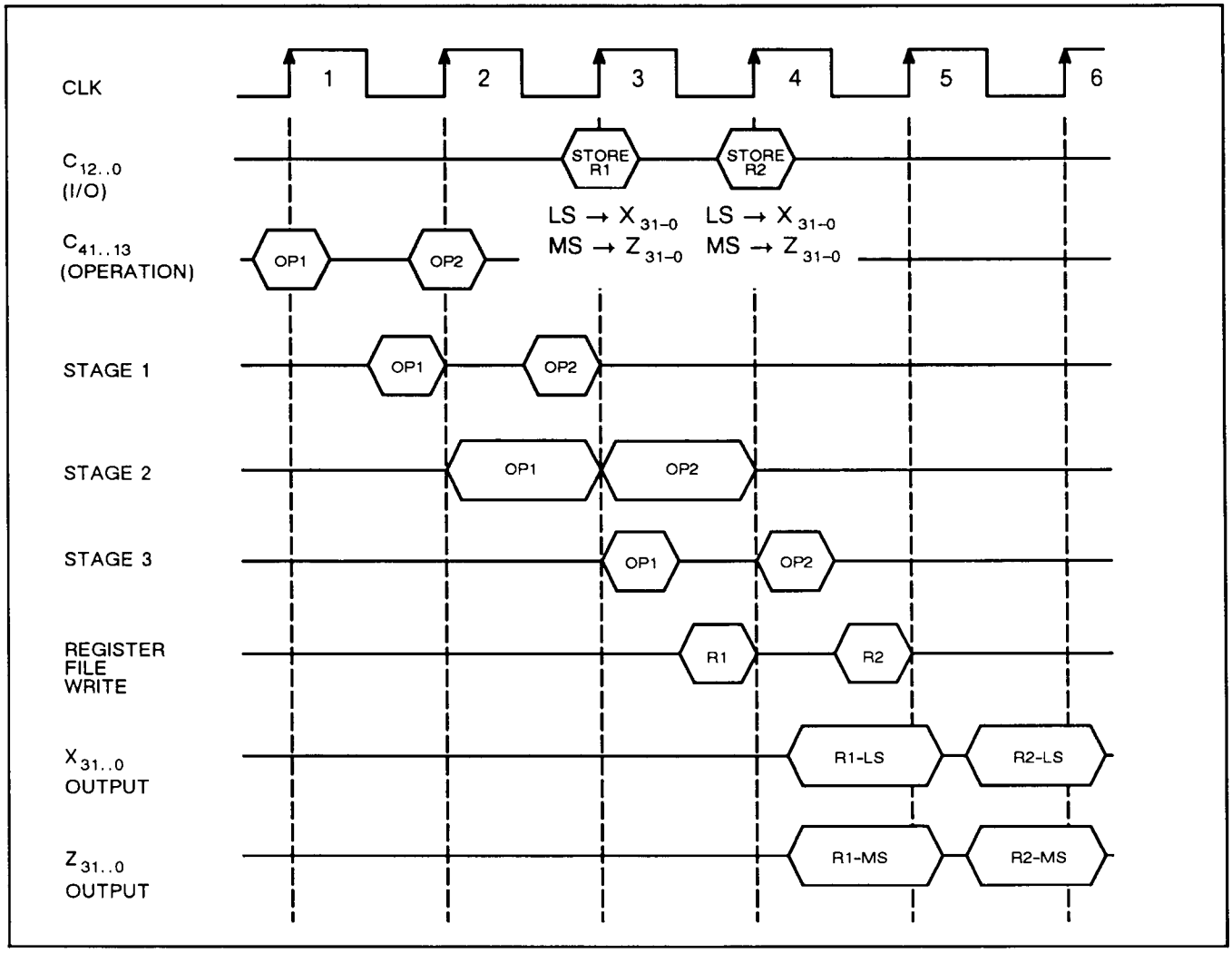


Figure 28. Single-pump delayed-data store—Configurations A and B

5.5. Load/Store Operations and Their Control, continued

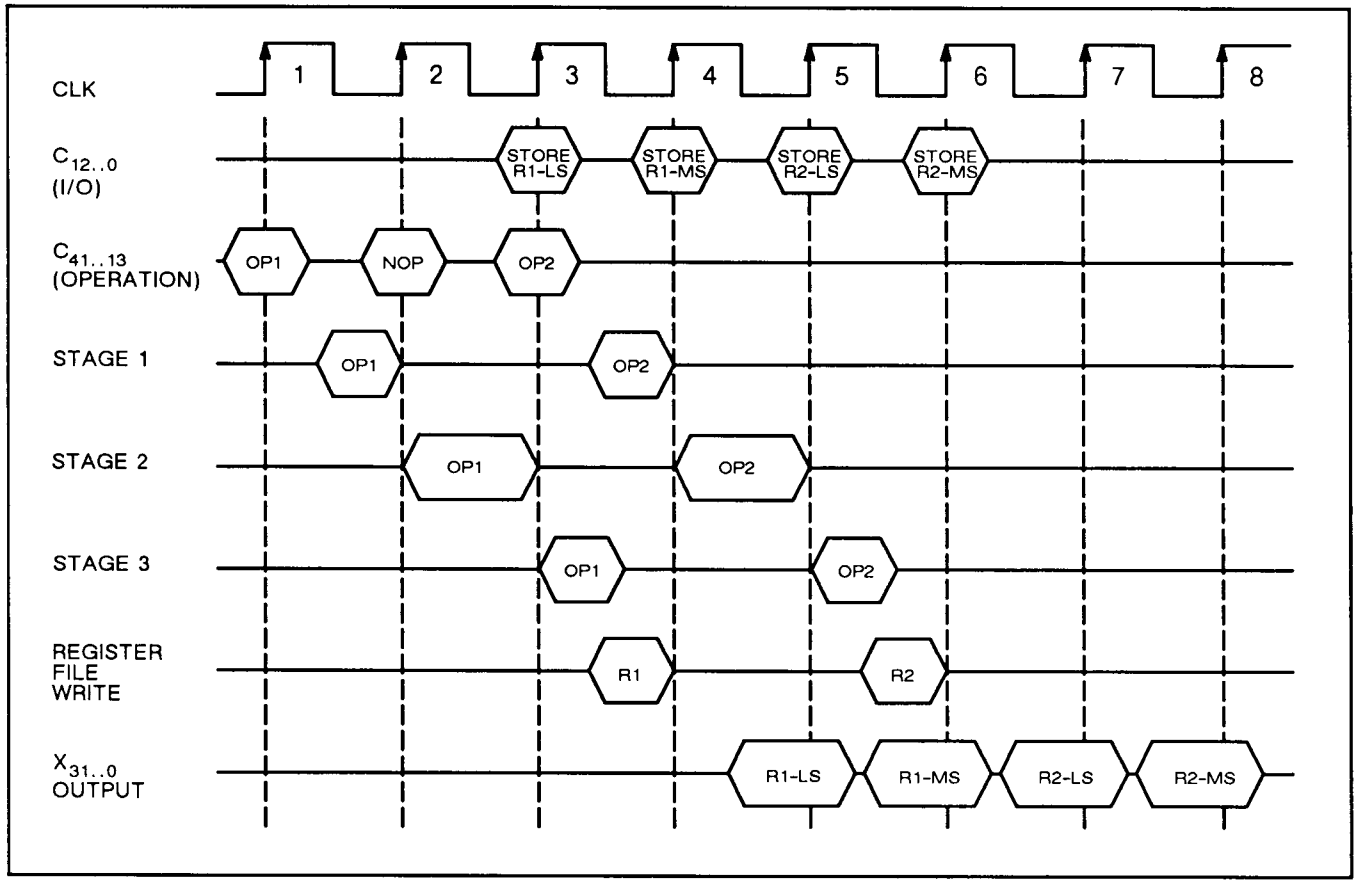


Figure 29. Single-pump delayed-data store—Configuration C

5.5. Load/Store Operations and Their Control, continued

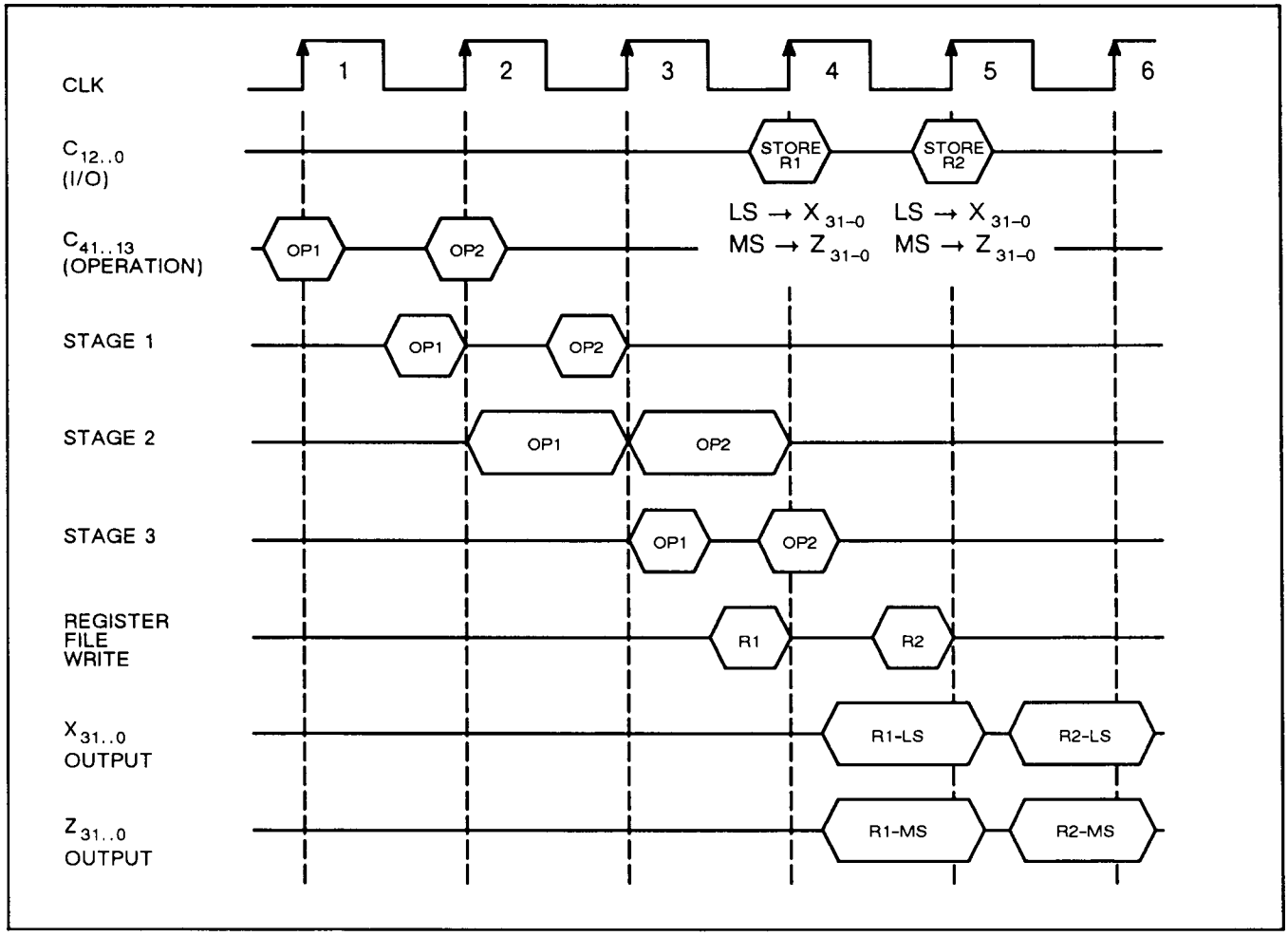


Figure 30. Single-pump delayed store—Configurations A and B

5.5. Load/Store Operations and Their Control, continued

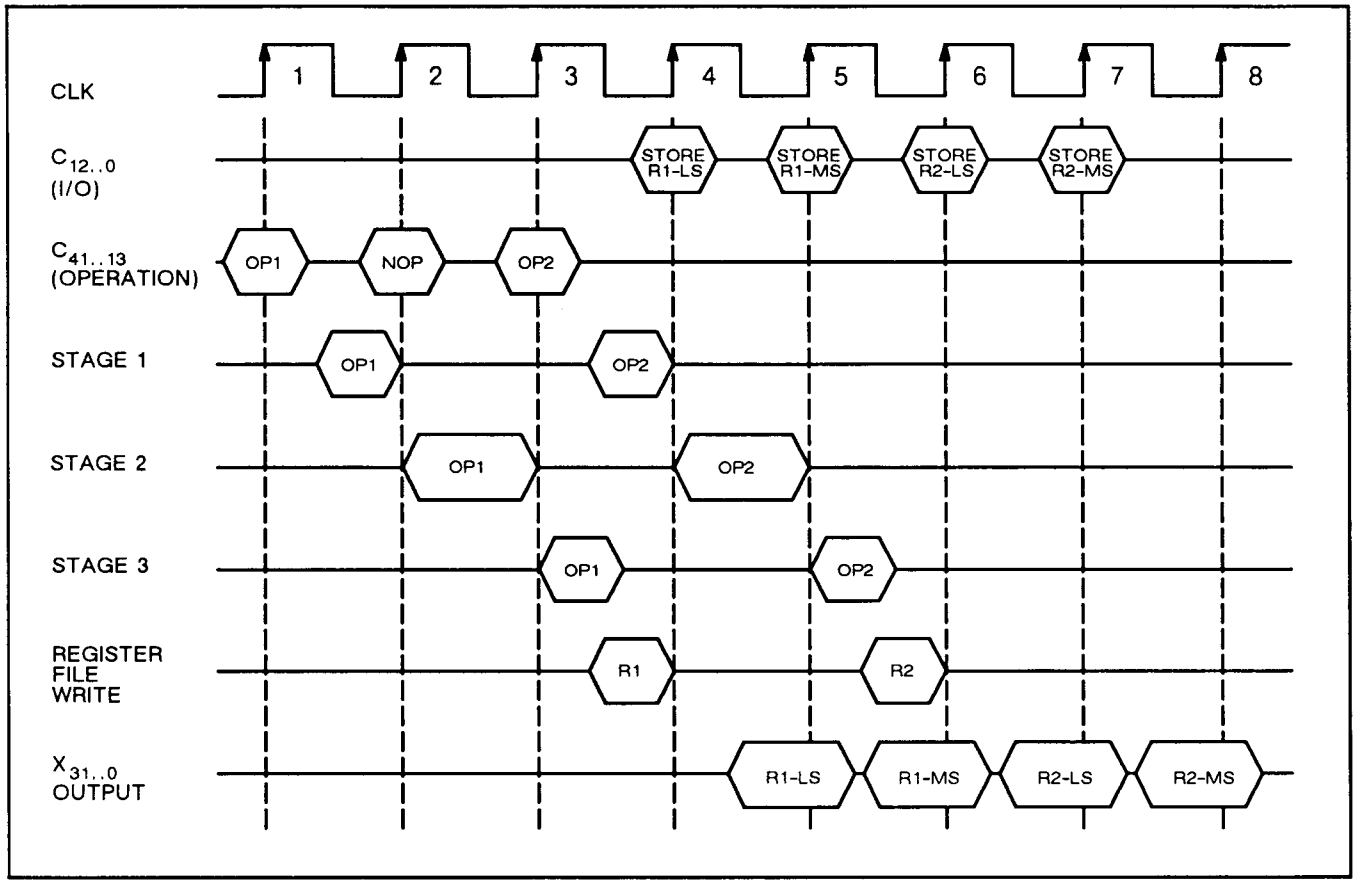


Figure 31. Single-pump delayed store—Configuration C

5.5. Load/Store Operations and Their Control, continued

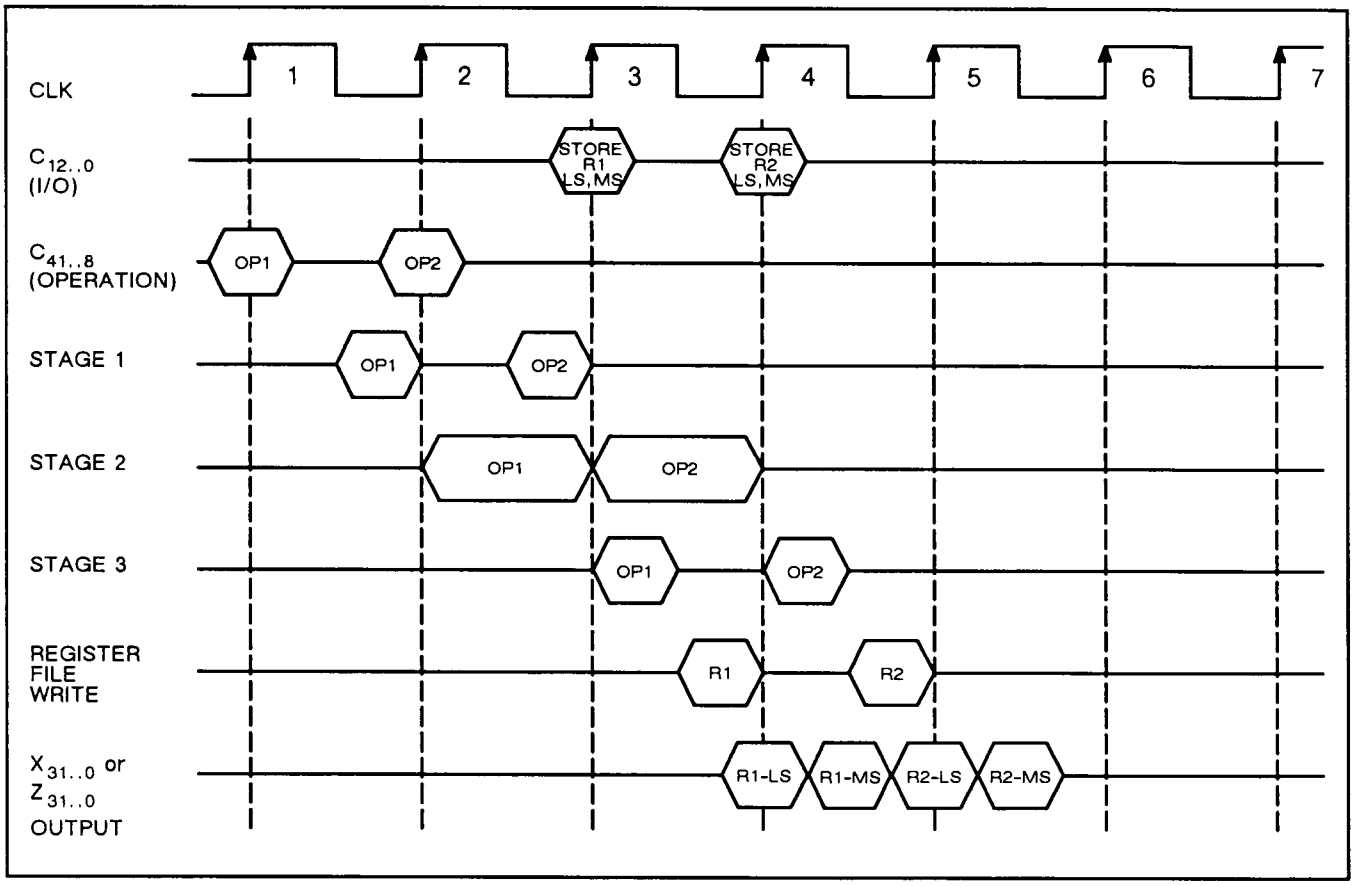


Figure 32. Double-pump delayed-data store—Configurations A and C

5.5. Load/Store Operations and Their Control, continued

5.5.5. ALLOWABLE LOAD/STORE COMBINATIONS

Some load/store combinations have a timing conflict. One example of such a conflict is shown in figure 33.

From the timing standpoint, a store can follow any load, but not necessarily vice versa. Just because a load followed by a store may not have a timing conflict, however, does not necessarily mean that such code may be interruptible. Interruptibility is discussed in section 16. Allowable load/store combinations are listed in figure 13.

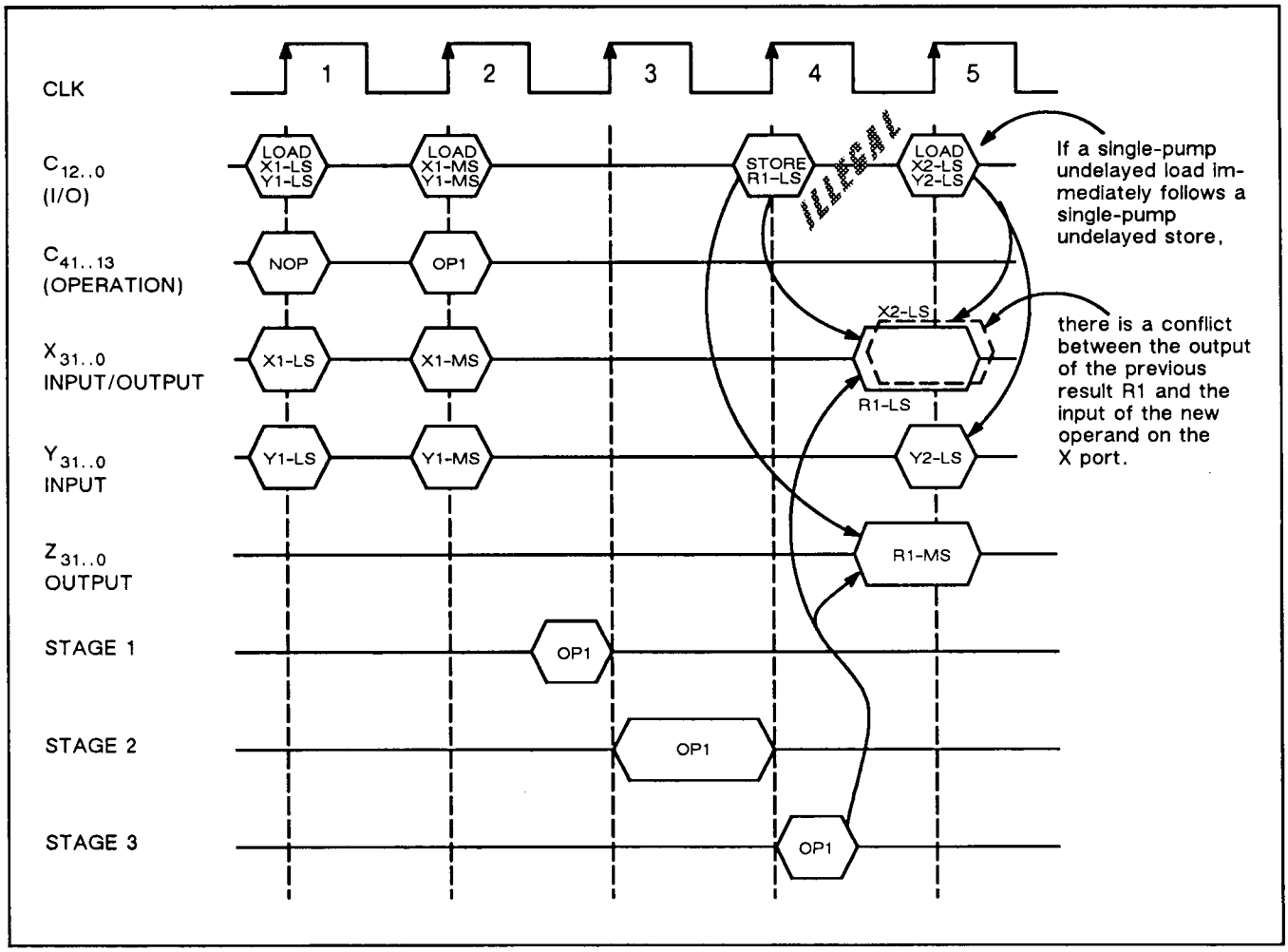


Figure 33. An example of load/store conflict for Configuration A; a single-pump undelayed store followed by a single-pump undelayed load

6. Register File

The 3x64 contains a six-port, 32-word by 64-bit register file. See figure 34.

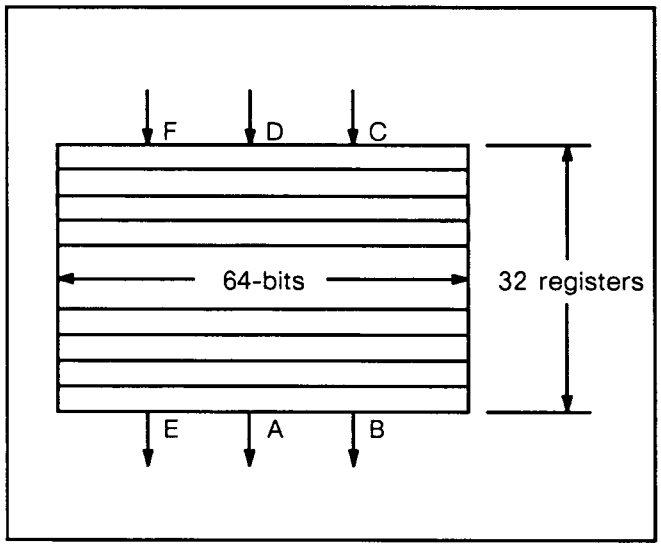


Figure 34. The six-port register file

6.1. Storage Of Data Types

The data types supported in the 3x64 are stored in registers as shown in figure 35 and described below. This applies to X and Y registers as well as to the register file registers. It does not apply to T-latches (see section 10).

Double-precision floating-point and 64-bit logical data occupy the entire 64-bit width of a register. In single-pump mode, the two halves of a 64-bit floating-point or logical quantity can be loaded into the most-significant and least-significant halves of the register in any order.

Single-precision floating-point data is stored in the upper 32 bits (bits 63..32) of a register. The unused bits in the lower half of the register (bits 31..0) remain unchanged. (However, if a single-precision operation is performed, and the result is also specified to be single-precision, then when this result is written into the upper half of a register, bits 31..0 of this register are set to zero.) Any one register may contain only one single-precision floating-point number; that is, when the upper half of a register is used to store one single-precision floating-point number, the lower half may not be used to store another.

Integer data is stored in bits 52..21 of a register. The unused bits (63..53 and 20..0) are automatically set to zero by the load instruction as well as by any integer instruction. Any one register may contain only one 32-bit integer number.

Note that when writing to, or reading from, the register file, the load/store controls XCNT, YCNT, and ZCNT of the code word require the user to differentiate integers from other data types. Thus the user (or compiler) must know the data type when the data is being loaded, and must keep track of which register stores which data type so that the correct one can be specified when storing. For more information on loading and storing of various data types, see section 5.5.

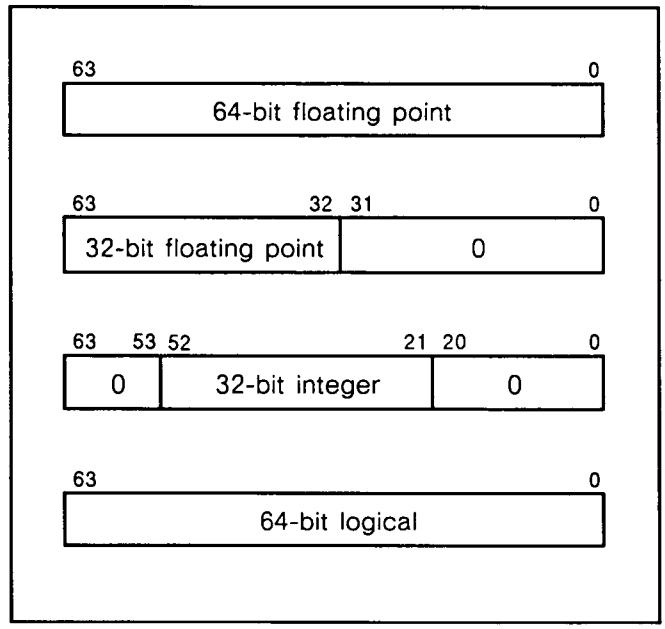


Figure 35. Storage of data types

6.2. Register File Ports and Internal Buses

The register file has six ports: three read-only ports, A, B, and E; and three write-only ports, C, D, and F. See figures 1–3. Each port can transfer a 64-bit data word on every cycle.

The A and B ports can be used to supply operands to the ALU, the multiplier, and the divide/square root unit, from two internal 64-bit buses: the A bus and the B bus, respectively.

The C port receives the result of the previous ALU operation, via the internal 64-bit C bus; the D port receives the result of the previous multiplier or divide/square root unit operation, via the internal 64-bit D bus.

6.2. Register File Ports and Internal Buses, continued

The E port is used to store the contents of a register in the file through the external X and/or Z ports, via the internal 64-bit E bus. The F port is used to load a register in the file from the external X port, and, when the 3364 is used in configuration B (single 64-bit I/O bus), also from the Y port. The F port loads occur via the internal 64-bit F bus. The E port may be bypassed on stores, and the F port on loads; see section 6.5.

This organization allows calculation to proceed in parallel with I/O transfers, maximizing system performance.

Port	Address	Code Word Bits
A port	AADD	$C_{32..28}$
B port	BADD	$C_{27..23}$
C port	CADD	$C_{22..18}$
D port	DADD	$C_{17..13}$
E port	EFADD	$C_{12..8}$
F port	EFADD	$C_{12..8}$

Figure 36. Ports on the register file

6.3. Register File Addressing

The A and B ports supply operands to arithmetic units (ALU, multiplier, and divide/square root unit). The C and D ports receive the results from the arithmetic units. Each port has a separate and independent 5-bit address field in the code word. This allows two operations to be initiated, and two results from previous operations to be written into different registers in the register file, on every cycle.

The E port address and the F port address share the same 5-bit field in the code word. For this reason, either a load *or* a store may be specified on any given cycle.

It is possible to perform both a load and a store on the same cycle: since a register file read is performed during the first half of a cycle, and a write during the second half, the "old" contents of a register may be stored before it is replaced by the "new" value upon a load. An example is described in figure 37. In *a*, the $C_{12..0}$ pins of the code input specify a single-pump delayed load, followed by a single-pump delayed store. Even though the load instruction precedes the store instruction, the register file read of the store instruction happens before the write of the load instruction. This is example of uninterruptible code. The example in *b* is interruptible, because both load and store are specified in the same instruction.

6.3. Register File Addressing, continued

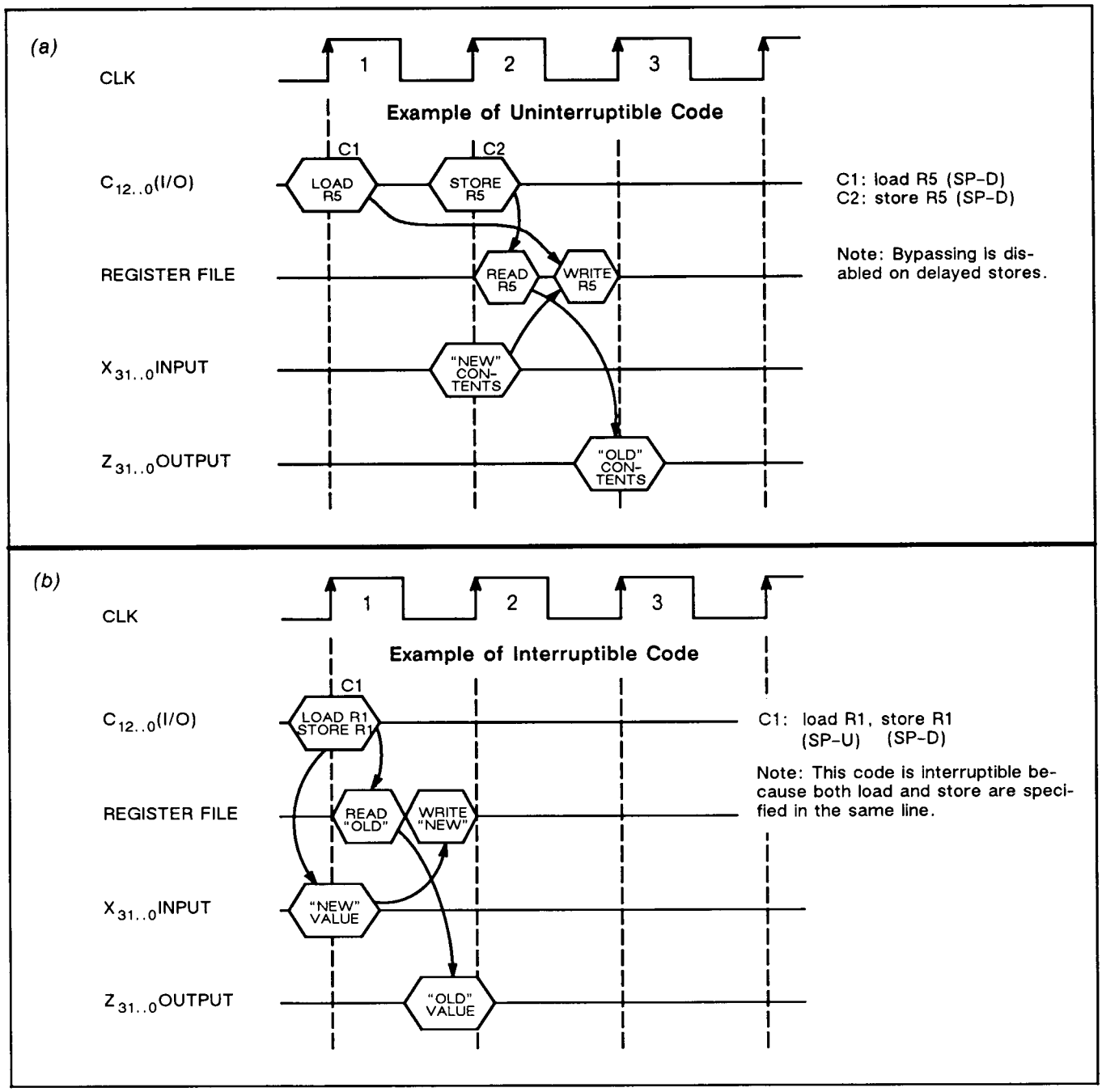


Figure 37. Reading and writing the same register in the same cycle

6.4. Register File Write Priority

It is possible that up to three different sources specify the same register write address: the result of a multiplier operation, the result of an ALU operation, and a load. In other words, at least one pair of the CADD, DADD, and EFADD fields point to the same file register. When such a conflict arises, it is arbitrated according to the these priorities:

Register File Input	Priority
Load	Highest
ALU output (C bus)	
Multiplier output (D bus)	Lowest

Figure 38.

EXAMPLE

In general, if register file priority must be invoked to resolve conflicts, the code that causes these conflicts is usually "bad" code.

Instruction	Operation
C1	R1 OP1 R2 → R4
C2	LOAD R4 (Delayed load)
C3	R4 OP3 R5 → R0

Figure 39.

Refer to figures 39 and 40. If OP1 is a 32×32 floating-point multiply, then the result of the multiply operation will attempt a write to R4 at the same time that the delayed Load R4 will try to write to the same register. In this case, the load will take priority. Then the third operation OP3 will use the data just loaded as one of its operands, rather than the result of the previous multiply operation.

Section 16 deals with illegal code sequences and interruptibility in more detail.

6.4. Register File Write Priority, continued

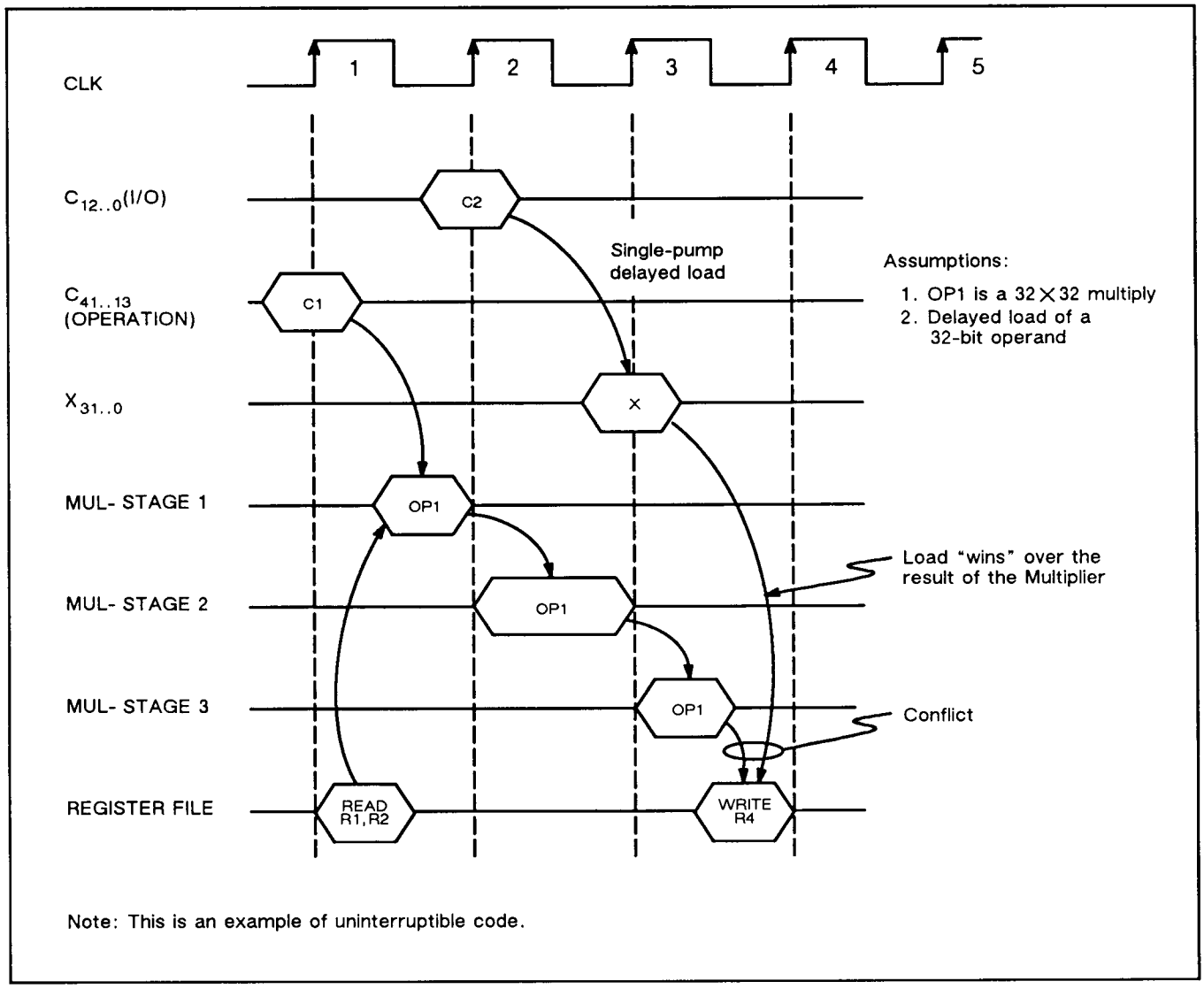


Figure 40. Resolution of register file write conflicts

6.5. Register File Bypassing

Bypassing logic is enabled if the Bypass-on bit in the status register (SR1$_6$) is set. The X and Y registers are always bypassed on loads, regardless of the state of SR1$_6$.

The purpose of bypassing is to reduce latency, by one cycle, in the following three cases:

- On stores, to output the result of an operation simultaneously with writing it to the register file
- On loads, to provide the data being loaded as an operand simultaneously with writing it to the register file
- Register-to-register operations

We will consider each of the three cases in turn.

6.5.1. BYPASSING ON STORE OPERATIONS

Except for delayed store mode, bypassing on stores allows the result of an operation to be output through an external port simultaneously with being written to the register file. If the EFADD field of a store operation is equal to the CADD (or DADD) field of an arithmetic operation whose result is placed on the C bus (or D bus) on the same cycle when the store operation is active, then this result is bypassed to the E bus for output, while at the same time this result is written to the register file through the C port (or D port). Note that the result is always written to the register file. Without bypassing, the result would have to be written to the register file and then read out by a subsequent instruction. Bypassing reduces latency by a cycle. In delayed store mode, bypassing on stores is automatically disabled, and stores occur from the register file.

Figure 41 provides an example of bypassing on store. Consider this sequence of instructions:

C1 R1 × R2 → R3
C2 R5 OP R6 → R7
C3 Store R3

This store can be single-pump undelayed or single- or double-pump delayed-data; in this example we assume single-pump undelayed store.

Bypassing is activated when the register file address specified by the EFADD field of the store instruction is the same as the DADD specified for the result of the just-completed multiply operation. Since these two addresses are the same, the result of the operation is written into a register and simultaneously output through an external port.

If there is a conflict, that is, CADD = DADD = EFADD, it is resolved according to the same priorities as register file writes (see section 6.4). This is true for all three cases of bypassing.

6.5.2. BYPASSING ON LOAD OPERATIONS

Bypassing on loads allows an input through an external port to be used as an operand simultaneously with being written to the register file. If the EFADD field of a load operation is equal to the AADD (or BADD) field of an arithmetic operation which is being activated on the same cycle with the data being loaded, then this input data is bypassed to the A bus (or B bus) to be used as an operand, while at the same time this data is written to the register file through the F port. Note that the load always writes to the register file. Without bypassing, the input data would have to be first loaded into the register file, and then used as an operand of an arithmetic operation on the following cycle. Bypassing on loads also reduces latency by a cycle. Figure 42 provides an example.

Note that bypassing of X or Y registers on loads cannot be disabled regardless of Bypass Enable bit in the status register.

6.5.3. BYPASSING ON REGISTER-TO-REGISTER OPERATIONS

Bypassing is also used in register-to-register operations. When the result of an operation is used as an input to a subsequent operation, bypassing allows the second operation to begin as soon as the result of the first is completed — eliminating a trip through the register file. Note that the register file is still written with the result of the first operation.The logic diagram of bypassing is given in figure 43. Using figure 44 as an example, bypassing is accomplished by specifying the destination address of the result of instruction C1 to be the same as the address of one of the operands of instruction C3.

6.5. Register File Bypassing, continued

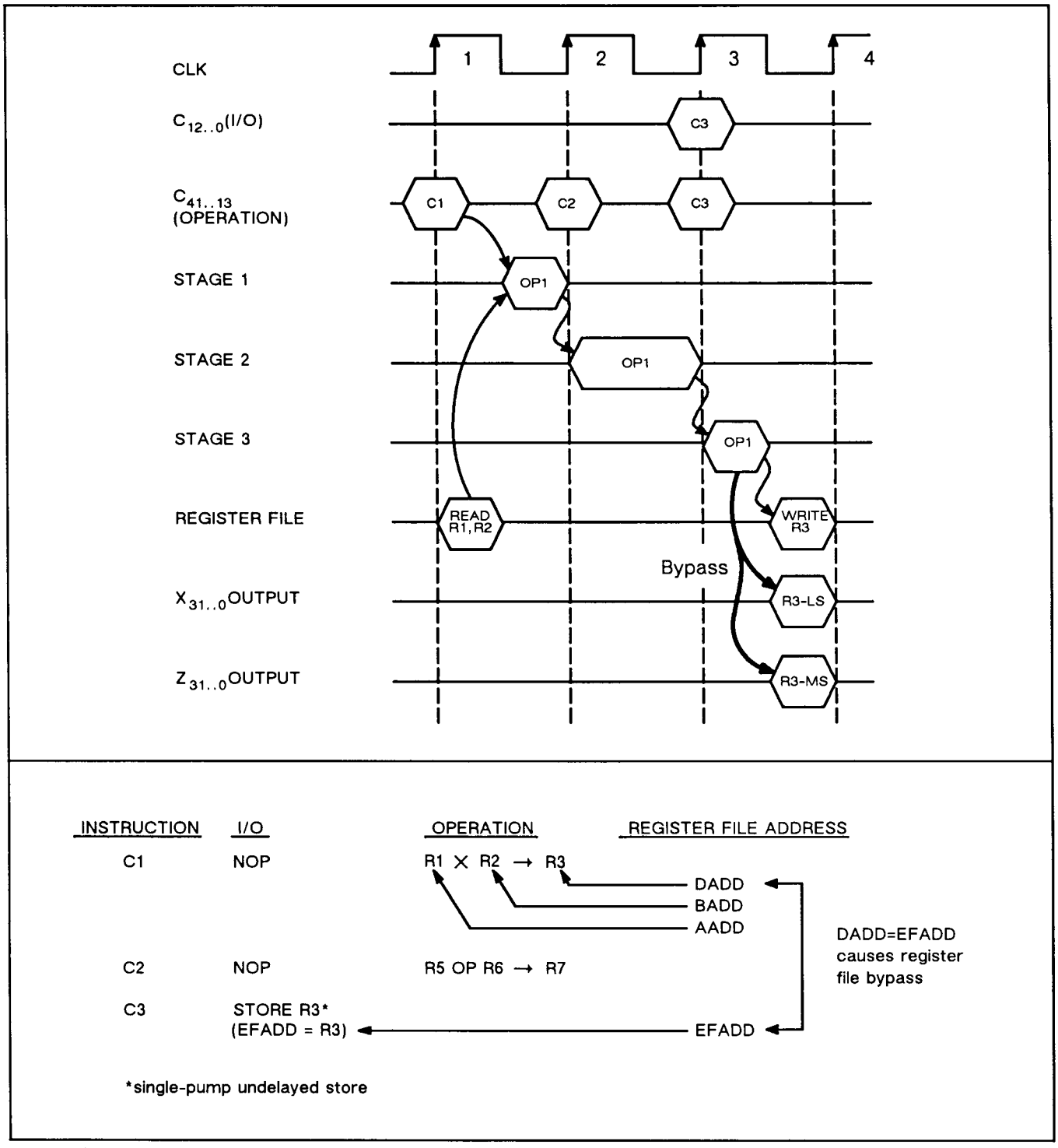


Figure 41. Register file bypassing on store

6.5. Register File Bypassing, continued

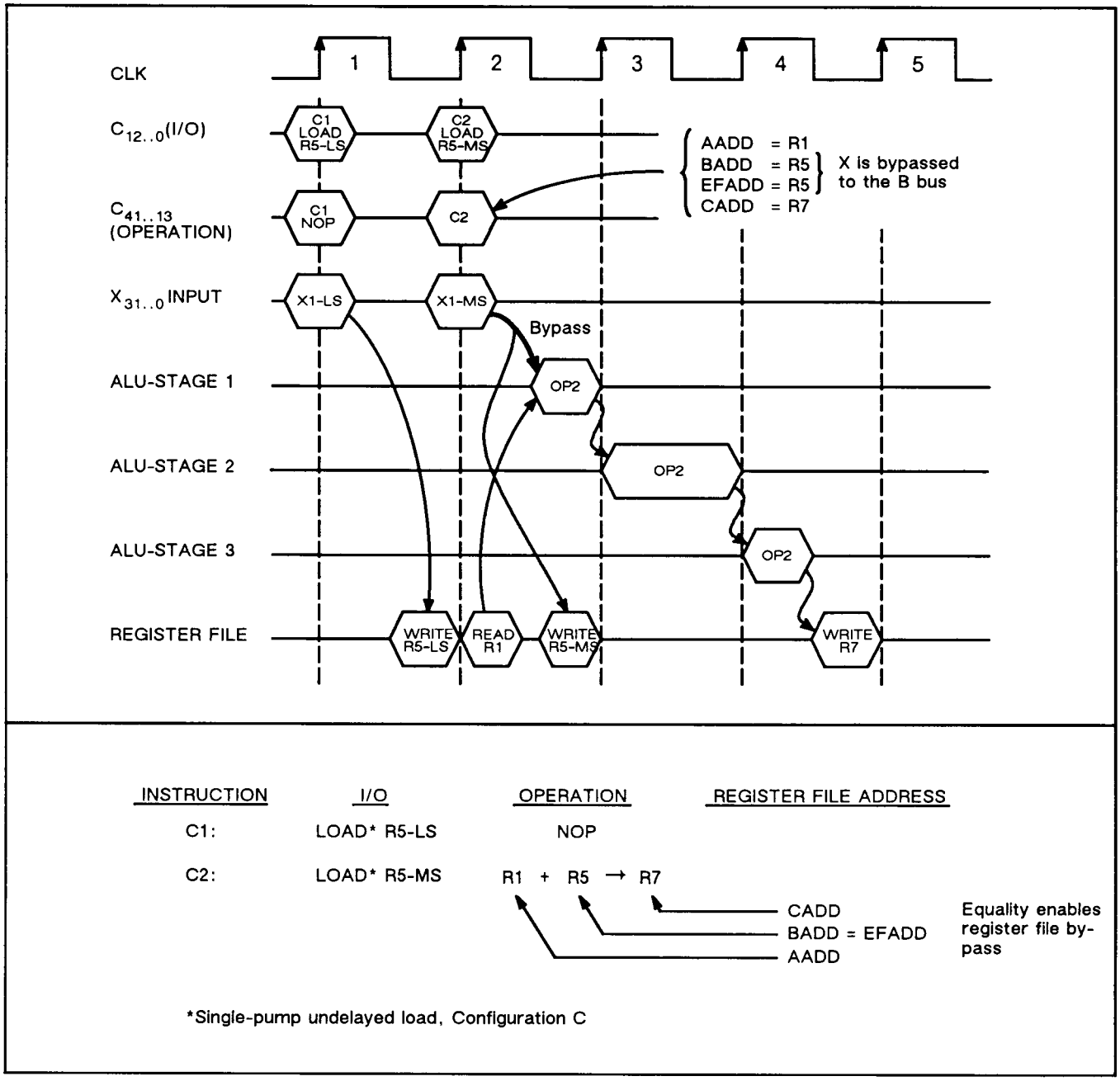


Figure 42. Register file bypassing on load

6.5. Register File Bypassing, continued

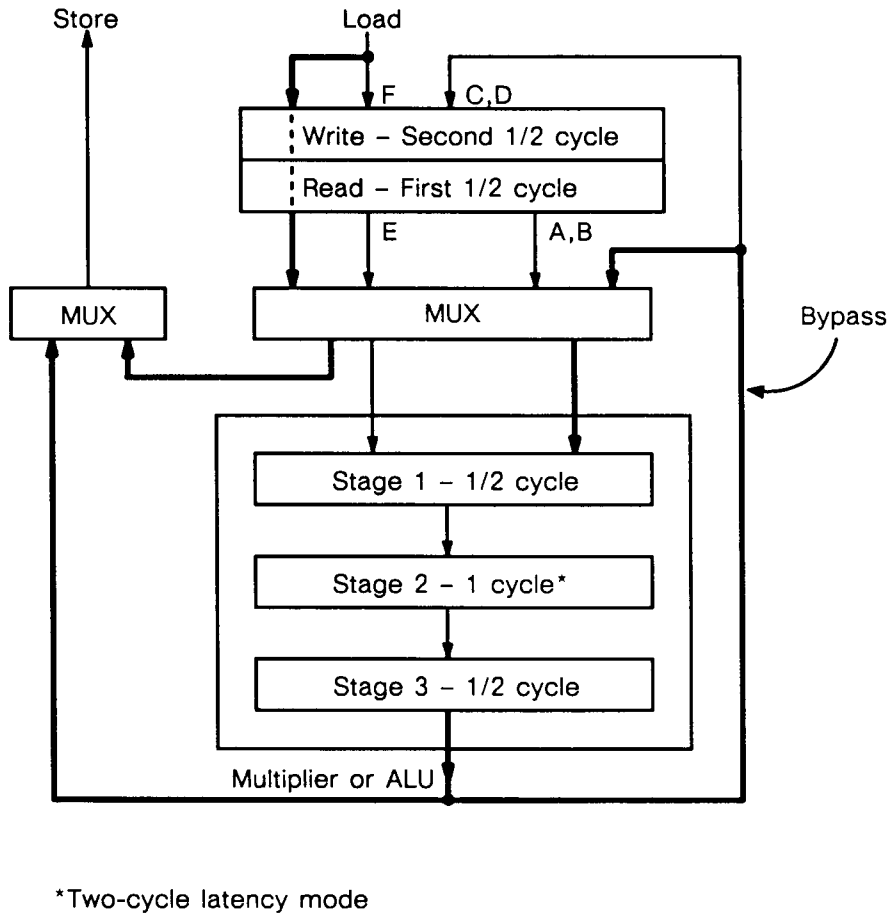


Figure 43. Diagram of bypass logic

6.5. Register File Bypassing, continued

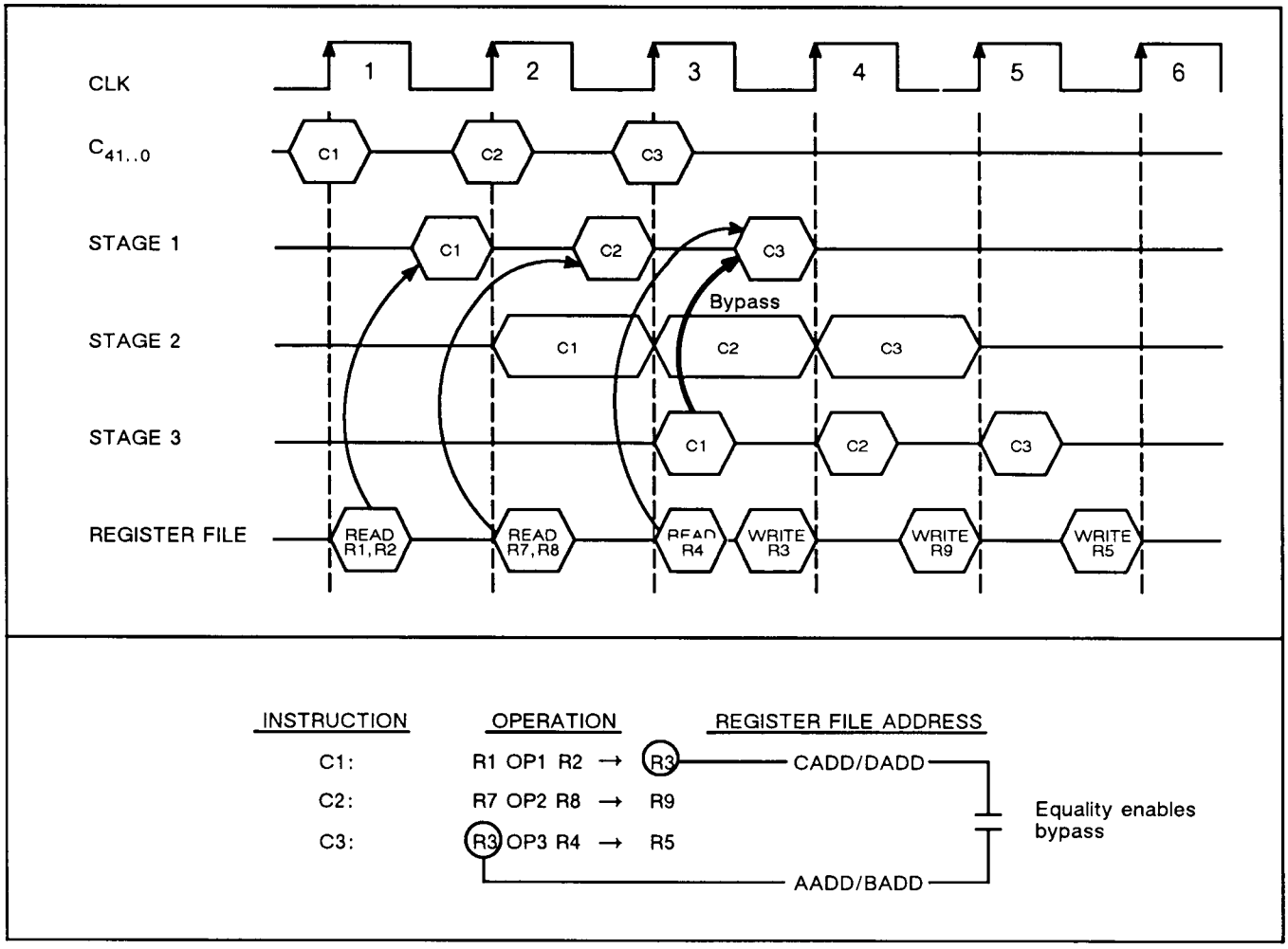


Figure 44. Bypassing on register-to-register operations

7. Multiplier

The 3x64 multiplier is pipelined, with two pipeline registers and three stages. See the block diagram in figures 2 and 3. Every multiply operation passes through these stages in succession, as described in figures 45 and 47. The multiplier may be set to operate in either two-cycle latency mode or three-cycle latency mode. In two-cycle latency mode, every multiplication takes two cycles if bypassing is used and three cycles if it is not. The three-cycle latency mode affects only integer and double-precision floating-point multiplications: they take three cycles with bypassing and four cycles without; other operations execute exactly the same as in the two-cycle latency mode. At any given instant, two different multiply operations may be executing in the multiplier.

7.1. Multiplier Stages

Stage 1 contains front-end circuits to detect source exceptions, for example, denormalized numbers, NaNs, invalid operations; and an adder to add exponents of floating-point numbers. If an enabled source exception or an invalid operation is detected in stage 1, the floating-point exception output pin, FPEX, will be asserted, to indicate the exception to the rest of the system. The timing of FPEX assertion depends on FPEX modes (delayed/undelayed and sticky/pulsed) and is described in section 15. Stage 1 always operates after the rising edge of the clock but no sooner than the operands are available, usually after the falling edge.

Stage 2 contains a fixed-point half-array multiplier to multiply integers or mantissas of floating-point numbers. Single-precision and mixed-precision floating-point multiplications complete in one pass through the array, while double-precision floating-point and integer multiplications take two passes. These two passes take one or two entire clock cycles, depending on the selected latency mode. In two-cycle latency mode, stage 2 operation takes one cycle, that is, both passes of the integer or double-precision floating-point multiplications complete in one cycle. In three-cycle latency mode, each pass takes a cycle; thus stage 2 operation takes two entire clock cycles. However, since it is necessary to pass through the array only once per cycle, the cycle can be shorter. Multiplier latency modes are discussed in section 7.3.

Stage 3 contains IEEE rounding circuits and result exception circuits. Stage 3 operation always takes place in the first half of a cycle following the completion of stage 2 operation.

Integer add/subtract, logical (except single-bit shift), compare, and min/max operations are performed in the multiplier. Logical operations are always performed on 64-bit words.

7.2. Multiplier Operand Sources and Result Destinations

The multiplier has two input ports and one output.

The selection of input port operands is controlled by the MAIN and MBIN multiplexer controls in the code word. The mnemonics "MAIN" and "MBIN" stand for multiplier A input and multiplier B input, respectively. The sources of multiplier operands are the X and Y registers, and the A and B ports of the register file. The selection of the multiplier operands is specified in section 17.

The output of the third stage of the multiplier is placed on the D bus and:

- is always (except for compare operations) routed to the D port of the register file (the address of the register in the file is given by DADD field of the code word in the cycle in which the instruction was latched);
- can also be forwarded to the internal E bus for output, bypassing the register file on a store operation (this is discussed in the section 6.5);
- in multiply-add operations, is routed to one of the temporary latches T0 or T1 (this is discussed in sections 10 and 17);
- in register-to-register operations, can be forwarded directly to an input of the multiplier or the ALU, bypassing the register file (this is discussed in sections 6.5 and 17).

7.2. Multiplier Operand Sources and Result Destinations, continued

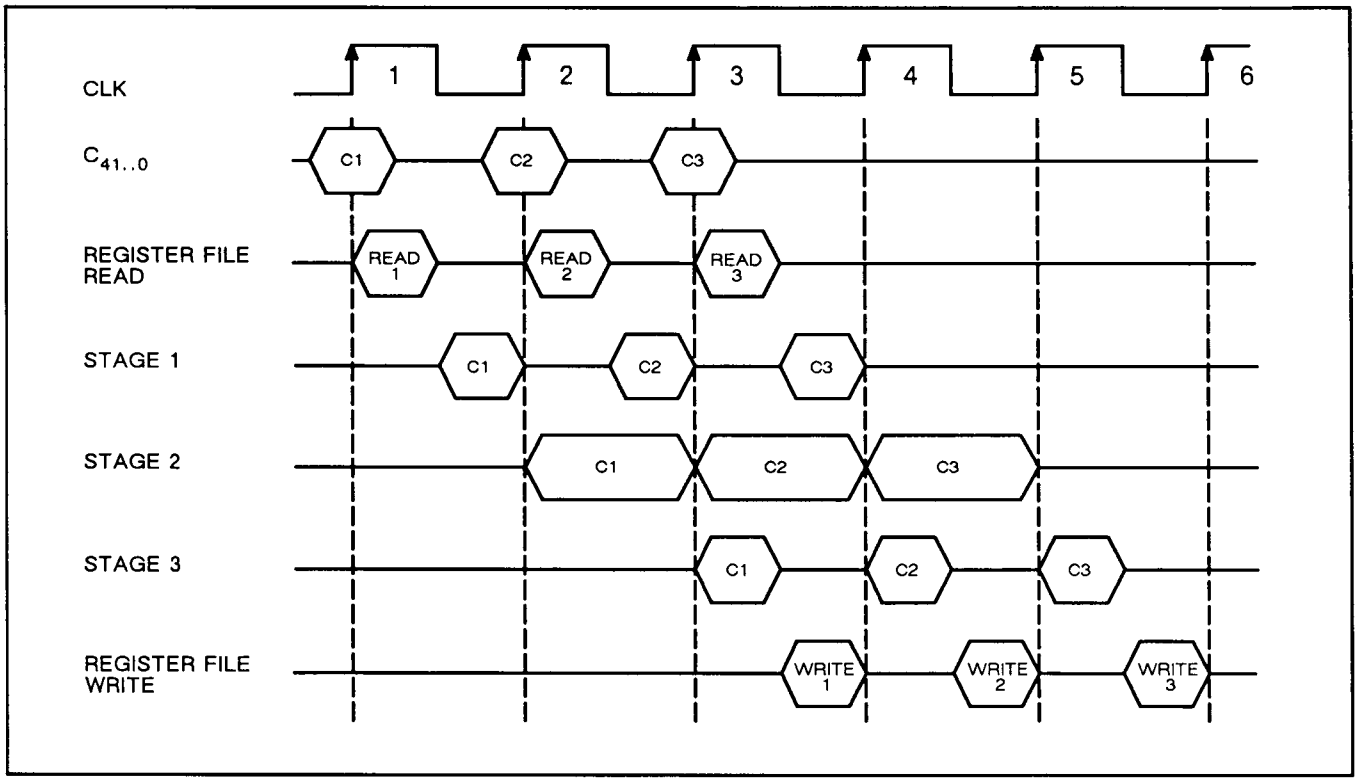


Figure 45. Operation of the ALU and the multiplier in two-cycle latency mode

7.3. Multiplier Latency

The 3x64 provides for two multiply latency modes: *two-cycle latency mode* and *three-cycle latency mode*. The latency mode is controlled by a Multiplier Latency mode bit in the status register, $SR0_7$.

$SR0_7 = 0$	Two-cycle multiply latency mode
1	Three-cycle multiply latency mode

Figure 46.

The purpose of providing the two modes is to allow the user to have faster single-precision floating-point and integer multiplications at the expense of double-precision multiply performance.

It is possible to switch between the two multiplier latency modes by changing the value of the mode bit in $SR0_7$. This is accomplished by loading a new value into SR0. Storing and loading of the status register is described in section 12.2.

Figure 47 provides a timing diagram of a multiply in three-cycle latency mode. Figure 48 details the latency and throughput of the multiplier for integer and single- and double-precision multiplications in both latency modes. The ALU performance is shown in the figure for completeness only. It is not affected by the latency mode.

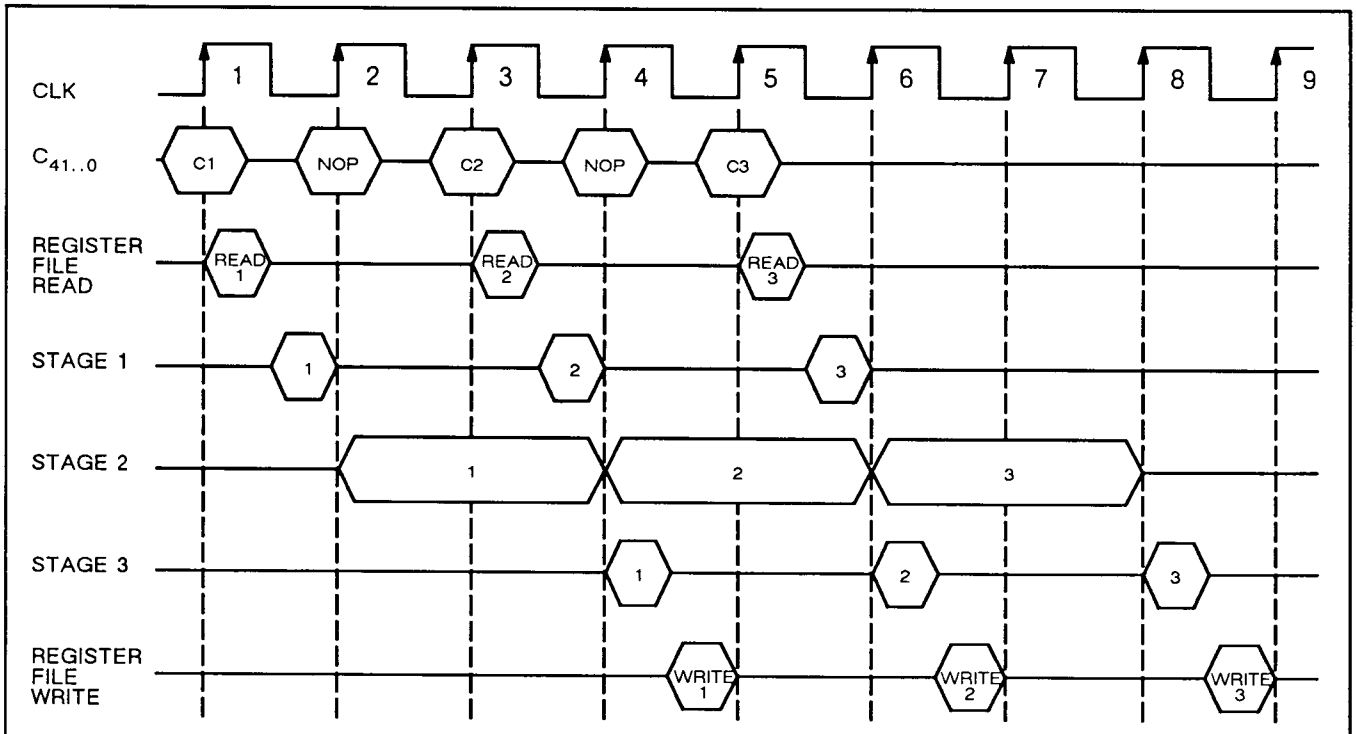


Figure 47. Operation of the multiplier in three-cycle latency mode

7.3. Multiplier Latency, continued

The following are advantages and disadvantages of the two modes.

Advantages of two-cycle latency mode:

- Shorter double-precision multiply latency than in the three-cycle latency mode
- Greater I/O bandwidth because double-pumping may be used
- An integer or a double-precision floating-point multiply instruction can be followed by another multiplier instruction on immediately successive clock cycles
- Multiplier and ALU always have the same latency, two cycles

Disadvantages of two-cycle latency modes:

- Longer cycle time

Advantages of three-cycle latency mode:

- Shorter cycle-time than in the two-cycle latency mode

Disadvantages of three-cycle latency mode:

- Longer double-precision multiply latency
- Double-precision floating-point and integer multiplications have different latencies in the multiplier (three cycles) and ALU (two cycles). Even two different multiplies can have different latencies: integer or double-precision floating-point three cycles and all other operations (except divide and square root) two cycles. An integer or a double-precision floating-point multiply instruction must not be immediately followed by another instruction executed in the multiplier which also places its result on the internal D bus. This restriction is necessary to avoid stage 2 conflict, an example of which is illustrated in figure 49. If this restriction is violated, the second instruction will be ignored.

	Latency Mode	
	Two-cycle	Three-cycle
Multiplier		
Integer and double-precision floating-point multiplications		
Register-to-register latency, cycles	2	3
Throughput, cycles	1	2
All other multiplier operations, including 64×32 multiplications		
Register-to-register latency, cycles	2	2
Throughput, cycles	1	1
ALU		
Latency, cycles	2	2
Throughput, cycles	1	1
I/O Pumping	Single or double	Single only
MAX DIVCLK Frequency	2×CLK	2×CLK

Figure 48. Multiplier latency modes and their effect on performance

7.3. Multiplier Latency, continued

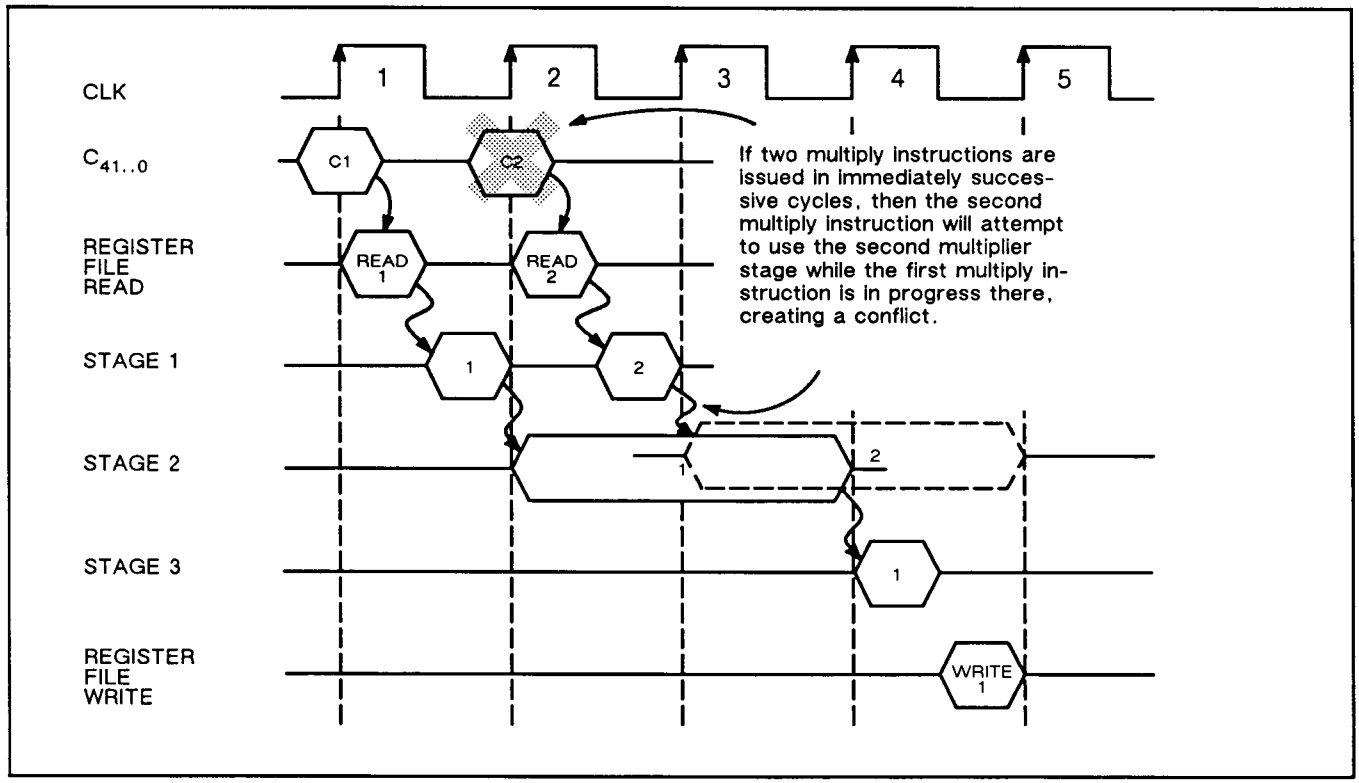


Figure 49. Why, in three-cycle latency mode, two integer or double-precision floating-point multiply instructions may not follow each other

8. ALU

The 3x64 ALU is a pipelined unit, with two pipeline registers and three stages. See block diagram in figures 2 and 3. Every ALU operation passes through these stages in succession, as described in figure 45. Because the ALU is pipelined, every ALU operation takes two cycles, but a new operation can be initiated, and a result can be obtained, on every cycle. At any given instant, two different operations may be executing in the ALU.

From a programmer's standpoint, the operation of the ALU is similar to that of the multiplier in two-cycle latency mode. The latency of ALU operations is always two cycles; its throughput is one cycle.

8.1. ALU Stages

Stage 1 contains front-end circuits to detect source exceptions, such as denormalized numbers and NaNs, as well as invalid operations. If an enabled source exception or an invalid operation is detected in stage 1, floating-point exception output pin, FPEX, will be asserted. The timing of FPEX assertion depends on FPEX modes (delayed/undelayed and sticky/pulsed) and is described in section 15. Stage 1 always operates after the rising edge of the clock but no sooner than the operands are available.

Stage 2 contains a shifter to denormalize the fraction of the smaller of the two operands or to renormalize the result; and an adder. Stage 2 always takes one cycle following the completion of stage 1 operation. This stage performs integer and single- or double-precision floating-point additions and conversions, as well as single-bit shifts.

Stage 3 contains IEEE rounding circuits and result exception circuits. Stage 3 operation always takes place in the first half of a cycle following the completion of stage 2 operation.

8.2. ALU Operand Sources and Result Destinations

The ALU has two input ports and one output.

The selection of input port operands is controlled by single-bit AAIN and ABIN multiplexer controls in the code word. The mnemonics "AAIN" and "ABIN" stand for ALU A input and ALU B input, respectively. The sources of ALU operands are the X and Y registers, the A and B ports of the register file, and temporary latches. The selection of these sources is detailed in section 17.

The output of the third stage of the ALU is placed on the C bus and:

- is always routed to the C port of the register file (the address of the register in the file is given by CADD field of the code word in the cycle in which the instruction was latched);
- can also be forwarded to the internal E bus for output, bypassing the register file on a store operation (see section 6.5);
- in register-to-register operations, forwarded directly to an input of the multiplier or the ALU, bypassing the register file (see section 6.5).

9. Divide/Square Root Unit

The same multiplexers that select multiplier operands also select the operands for the divide/square root (DSR) unit. When a DSR operation is initiated, no other operation may be initiated on the same cycle. The results of a DSR operation are placed on the D bus through the multiplier.

The DSR has a separate clock input, DIVCLK. It may operate at either the same (1×), or double (2×) the frequency of the main clock CLK. In two-cycle latency mode, the maximum DIVCLK frequency is 2× CLK; in three-cycle latency mode, 1× CLK.

Divide operations are performed at the rate of two bits of the result fraction per DIVCLK cycle. Square root operations (SQRT) operate at the rate of one bit of the result fraction per DIVCLK cycle.

When a DSR operation is completed, the DSR automatically waits for the first cycle in which the multiplier is not used, at which time its result is unloaded through the multiplier and stored via the internal D bus. Consider a sequence of events in the DSR unit, for the case of 2× DIVCLK. See figure 50. A DSR operation (divide or SQRT) is clocked in on the rising edge of cycle 1. In the first half of this cycle operand(s) are read from the register file, and in the second half of the cycle they are latched in the DSR unit. The actual DSR loop starts on the rising edge of cycle 2. After the loop is completed in the DSR unit, its result is loaded into the multiplier stage 1, provided the multiplier is unoccupied. To assure that the multiplier is unoccupied when the loop is finished, and to unload the DSR result through the multiplier without delay, a NOP can be inserted in cycle 9 (in the case of single-precision divide). If the unload operation starts in the first half of cycle N, the result is written in the register file in the second half of cycle N + 2, so the entire unload operation takes three cycles. Once the unload starts, it essentially "looks" like any other two-cycle latency multiplier operation. If the multiplier is occupied by another operation when the DSR loop is complete and unload could be initiated, the DSR unit waits until the multiplier becomes available to it. The DSR result will be stored in the first "empty" timing slot after the multiplier has finished its operation. The result of the DSR operation is stored in a register file location whose address is given by the DADD field of the instruction which started the operation in cycle 1.

Once a DSR operation has started, the multiplier may be used for other operations. This multiplier operation may be initiated on any cycle after the DSR instruction has been clocked in.

When a DSR operation is initiated on cycle 1, a special bit, called DSRINP (DSR operation in progress) in the status register, $SR11_6$, is set in cycle 3; it is automatically cleared in the cycle preceding the one in which the DSR result is written into the register file. Using DIV32 as an example, a store status instruction clocked in on the rising edge of cycle 3 will "see" DSRINP = 1, and a store status instruction clocked in on cycle 11 would indicate that DSRINP = 0; that is, there is no DSR operation in progress.

9. Divide/Square Root Unit, continued

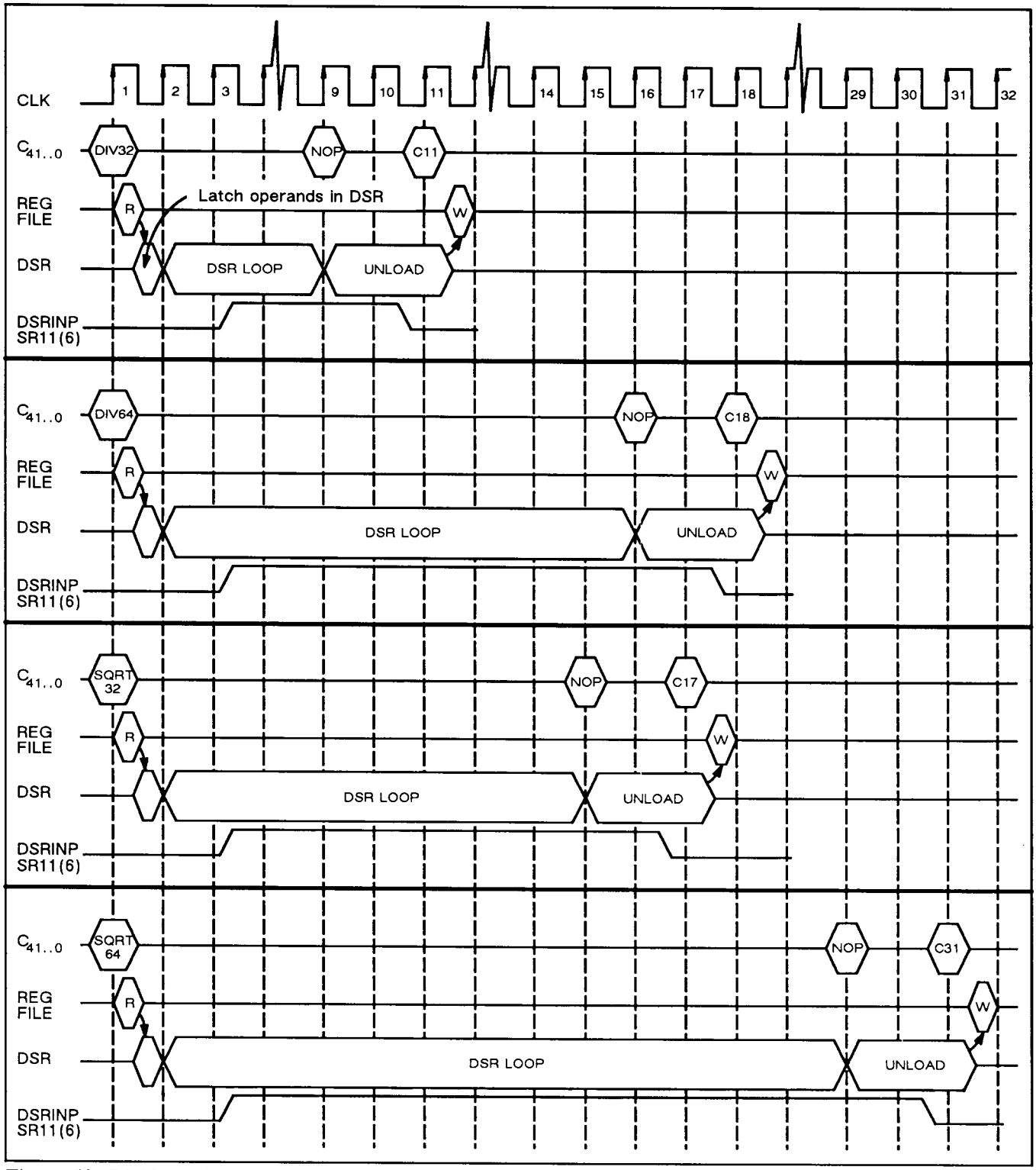


Figure 50. Divide and square root timing when DIVCLK frequency is 2×CLK frequency

9.1. Consecutive DSR Operations

If two DSR operations must follow one another back-to-back, the second operation may be clocked in on the code port only after the first operation has propagated to stage 3 of the multiplier and generated any potential result exceptions. For example, if two single-precision divides must be executed consecutively, and the first one was initiated on cycle 1, then the second one may be initiated on cycle 11. If the second DSR operation is clocked in before the first one has propagated to multiplier stage 2, it will be ignored.

Refer to figure 50, double-precision divide. Any DSR operation issued on cycles 2 through 16 will be ignored. If a DSR operation is issued on cycle 17 (C17), that is, prior to the result of the previous DSR operation, (C1) having propagated to the third stage of the multiplier, it is possible that *both* C1 and C17 will have result exceptions. If this happens, by the time an interrupt handler can examine the status and destination address for C1 this information will have been overwritten as the consequence of an exception on C17. To avoid this situation, C1 should be followed by another DSR operation no sooner than on cycle 18.

9.2. Latency of DSR Operations

The latencies of register-to-register DSR operations, in terms of CLK cycles, are determined according to the following formulas:

- Latency (DIV32) = $\lceil 14/N \rceil + 3$
- Latency (DIV64) = $\lceil 28/N \rceil + 3$
- Latency (SQRT32) = $\lceil 25/N \rceil + 3$
- Latency (SQRT64) = $\lceil 54/N \rceil + 3$

From these formulas, the latencies for the DSR operations with 1× and 2× DIVCLK, are as follows:

In these formulas, the symbols $\lceil\ \rceil$ indicate the ceiling (round up) function, and N is the factor by which the DIVCLK frequency is greater than the CLK frequency; in the case of 2× DIVCLK, for example, N = 2. The number calculated by the ceiling function gives the number of cycles in the loop; the other three cycles are necessary to latch operands into the DSR unit (one cycle) and to unload results through the multiplier (two cycles). The formulas assume that bypassing is used; if not, the latency is one cycle greater in each case.

The latency count is from the DSR instruction to the first instruction that could use the result of the DSR operation as an operand. (This count includes everything necessary to produce the result, including the generation of guard and round bits.) For example, in the case of DIV32 instruction, the result is written into the register file in the second half of cycle 11. However, an instruction that *uses* this result may be clocked in on the rising edge of cycle 11, using register file bypass. This instruction is denoted in the figure as C11; it could be any instruction, including another DSR operation. Therefore, in this case the latency is from the rising edge of cycle 1 to that of cycle 11, which is 10 cycles. This reasoning applies equally to the other cases.

Operation	Latency in cycles of CLK	
	1× DIVCLK	2× DIVCLK
Single-precision divide	17	10
Double-precision divide	31	17
Single-precision SQRT	28	16
Double-precision SQRT	57	30

Figure 51.

10. Temporary Latches

The purpose of temporary latches is to provide necessary bandwidth for chained multiply-add operations. The number of operands needed is four, yet the register file provides only two, and the X or Y register provides the third. The T-latches provide the path from the multiplier output to the ALU input for the fourth operand. This is illustrated in figure 53.

The 3x64 has two temporary latches: T0 and T1. Two latches are needed in order to be able to write interruptible code; in applications where interruptibility is unimportant, only one T-latch need be used.

Since it is impossible to specify a T-latch without also specifying a file register (in the DADD field of the code word), the intermediate result is both latched in a T-latch and written into the register file through the D port; the timing for both activities is the same.

The use of T-latches is demonstrated with a sum-of-products example. In the example, it is desired to perform

$$\sum_{i=1}^{6} XiBi$$

Xi are data inputs through the X port and *Bi* are coefficients read from the B port of the register file.

Cycle	Instruction	Comment
1	$X_1 \times B_1 \rightarrow R3$;	
2	$X_2 \times B_2 \rightarrow R4$;	
3	$X_3 \times B_3 \rightarrow T0$, R30; R29 + T0 → R29;	R3 has $X_1 \times B_1$
4	$X_4 \times B_4 \rightarrow T1$, R31; R29 + T1 → R29;	R4 has $X_2 \times B_2$
5	$X_5 \times B_5 \rightarrow T0$, R30; R3 + T0 → R3;	
6	$X_6 \times B_6 \rightarrow T1$, R31; R4 + T1 → R4;	
7	$X_7 \times B_7 \rightarrow T0$, R30; R3 + T0 → R3;	R3 has $X_1 \times B_1 + X_3 \times B_3$
8	$X_8 \times B_8 \rightarrow T1$, R31; R4 + T1 → R4;	R4 has $X_2 \times B_2 + X_4 \times B_4$
9	NOP	R3 has $X_1 \times B_1 + X_3 \times B_3 + X_5 \times B_5$
10	R3 + R4 → R5;	R4 has $X_2 \times B_2 + X_4 \times B_4 + X_6 \times B_6$
11	...	
12	...	R5 has $\sum X_iB_i$, i = 1 to 6

Figure 52. Example of the use of temporary latches

In this example, register R3 accumulates the sum of odd products, and R4 even ones; registers R29–R31 are "garbage" registers. In cycles 1 and 2, the first multiplications are performed, using ordinary multiplication operations. Beginning with cycle 3, chained multiply-add instructions are used. On cycles 3 and 4 only the multiplication is relevant; chained multiply-add instructions are used to write the result into a T-latch and bypass it into the ALU as an operand on cycles 5 and 6, respectively.

In cycle 7, T-latch bypassing is activated when T0 is used as an operand in the addition while at the same time being written with the result of the multiplication in cycle 5. In cycle 7, register file bypassing in register-to-register operations is activated when R3 is used in the addition as an operand while at the same time R3 is being written with the result from addition in cycle 5. Note that operands X_7, B_7 and X_8, B_8 are not used in this example to calculate the sum, but are shown to illustrate that this example can be extended. On cycle 9, R3 holds the sum of odd products, and on cycle 10, R4 holds the sum of even products. On cycle 10, the two registers are added, and the final result is available on cycle 12.

The example shows that cycle 9 contains a NOP; of course, another instruction (unrelated to the task at hand) could be specified in this cycle.

T-latch bypassing is independent of the state of the Bypass On bit in the status register ($SR1_6$) and is always enabled on chained multiply-add instructions.

Note that the T0 and T1 latches are used on alternate cycles to assure the interruptibility of this code. Different registers R30 and R31 are used to assure that this code is interruptible not only in the general sense but also IEEE interruptible. See section 16 for more details.

The operation of the T-latches in this example is shown in figure 54.

10. Temporary Latches, continued

The T-latches are used only as part of chained multiply-add instructions operating only on single- and double-precision floating-point operands. Values in the T-latches remain there until overwritten by a subsequent chained multiply-add instruction.

Bypassing of temporary latches cannot be disabled regardless of the Bypass On bit in the status register.

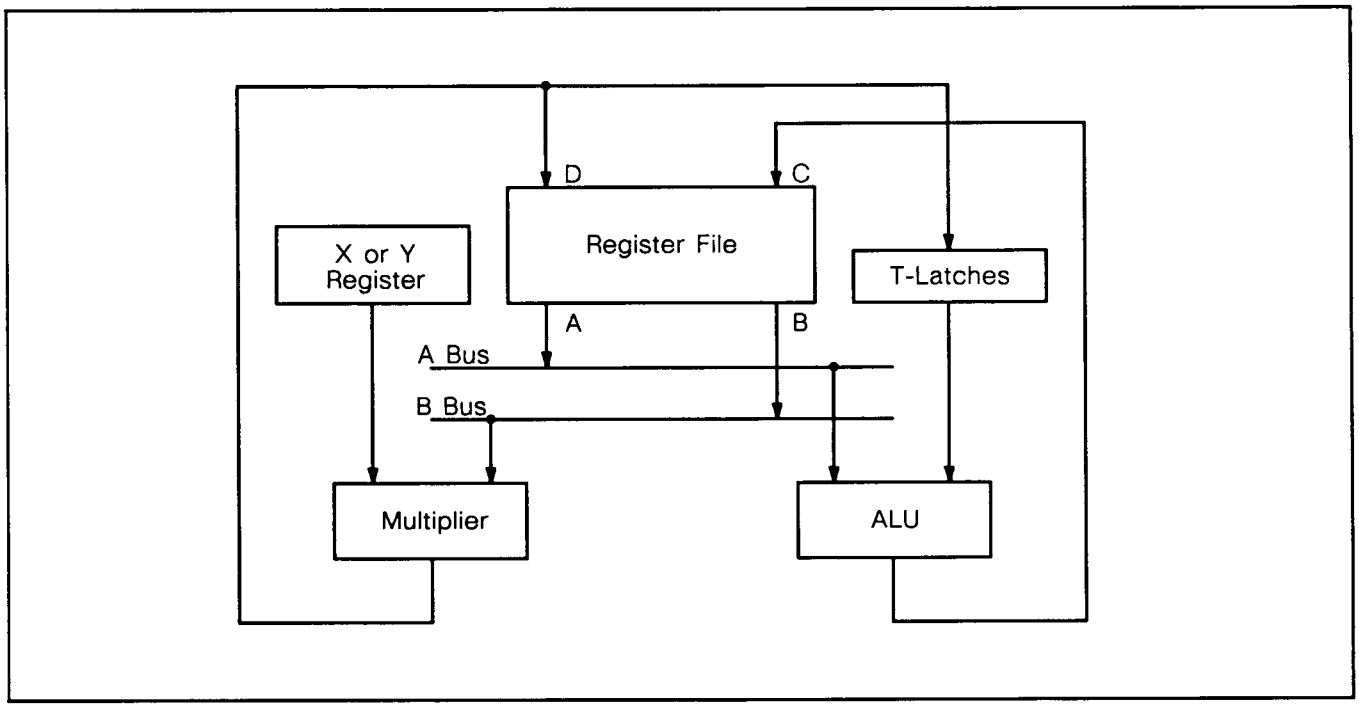


Figure 53. Temporary latch data path

10. Temporary Latches, continued

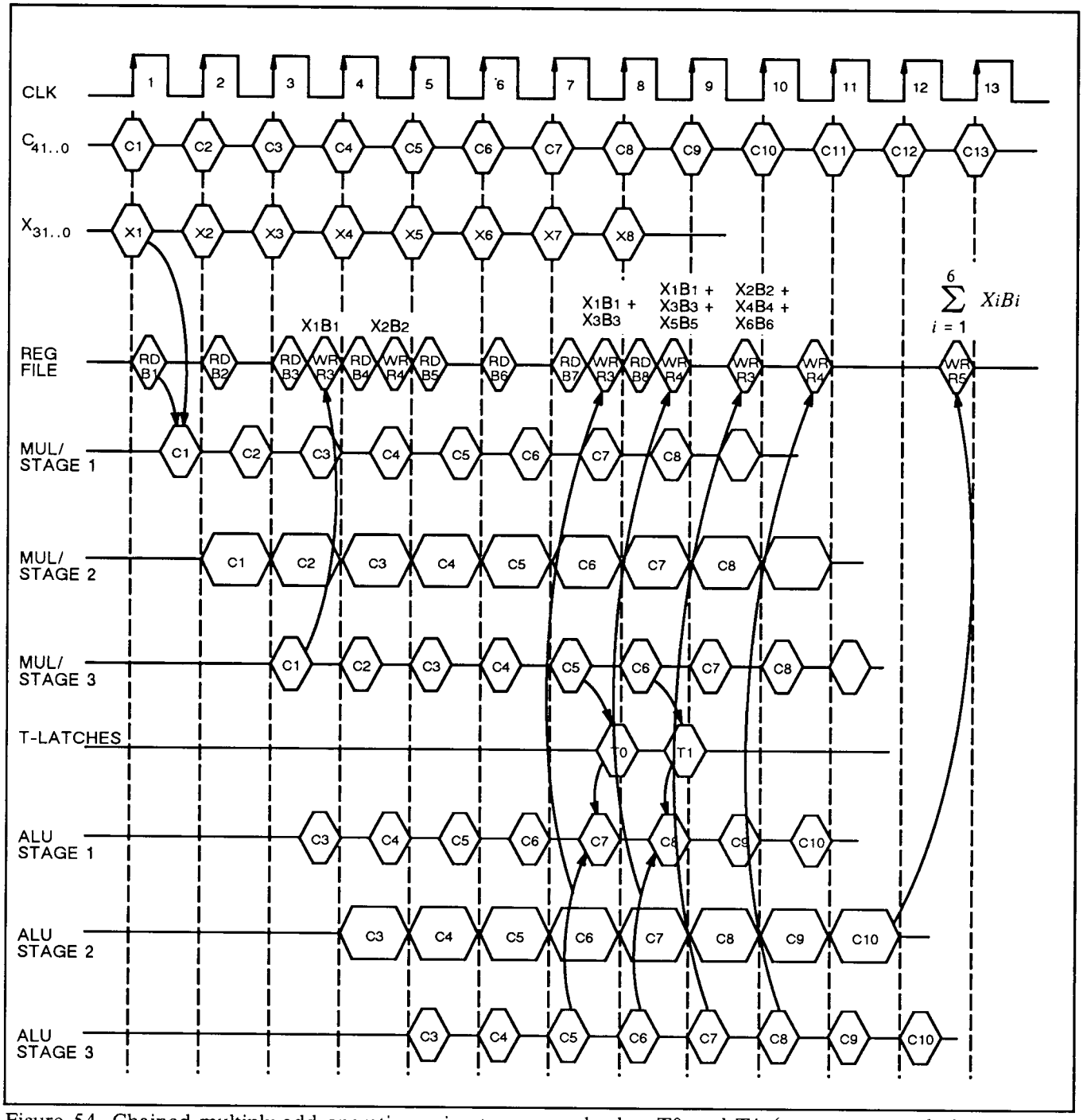


Figure 54. Chained multiply-add operation using temporary latches T0 and T1 (assume two-cycle latency mode and undelayed load)

11. Status Output

The 3x64 provides a 4-bit status output, $S_{3..0}$.

Source exceptions are delayed internally so that they are signaled on the same cycle on which result exceptions are signaled. This is true regardless of the latency of the operation. The exception signaled for one operation is either source or result, never both; conflicts are resolved according to priorities in figure 56.

The timing diagrams of the status output generation are shown in figure 58, for the following three cases:

- Multiplier operation only (no concurrent ALU operation)
- ALU operation only (no concurrent multiplier operation)
- Concurrent multiplier and ALU operations

The status of a DSR operation is generated following the unload of the DSR result through the multiplier; it has the same timing as that of a multiplier operation.

The state of the $S_{3..0}$ pins remains unchanged until updated by another operation.

Note that if an instruction is neutralized, stalled, or aborted, its status is not provided on the $S_{3..0}$ pins. (The status indicated is 4 or EXT.POS.)

The status output codes and the priorities according to which any conflicts are resolved—as well as the mnemonics used—are listed below. Note that these status outputs are not necessarily the same as the status information stored in the status register upon register file writes.

NRM	Normalized number
EXT	Exact
INX	Inexact
UNRM	Unrounded normalized number
UNF	Underflow
OVF	Floating-point overflow
IOVF	Integer overflow
POS	Positive
NEG	Negative
NaN	Not a Number
INV	Invalid operation
DVZ	Divide by zero
. (period)	And

Figure 55. Status mnemonics

$S_{3..0}$		Status/Exception	Priority
Decimal	Binary		
0	0000	Result is + 0	Lowest
1	0001	Result is − 0	
2	0010	Result is + ∞	
3	0011	Result is − ∞	
4	0100	EXT.POS	
5	0101	EXT.NEG	
6	0110	INX.POS	
7	0111	INX.NEG	
8	1000	IOVF	
9	1001	OVF	
10	1010	UNRM.EXT	
11	1011	UNRM.INX	
12	1100	DNRM	
13	1101	DVZ*	
14	1110	INV	
15	1111	NaN	Highest

* This code is not applicable to the ALU.

Figure 56. Status encoding and priority

11.1. Status Output for Compare Operations

The meaning of the $S_{3..0}$ output for compare operations is given in figure 57.

$S_{3..0}$	Result of compare operation		
0	zero	eq	(x = y)
4	greater than	gt	(x > y)
5	less than	lt	(x < y)
15	unordered	uord	(x ? y)

Figure 57.

The timing diagram for the compare operation is shown in figure 58.

Compare operations are executed by the multiplier. The timing for status output of compare operations is the same as that for the status of ALU or two-cycle latency multiplier operations. For details on compare instructions, see section 17.7.

11.1. Status Output for Compare Operations, continued

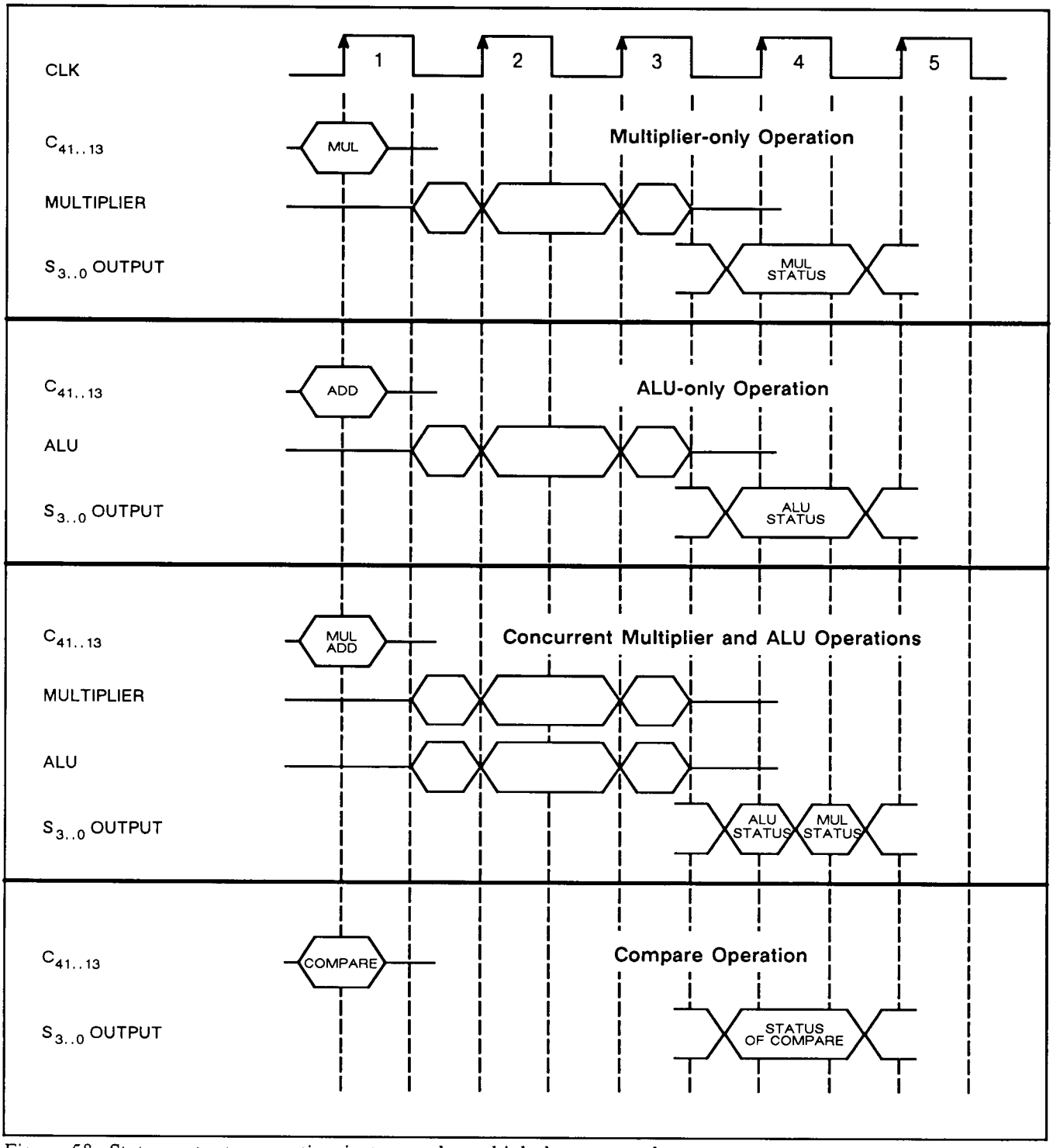


Figure 58. Status output generation in two-cycle multiply latency mode

12. Status Register

The status register is used to specify the device's mode bits and to store status. It is used to recover from exceptions in a pipelined environment and for context switching.

When switching modes, the change takes place in the very next cycle, potentially corrupting current computations; therefore no operations should be in progress.

12.1. Status Register Structure

The status register consists of twelve 8-bit registers whose structure is given in figures 59 and 60. These registers are named SRNs, where N identifies one of the twelve registers and ranges from 0 to 11, and the subscript s indicates the range of bits in the register, from 0 (LSB) to 7 (MSB).

SR#	BIT#								COMMENTS
	7	6	5	4	3	2	1	0	
SR0	Multiplier Latency	FPEX-Sticky	0	0	Internal NEUT ON	Rounding Mode		Fast Mode	Modes
SR1	IEEE Software Underflow	Bypass on	FPEX-Delay	I/O Mode					Modes
SR2	NaN EN	INV EN	DVZ EN	DNRM CONTROL	OVF EN	UNF CONTROL	INX EN	IOVF EN	Trap Enables
SR3	NaN	INV	DVZ	DNRM	OVF	UNF	INX	IOVF	Sticky Bits
SR4	0	TDEST0		MDEST0					Destination
SR5	0	0	0	ADEST0					Destination
SR6	ASTAT0			MSTAT0					Status
SR7	0	0	0	DIVDEST					Destination
SR8	0	TDEST1		MDEST1					Destination
SR9	0	0	0	ADEST1					Destination
SR10	ASTAT1			MSTAT1					Status
SR11	FPEX-Taken	DSR in progress	FPCN	Carry	DIVSTAT				Status

Figure 59. Status register structure

12.1. Status Register Structure, continued

Field	Name	Values	Meaning
$SR0_0$	FAST MODE	0 1	IEEE Fast
$SR0_{2..1}$	ROUNDING MODE	00 01 10 11	Round to nearest Round to zero Round to positive infinity Round to negative infinity
$SR0_3$	INTERNAL NEUT ON	0 1	Internal NEUT off (no internal NEUT source) Internal NEUT on (causes one-cycle NEUT)
$SR0_{5..4}$	RESERVED	00	Reserved, must be loaded as zero
$SR0_6$	FPEX-STICKY	0 1	FPEX- pulsed (high true) FPEX- sticky (low true)
$SR0_7$	MULTIPLIER LATENCY	0 1	Two-cycle multiplier latency Three-cycle multiplier latency
$SR1_{4..0}$	I/O MODE	00000-11111	Load/store modes, see section 5.5.1.
$SR1_5$	FPEX- DELAY	0 1	FPEX- undelayed FPEX- delayed
$SR1_6$	BYPASS ON	0 1	Register file bypass disabled Register file bypass enabled
$SR1_7$	IEEE SOFTWARE UNF	0,1	Software underflow bit, used by IEEE trap handlers In non-IEEE systems, should be loaded as zero
$SR2_0$	IOVF EN	0 1	Integer overflow exception disabled Integer overflow exception enabled
$SR2_1$	INX EN	0 1	Inexact exception disabled Inexact exception enabled
$SR2_2$	UNF CONTROL	0 1	Underflow result not rounded Underflow result rounded
$SR2_3$	OVF EN	0 1	Overflow exception disabled Overflow exception enabled
$SR2_4$	DNRM CONTROL	0 1	Denormalized inputs treated as zero Denormalized inputs cause exception
$SR2_5$	DVZ EN	0 1	Divide-by-zero exception disabled Divide-by-zero exception enabled
$SR2_6$	INV EN	0 1	Invalid operation exception disabled Invalid operation exception enabled
$SR2_7$	NaN EN	0 1	Not-a-number exception disabled Not-a-number exception enabled

Figure 60. Status register fields

12.1. Status Register Structure, continued

Field	Name	Values	Meaning
$SR3_0$	IOVF	0	Integer overflow exception has not occurred
		1	Integer overflow exception has occurred
$SR3_1$	INX	0	Inexact exception has not occurred
		1	Inexact exception has occurred
$SR3_2$	UNF	0	Underflow exception has not occurred
		1	Underflow exception has occurred
$SR3_3$	OVF	0	Overflow exception has not occurred
		1	Overflow exception has occurred
$SR3_4$	DNRM	0	Denormalized input exception has not occurred
		1	Denormalized input exception has occurred
$SR3_5$	DVZ	0	Divide-by-zero exception has not occurred
		1	Divide-by-zero exception has occurred
$SR3_6$	INV	0	Invalid operation exception has not occurred
		1	Invalid operation exception has occurred
$SR3_7$	NaN	0	Not-a-number exception has not occurred
		1	Not-a-number exception has occurred
$SR4_{4..0}$	MDEST0	00000-11111	Multiplier destination address for operation 0
$SR4_{6..5}$	TDEST0		Temporary latch destination for operation 0
		00	T-latches not used as destination
		01	T-latch destination is T_0
		10	T-latch destination is T_1
$SR4_7$	RESERVED	0	Reserved, must be loaded as zero
$SR5_{4..0}$	ADEST0	00000-11111	ALU destination address for operation 0
$SR5_{7..5}$	RESERVED	0	Reserved, must be loaded as zero
$SR6_{3..0}$	MSTAT0	0000-1111	Multiplier status for operation 0
$SR6_{7..4}$	ASTAT0	0000-1111	ALU status for operation 0
$SR7_{4..0}$	DIVDEST	00000-11111	DSR destination address
$SR7_{7..5}$	RESERVED	0	Reserved, must be loaded as zero

Figure 60. Status register fields, continued

12.1. Status Register Structure, continued

Field	Name	Values	Meaning
$SR8_{4..0}$	MDEST1	00000-11111	Multiplier destination address for operation 1
$SR8_{6..5}$	TDEST1		Temporary latch destination for operation 1
		00	T-latches not used as destination
		01	T-latch destination is T_0
		10	T-latch destination is T_1
$SR8_7$	RESERVED	0	Reserved, must be loaded as zero
$SR9_{4..0}$	ADEST1	00000-11111	ALU destination address for operation 1
$SR9_{7..5}$	RESERVED	0	Reserved, must be loaded as zero
$SR10_{3..0}$	MSTAT1	0000-1111	Multiplier status for operation 1
$SR10_{7..4}$	ASTAT1	0000-1111	ALU status for operation 1
$SR11_{3..0}$	DIVSTAT	0000-1111	DSR operation status
$SR11_4$	CARRY	0	Carry/borrow bit is zero
		1	Carry/borrow bit is one
$SR11_5$	FPCN	0	Result of compare operation is false
		1	Result of compare operation is true
$SR11_6$	DSRINP	0	A DSR operation is not in progress
		1	A DSR operation is in progress
$SR11_7$	FPEX TAKEN	0	No enabled exception has occurred
		1	An enabled exception has occurred

Figure 60. Status register fields, continued

12.2. Status Register Load/Store

Each of the twelve status registers can be loaded through the X port and stored through both the X and Z ports. The timing depends on the I/O mode ($SR1_{4..0}$). See figure 61. The bits within each register are not individually addressable; it is possible to load/store only an entire register at a time, one per cycle.

Status register loads occur through the most-significant byte of the X port, $X_{31..24}$. The other 24 bits on that port must be set to zero.

The contents of a status register are stored simultaneously through the most-significant byte of the X port ($X_{31..24}$) and Z port ($Z_{31..24}$). The other bits of the X and Z ports, bits 23..0, are undefined. If a double-pump store mode is in effect, the output is valid for the entire length of time when both MS and LS halves of a corresponding double-precision data word would be valid. Status register load/store instructions are detailed in section 17.

The following sections explain each of the twelve status registers. Refer to figures 59 and 60.

12.2. Status Register Load/Store, continued

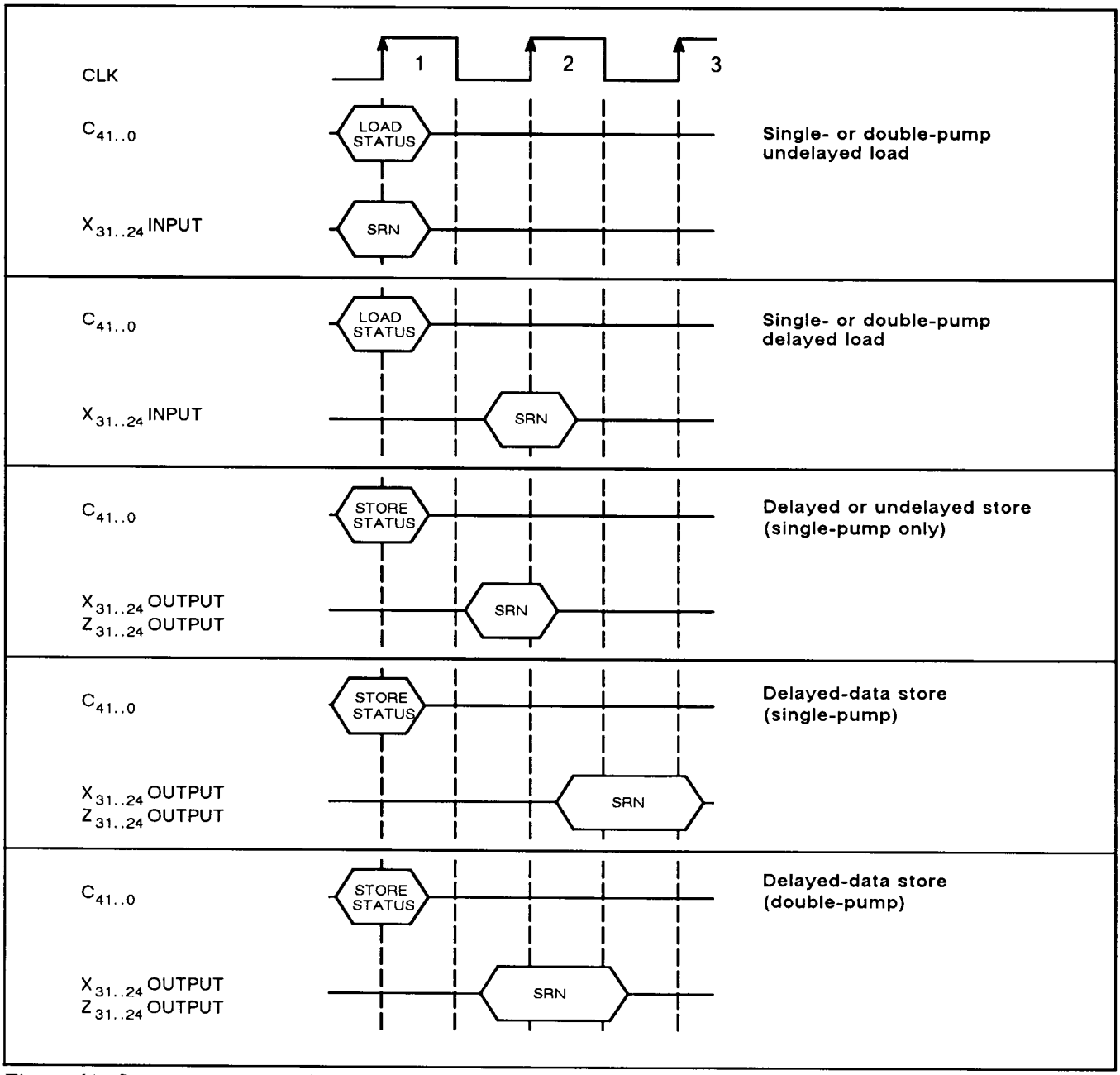


Figure 61. Status register load/store

12.3. Mode Registers: SR0 and SR1

These two registers are used to specify the modes in which the 3x64 will be used.

12.3.1. FAST MODE

The 3x64 has two modes of handling operations with denormalized operand inputs and operations that produce underflowed results. The FAST mode affects only results: underflows are "flushed" to zero. Inputs and outputs are decoupled, as shown in figure 66: the setting of the DNRM control bit ($SR2_4$) does not determine whether underflowed results are flushed to zero; conversely, the setting of the FAST mode bit ($SR0_0$) does not determine whether denormalized inputs are treated as zero.

In this table "DNRM" refers to denormalized numbers and "UNRM" to unrounded normalized numbers. Both are defined in section 19.

In the IEEE mode, the denormalized operands and results can be handled, in system software, in a manner consistent with the *IEEE Standard For Binary Floating-Point Arithmetic*.

12.3.2 ROUNDING MODE

All four rounding modes specified by the IEEE standard are supported.

12.3.3. INTERNAL NEUT ON

If INTERNAL NEUT is on and an enabled exception occurs, then the *operation* portion ($C_{41..13}$) of the current instruction is neutralized and the *entire* code word ($C_{41..0}$) is saved on-chip in the code register and FPEX TAKEN bit ($SR11_7$) is asserted.

The $SR0_3$ bit has no effect on the I/O portion of the instruction ($C_{12..0}$) and it completes without delay. Refer to sections 14 and 15.

12.3.4. FPEX STICKY

When an enabled exception is detected, the FPEX output is asserted, and the FPEX TAKEN bit in the status register, $SR11_7$, is set. In sticky mode, the FPEX output stays asserted until the FPEX TAKEN bit is explicitly cleared. This is done by loading SR11 with a new value, such that $SR11_7 = 0$. In pulsed mode, FPEX output stays asserted for one cycle only. See figures 70 and 71.

$SR0_0$ =	0	IEEE mode
	1	FAST mode

Figure 62. Fast and IEEE modes

$SR0_{2..1}$ =	00	Round to nearest
	01	Round to zero
	10	Round to positive infinity
	11	Round to negative infinity

Figure 63.

$SR0_3$ =	0	INTERNAL NEUT off
	1	INTERNAL NEUT on

Figure 64.

$SR0_6$ =	0	FPEX is pulsed, polarity is positive true
	1	FPEX is sticky, polarity is negative true

Figure 65.

Denormalized Number Control $SR2_4$	Denormalized Input	FAST Mode $SR0_0$	Underflowed Output
0	Treated as 0	0	UNRM and underflow exception
0	Treated as 0	1	0
1	DNRM exception	0	UNRM and underflow exception
1	DNRM exception	1	0

Figure 66.

12.3. Mode Registers: SR0 and SR1, continued

12.3.5. MULTIPLIER LATENCY

The effects of multiplier latency mode are described in section 7.

12.3.6. I/O MODE

This five-bit field $SR1_{4..0}$ specifies load/store modes. See section 5.5.1 for the specific bit combinations that select desired load/store modes.

12.3.7. FPEX DELAY

When an enabled exception occurs, two events always happen: the FPEX TAKEN bit, $SR11_7$ is set, and the FPEX output is asserted. The FPEX output is asserted either at the end of the same cycle in which the exception was detected, or in the beginning of the following one, depending on the FPEX delay mode. See figures 70 and 71.

If FPEX is in the undelayed mode, and an enabled exception occurs, the FPEX output will be asserted at the end of the same cycle in which the exception was detected.

If FPEX is in the delayed mode, and an enabled exception occurs, the FPEX output will be asserted on the cycle following, that is, delayed with respect to, the one in which the exception was detected. The effects of FPEX delay are treated in section 15.

12.3.8. BYPASS ON

Register file bypass logic is activated if $SR1_6$ is asserted and one of the following conditions occurs:

- CADD or DADD = EFADD on store (except on delayed store)
- AADD or BADD = EFADD on load
- The destination address from an earlier instruction equals the address for one of the operands of the current instruction
- EFADD of a delayed load instruction equals EFADD of an immediately following store instruction (except delayed store). In this case, the data just loaded will bypass the register file and will be stored.
- An undelayed load (data is supplied, say, through the X port) and same-cycle store (except delayed store) (data is driven out, say, through the Z port) both use the same EFADD field of the code word. In this case, the data loaded through the X port will be stored through the Z port on the same cycle.

Note that the equality in the above conditions exists not between the addresses of the same code word but with respect to a given cycle. For example, assume that on cycle 1 a multiply instruction is issued. In two-cycle latency mode, it completes and the result is automatically written into the register file in cycle 3. The write address is specified by the DADD field of the multiply instruction in cycle 1. Assume now that on cycle 3 an undelayed store instruction is issued. Register file bypass logic is activated if the address specified by the EFADD field of the store instruction of cycle 3 is equal to the DADD field of the multiply instruction of cycle 1.

C1 R1 × R2 → R3
C2 NOP
C3 Store R3

$SR0_7$ =	0	Two-cycle multiplier latency
	1	Three-cycle multiplier latency

Figure 67.

$SR1_5$ =	0	FPEX is undelayed
	1	FPEX is delayed

Figure 68.

$SR1_6$ =	0	Register file bypasss is disabled
	1	Register file bypasss is enabled

Figure 69.

12.4. SR2 and SR3: Sticky Bits and Their Enables

SR3 contains eight sticky bits to signal the occurrence of exceptions. The reason for having SR3 is to provide necessary information to an exception handling routine. SR2 contains trap enables that correspond to the sticky bits in SR3. A sticky bit *must* be enabled for the corresponding exception to be signaled by the FPEX output. There is one exception to the previous statement: in IEEE mode ($SR0_0$=0), underflow is signaled regardless of whether it is enabled.

When an exception occurs, the corresponding bit in SR3 is set regardless of the trap enable bits (see figures 70 and 71) and remains set until it is cleared by loading SR3 with a new value. The DNRM sticky bit is handled differently. If DNRM Enable = 0, then the DNRM bit in SR3 is never set, and DNRMs are treated as zero. Just exactly when the SR3 bit is set is immaterial so long as it has correct state by the time an interrupt handler uses it. It is possible that on the same cycle, say cycle 3, a result exception from instruction C1 and a source exception from a different instruction, C3, will both be detected, and corresponding sticky bits set. Since a single operation may set either source or result exception, but never both, any resulting ambiguities may be resolved in a trap handler.

The FPEX output is not affected by the state of sticky bits; for example, it cannot be reset by clearing the sticky bits in SR3. Conversely, merely loading sticky bits into SR3 will not cause the FPEX output to be asserted. FPEX output is asserted only as a result of an enabled exception. To reset FPEX output, FPEX TAKEN bit, $SR11_7$, must be explicitly cleared.

12.4.1. CHANGING TRAP ENABLES

When changing trap enables, old status information in MDEST0, MSTAT0, MDEST1, MSTAT1, ADEST0, ASTAT0, ADEST1, ASTAT1, TDEST0, TDEST1, DIVDEST, DIVSTAT, and SR3 (see figure 59) in the status register should be cleared prior to beginning operations in the arithmetic units. Also, trap enables should be changed only after all operations have completed. The reason for this is to avoid changing trap enables in the middle of an incomplete operation as this may cause unpredictable results.

12.4.2. $SR2_2$ UNDERFLOW CONTROL

This bit controls the behavior of underflow in IEEE mode ($SR0_0$=0). In FAST mode ($SR0_0$=1) the underflow control bit is ignored.

12.4.3. $SR2_4$ DENORMALIZED NUMBER CONTROL

This bit controls how denormalized numbers are treated when received as inputs to an instruction.

If $SR2_4$=1, denormalized inputs cause an exception.

If $SR2_4$=0, denormalized inputs are treated as zero and none of the following sticky bits is set: DNRM ($SR3_4$), INX ($SR3_1$), UNF ($SR3_2$). For example, when $SR2_4$=0, multiplying 1.0 and a denormalized number yields 0 and neither the INX nor the UNF nor the DNRM bit is set.

12.4. SR2 and SR3: Sticky Bits and Their Enables, continued

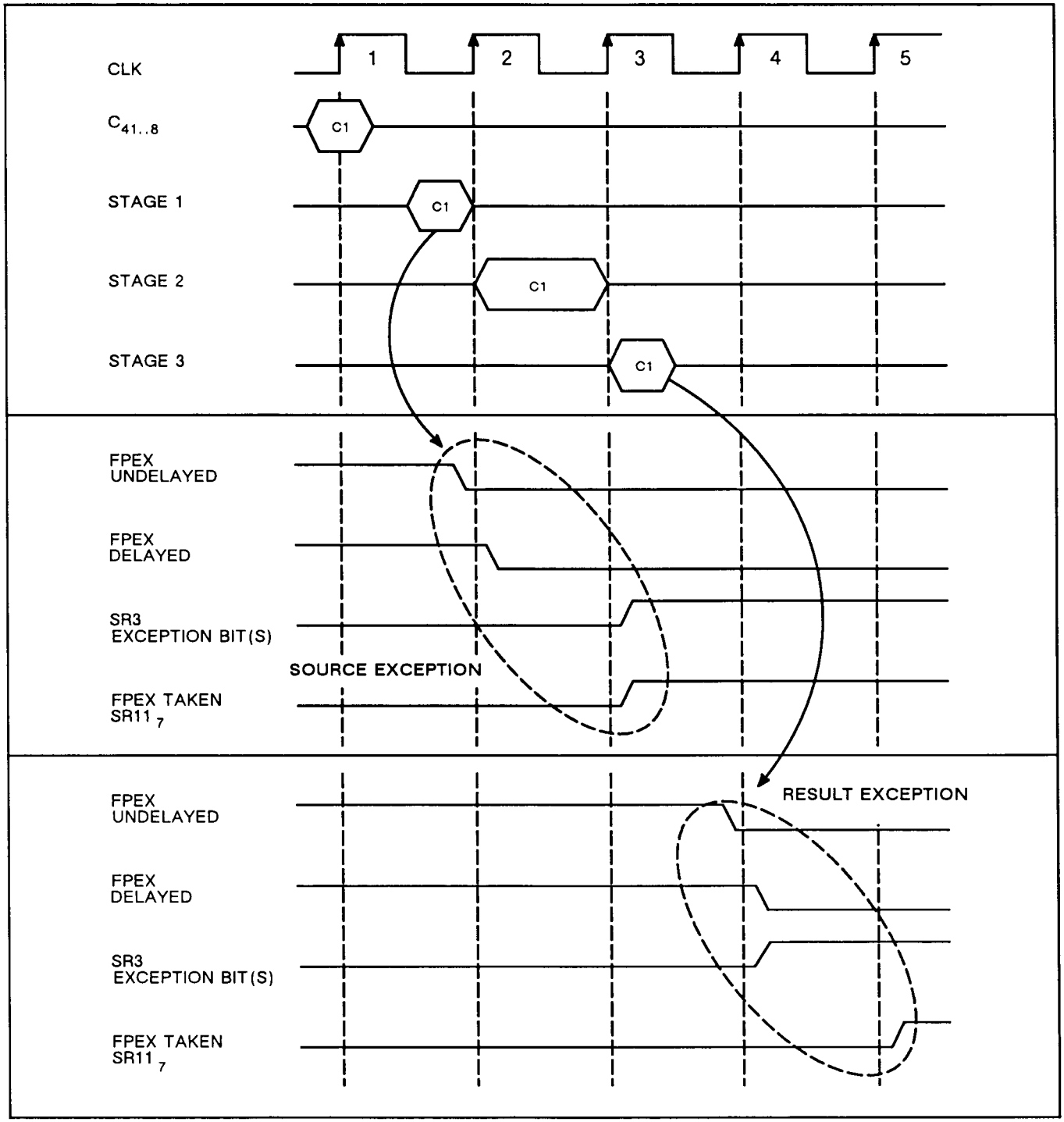


Figure 70. FPEX assertion for source and result exceptions (FPEX sticky)

12.4. SR2 and SR3: Sticky Bits and Their Enables, continued

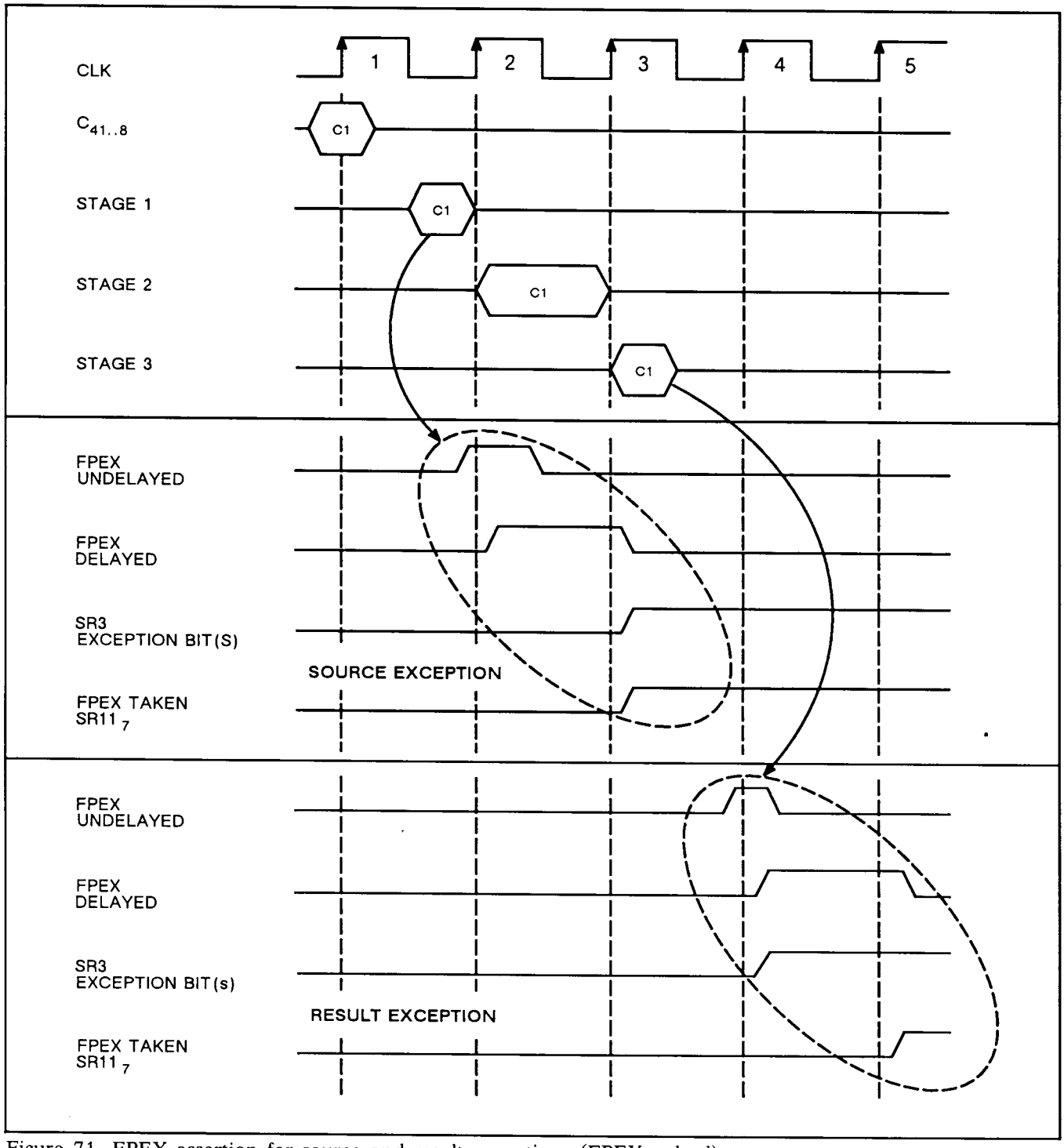


Figure 71. FPEX assertion for source and result exceptions (FPEX pulsed)

12.5. SR4 Through SR11: Status and Destination Addresses

Since the multiplier and the ALU each contains three stages, at any given time they may be executing six operations. In addition, there may be a seventh operation in the DSR unit. If an operation in either first stage causes a source exception, INTERNAL NEUT can still neutralize the instruction starting in the first stage. However, should the instructions in the second and third stages generate result exceptions, it is too late to "kill" them, and they will write to the register file. The same is true for the result of the DSR operation.

Thus, up to five registers in the file may contain result exceptions by the time an interrupt handler examines status. See figure 75. Status registers SR4 through SR11 store addresses of file registers containing exceptions and the associated status information which is needed to recover from the exceptions and "fix up" the results.

The destination addresses for multiplier operations are stored in the status register in registers MDEST0 and MDEST1. The status of these operations is also stored in the status register, in registers MSTAT0 and MSTAT1. Similar registers associated with the ALU are named ADEST0, ADEST1 and ASTAT0, ASTAT1. T-latch addresses are stored in TDEST0, TDEST1. (There is no need for corresponding STAT registers because this information is already stored in MSTAT0, MSTAT1. The destination of divide/square root unit operations is stored in the DIVDEST register, and the associated status in DIVSTAT register. All of these registers are illustrated in figures 59 and 60.

The status registers associated with pipelined arithmetic units — the multiplier and the ALU — are arranged as a two-slot queue (FIFO), as illustrated in figure 76. Every time the multiplier writes to the register file, the following status register updates take place:

Current multiplier destination and status	→	MDEST0, MSTAT0
MDEST0, MSTAT0	→	MDEST1, MSTAT1

Figure 72.

Whenever the ALU writes to the register file, the status register is updated similarly:

Current ALU destination and status	→	ADEST0, ASTAT0
ADEST0, ASTAT0	→	ADEST1, ASTAT1

Figure 73.

A simple example is illustrated in the timing diagram of figure 77.

The register set associated with the multiplier is completely independent from that associated with the ALU or the DSR unit; therefore, the timing of their updates is independent. For example, MDEST0, MSTAT0 update and ADEST0, ASTAT0 update do not have to be from instructions that occurred in the same cycle. The timing of T-latch update is the same as that for the multiplier, since a T-latch address can be specified only if a register file address is also specified.

If both the multiplier and the ALU are writing to the register file simultaneously, both queues are advanced.

The foregoing discussion implies that the multiplier and the ALU status information may be out of sync in time. For example, the last multiplier operation may have occurred several cycles in the past, and that operation caused no exception. Its destination and status are stored in the status register. Following that operation, only ALU instructions were executed. When one of them causes an exception, and an exception handler examines the contents of the status register, it will find the destination and status from the last multiplier operation and the destination and status from the current ALU operation which caused the exception. The interrupt handler would look at trap enables as well as status and will determine that the exception was caused by the current ALU operation, and that the old multiplier result is valid. If the multiplier operation did have an exception, the FPEX output would have been asserted at the time the operation was being executed.

12.5. SR4 Through SR11: Status and Destination Addresses, continued

Code		Status
Decimal	Binary	
0	0000	NRM.EXT.32
1	0001	NRM.INX.32
2	0010	UNRM.EXT.32
3	0011	UNRM.INX.32
4	0100	OVF.32
5	0101	*
6	0110	*
7	0111	*
8	1000	NRM.EXT.64
9	1001	NRM.INX.64
10	1010	UNRM.EXT.64
11	1011	UNRM.INX.64
12	1100	OVF.64
13	1101	OVF[1]
14	1110	UNF[2]
15	1111	IOVF[3]

* This code is reserved

[1] This status occurs in the ALU upon F64 → F32 operation.

[2] This status occurs in the ALU upon F64 → F32 operation if the underflow trap is enabled.

[3] This status occurs upon integer operations in the Multiplier and Float → Fix operations in the ALU.

Figure 74. Status encoding

When changing trap enables, one must be careful to first clear old status information. Before clearing it, this information should first be stored and examined for usefulness.

Figure 74 lists multiplier, ALU and divide/square root unit status codes which are used to update MSTAT, ASTAT, and DIVSTAT registers. These codes are identical for all three arithmetic units except for the last three codes. The mnemonics used in this figure are defined in figure 55.

For example, UNRM.INX.64 means that the result of an operation is a 64-bit unrounded normalized floating-point number which is inexact.

12.5. SR4 Through SR11: Status and Destination Addresses, continued

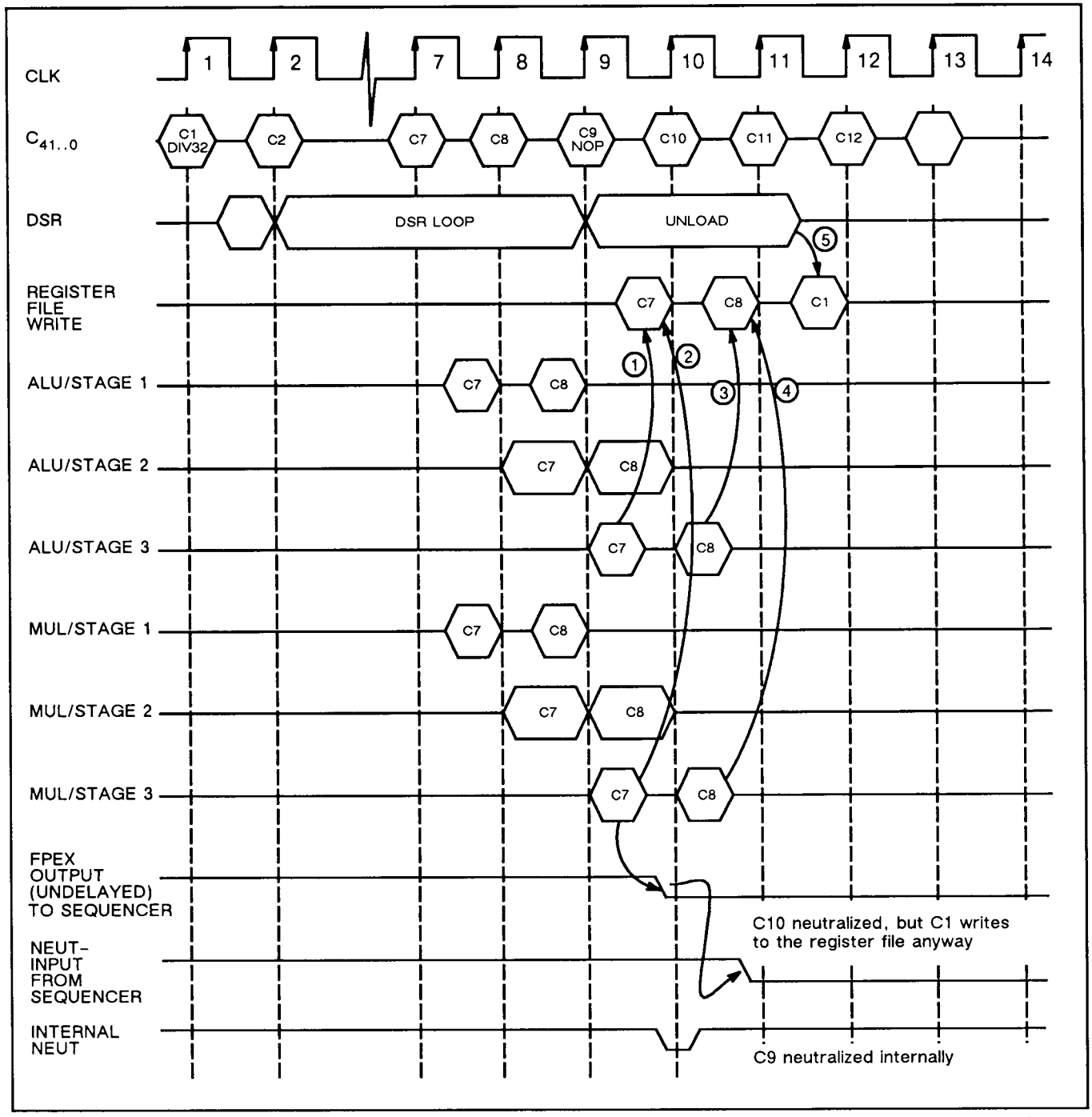


Figure 75. Up to five exception results may be stored in register file by the time interrupt handler examines status

12.5. SR4 Through SR11: Status and Destination Addresses, continued

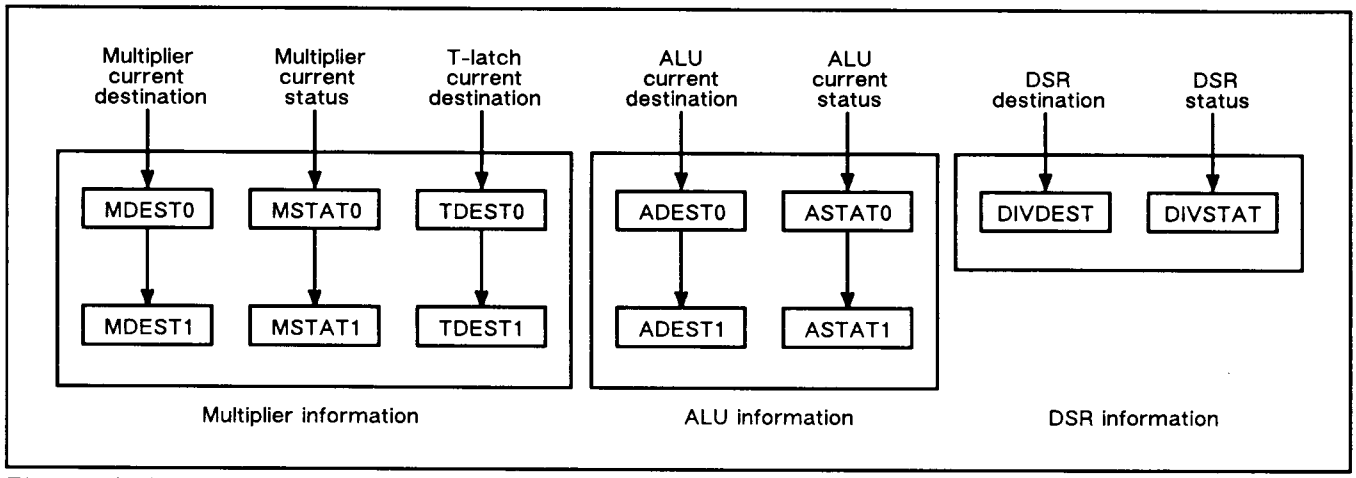


Figure 76. Operations' destination/status information and their storage in the status register

12.5. SR4 Through SR11: Status and Destination Addresses, continued

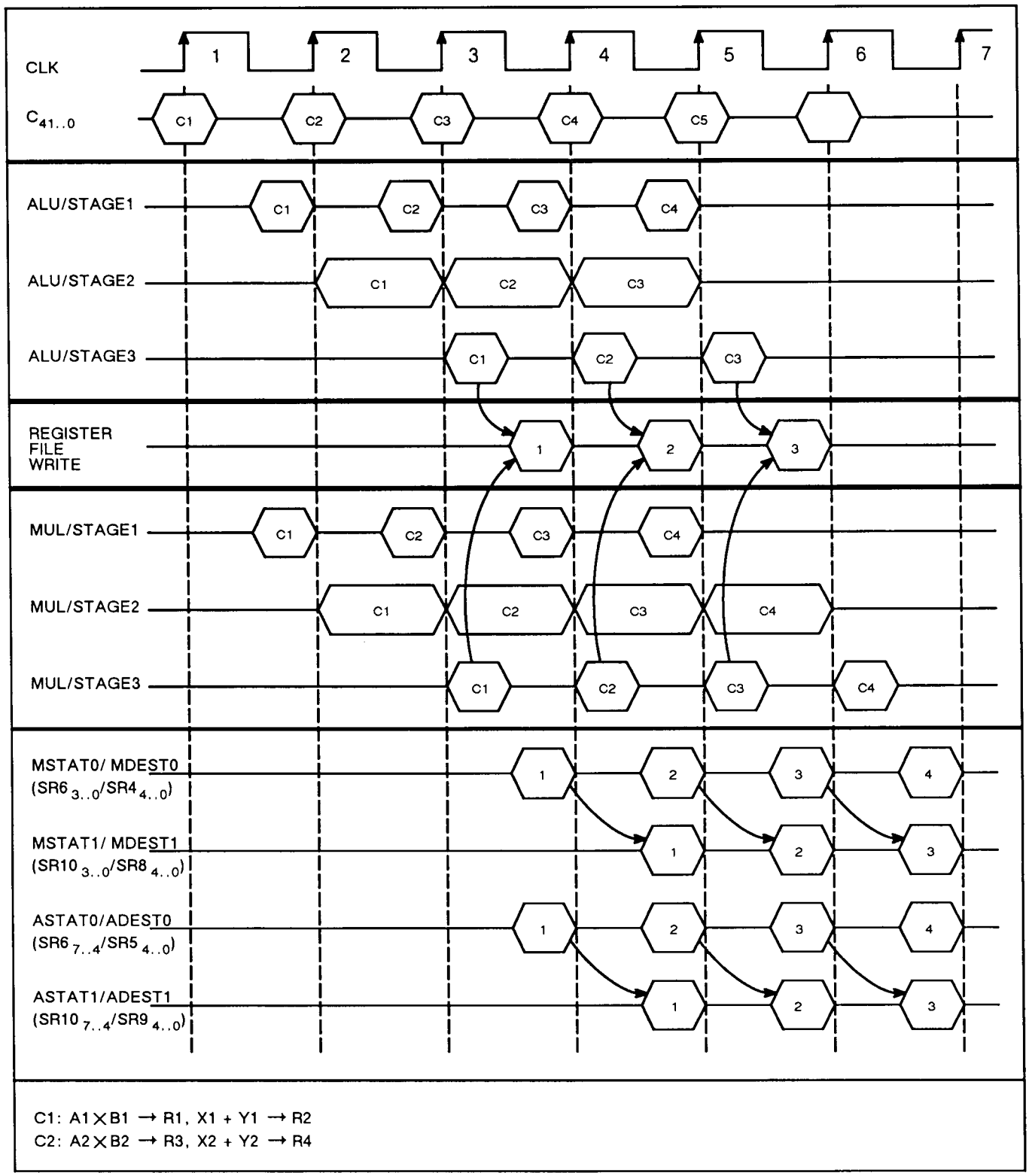


Figure 77. Result status update in the status register

12.5. SR4 Through SR11: Status and Destination Addresses, continued

12.5.1. $SR11_4$: THE CARRY BIT

The carry bit is provided for implementation of multi-precision integer operations, as shown in the example below.

The carry bit is updated only as a result of the following *integer* operations (in the following A′ = not A):

- Add (A + B)
- Add with carry (A + B + Carry)
- Subtract (A′ + B + 1)
- Subtract with borrow (A′ + B + Carry)
- Integer negation (I32′ + 1)
- Integer with carry (I32 + Carry)
- Integer negation with borrow (I32′ + Carry)

Any instruction that uses carry (or borrow) obtains it from $SR11_4$. This bit may also be pre-loaded with a known carry value by loading SR11.

The state of the carry bit is preserved until changed by a subsequent operation. The timing of carry bit update is given in figure 78. Even though there is not a separate carry bit output pin, the carry information may be obtained by using the store SR11 instruction.

EXAMPLE

A 64-bit integer ADD may be implemented as follows:

```
Add
NOP
Add with carry, store integer
NOP
Store integer
```

In this example, the NOPs may be replaced by 64-bit integer operations which are interleaved with the ones shown (however, this will produce uninterruptible code).

12.5.2. $SR11_5$: FLOATING-POINT CONDITION (FPCN)

Floating-point condition is updated as a result of 32-bit or 64-bit compare operations only; it stays unchanged until updated by the next compare instruction. In addition to a bit in the status register, the 3x64 provides an FPCN output pin. FPCN timing — both the status register update and the output pin update — is shown in figure 79. FPCN is high true. See also section 17.7.

12.5.3. $SR11_6$: DSR OPERATION IN PROGRESS (DSRINP)

The DSRINP bit is used to indicate that a divide or a square root operation is in progress. This bit is set on the third cycle after the DSR instruction is clocked in. It is cleared automatically in the cycle preceding the one in which the DSR result is written into the register file.

Note that regardless of the value loaded into the DSRINP bit by the load SR instruction, on the next cycle this bit will automatically be updated to indicate whether a DSR operation is in progress. For more details, see section 9.

12.5.4. $SR11_7$: FPEX TAKEN

The FPEX TAKEN bit is asserted if an enabled exception occurs. It is always set two cycles following the one in which the exception was detected. See figures 70 and 71 for timing information. Just exactly when the $SR11_7$ bit is set is immaterial so long as it is set by the time the interrupt handler is ready to use it.

This bit remains set until explicitly cleared by loading SR11 such that $SR11_7 = 0$. Clearing this bit is also the only way to reset the FPEX output, when in sticky mode. Conversely, setting this bit by loading SR11 such that $SR11_7 = 1$ asserts the FPEX output.

This bit is not affected by the state of SR3.

12.5.5. EFFECT OF LOGICAL OPERATIONS ON STATUS REGISTERS

None.

12.5. SR4 Through SR11: Status and Destination Addresses, continued

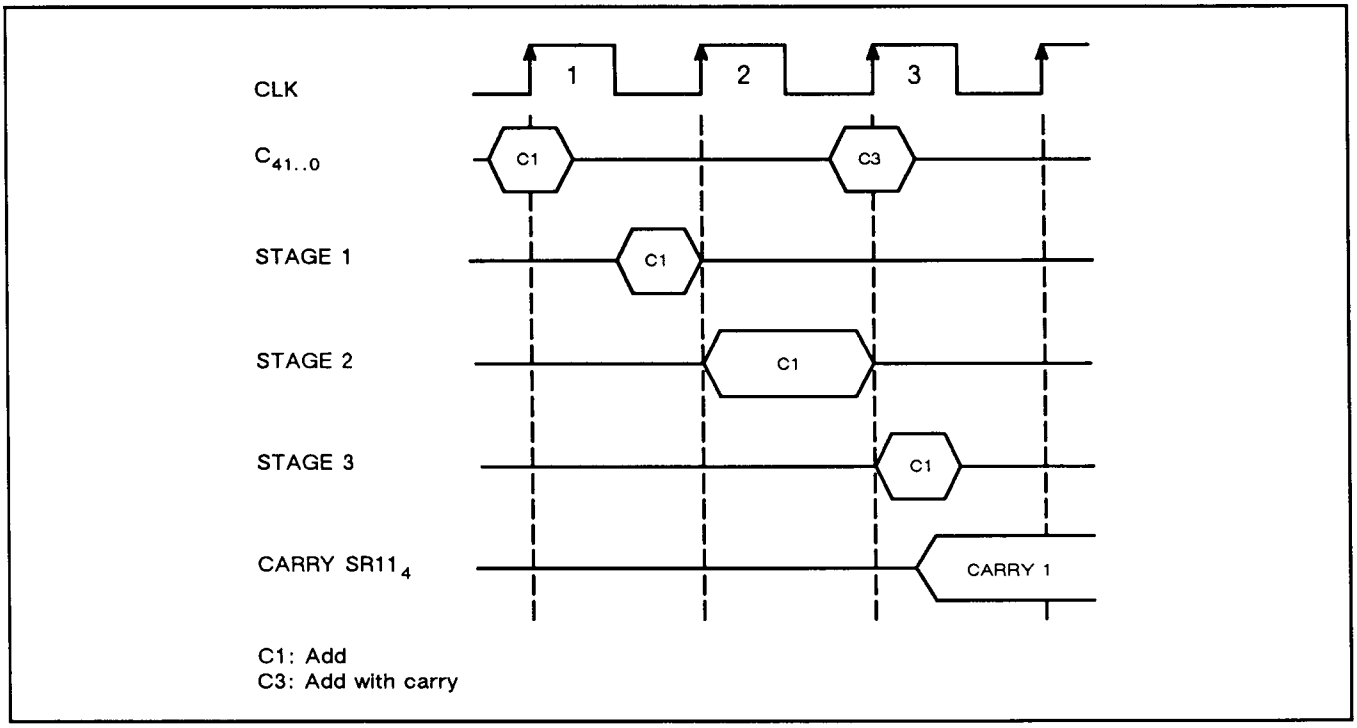


Figure 78. Timing of carry bit update

12.5. SR4 Through SR11: Status and Destination Addresses, continued

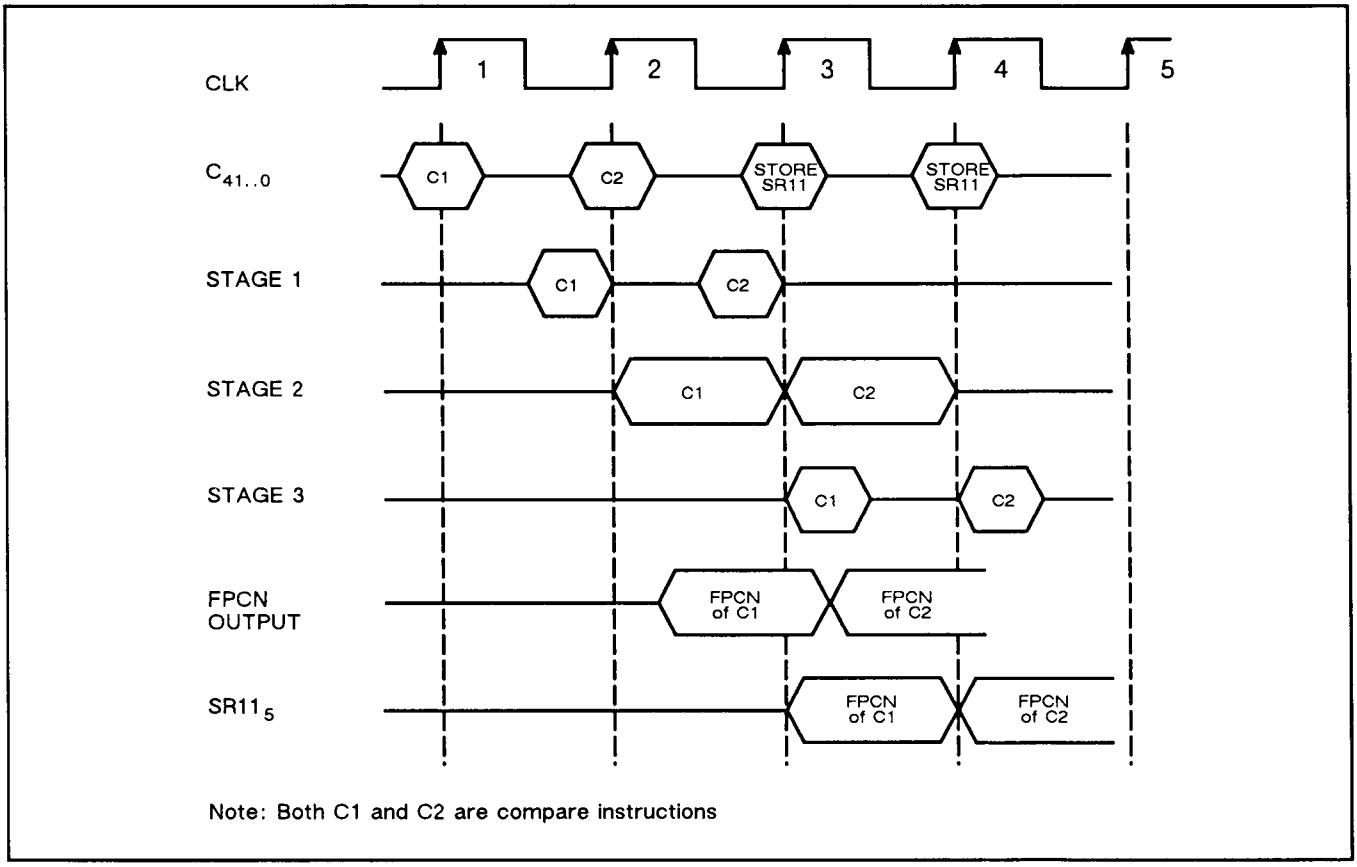


Figure 79. Timing of floating-point condition update

13. NEUT-, STALL-, and ABORT-

NEUT-, STALL-, and ABORT- inputs are provided to "kill" or to eliminate the effects of respectively the current instruction, the next instruction, or both (see Glossary, section 20, for the definition of current and next instructions).

13.1. NEUT- INPUT

When NEUT- is asserted, writing is inhibited to:

- the register file, except for loads in undelayed load mode
- the X and Y registers, except for loads in undelayed load mode
- the temporary latches
- the status register, except for status register loads in undelayed load mode
- the code register, except when there is an exception on the same cycle, see section 14.3

When NEUT- is asserted, outputs corresponding to the instruction that has been neutralized have the following values:

- $S_{3..0}$ has the status of EXT.POS (0100b), see section 11
- FPCN retains its previous value, see section 12.5.2
- in delayed FPEX mode, FPEX it is not asserted
- in undelayed FPEX mode, FPEX output due to source exceptions is erroneously asserted for one cycle (see figure 80) and then de-asserted

Note that assertion of NEUT- does not inhibit any stores, except single-pump delayed-data store. (Stores are inhibited by internally tri-stating output ports.)

13.2. STALL- INPUT

When STALL- is asserted:

- writing to all registers is inhibited
- all stores are inhibited by internally tri-stating output ports

When STALL- is asserted, the outputs corresponding to the instruction that has been stalled have the following values:

- $S_{3..0}$ has the status of EXT.POS (0100b)
- FPCN retains its previous value
- FPEX remains unasserted, regardless of FPEX modes

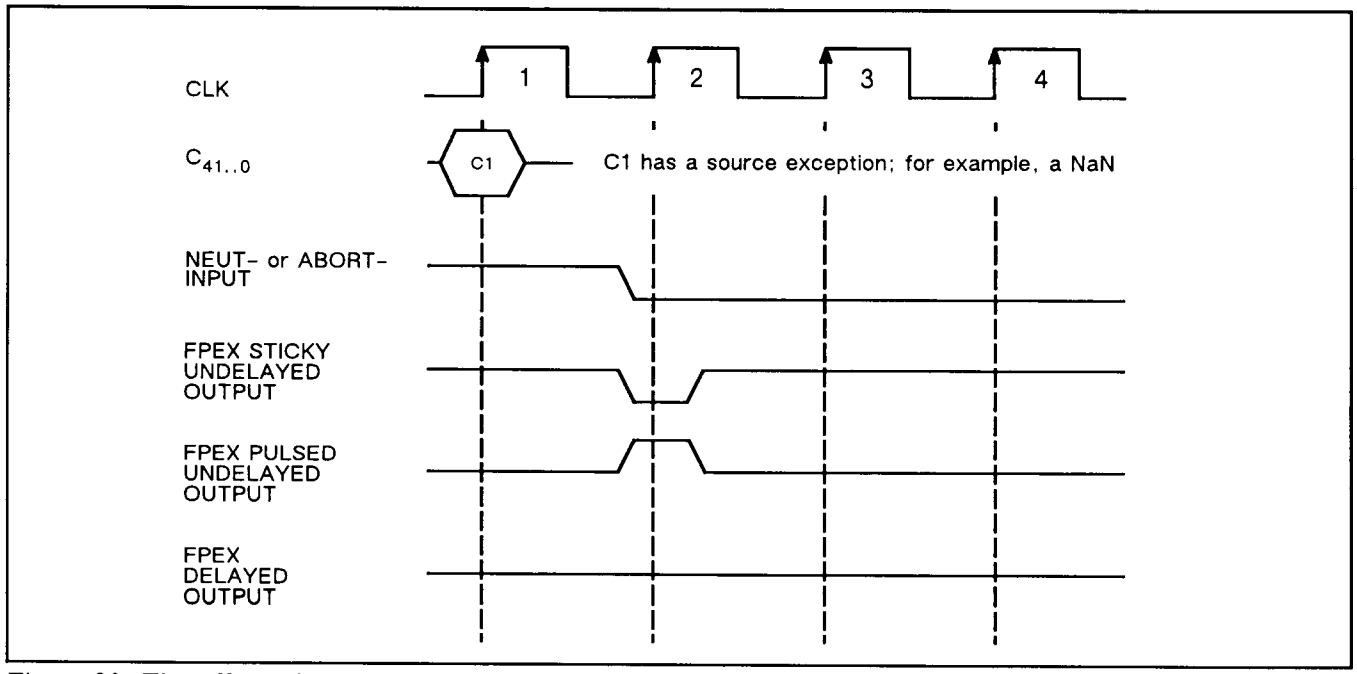


Figure 80. The effect of NEUT- or ABORT- on the current cycle on the FPEX output

13. NEUT-, STALL-, and ABORT-, continued

13.3. ABORT- INPUT

When ABORT- is asserted, its effect on the current instruction is identical to the effect of the NEUT- signal, and its effect on the next instruction is identical to those of the STALL- signal.

Specifically, when ABORT- is asserted, writing is inhibited to:

- the register file, except for current-cycle loads in undelayed load mode
- the X and Y registers, except for current-cycle loads in undelayed load mode
- the temporary latches
- the status register, except for current-cycle status register loads in undelayed load mode
- the code register, except when there is an exception in either the current or the next cycle (see section 14.3)

When ABORT- is asserted, outputs corresponding to the instructions that have been aborted have the following values:

- $S_{3..0}$ has the status of EXT.POS (0100b)
- FPCN retains its previous value
- in delayed FPEX mode, FPEX it is not asserted
- in undelayed FPEX mode, FPEX due to source exception on the current-cycle instruction is erroneously asserted for one cycle (see figure 80) and then de-asserted

Note that assertion of ABORT- does not inhibit current-cycle single-pump undelayed or delayed stores or double-pump delayed-data stores.

13.4. TIME-PUSHED PIPELINES

None of the NEUT-, STALL-, or ABORT- signals stops the pipeline, that is they do not internally stop CLK or DIVCLK. The 3x64 has a time-pushed, as opposed data-pushed, pipeline. When STALL-, NEUT-, or ABORT- is asserted, the pipelines continue to advance. Using figure 81 as an example, instructions C1 (a 64-bit divide) and C2 (a two-cycle latency multiplication) do complete and do update the register file and the status register even though NEUT-, STALL-, or ABORT- was asserted on cycle 3, prior to the completion of instructions C1 and C2. For details on the effects of STALL-, NEUT-, and ABORT- on the code register see section 14.3.

13.5. INTERNAL NEUT SIGNAL

The INTERNAL NEUT signal is asserted internally if it is enabled ($SR0_3 = 1$, see section 12.3.3) and an enabled exception occurs. The INTERNAL NEUT signal has the same effect as the NEUT- input except since INTERNAL NEUT signal is a consequence of a floating-point exception, it does not:

- inhibit any I/O (load/store) operations
- inhibit updating of source exception status bits $SR3_{7..4}$ (NaN, INV, DVZ, DNRM, see section 12.4)
- affect FPEX output or FPEX TAKEN status bit ($SR11_7$, see section 12.5.4)
- inhibit writing of the code register

13. NEUT–, STALL–, and ABORT–, continued

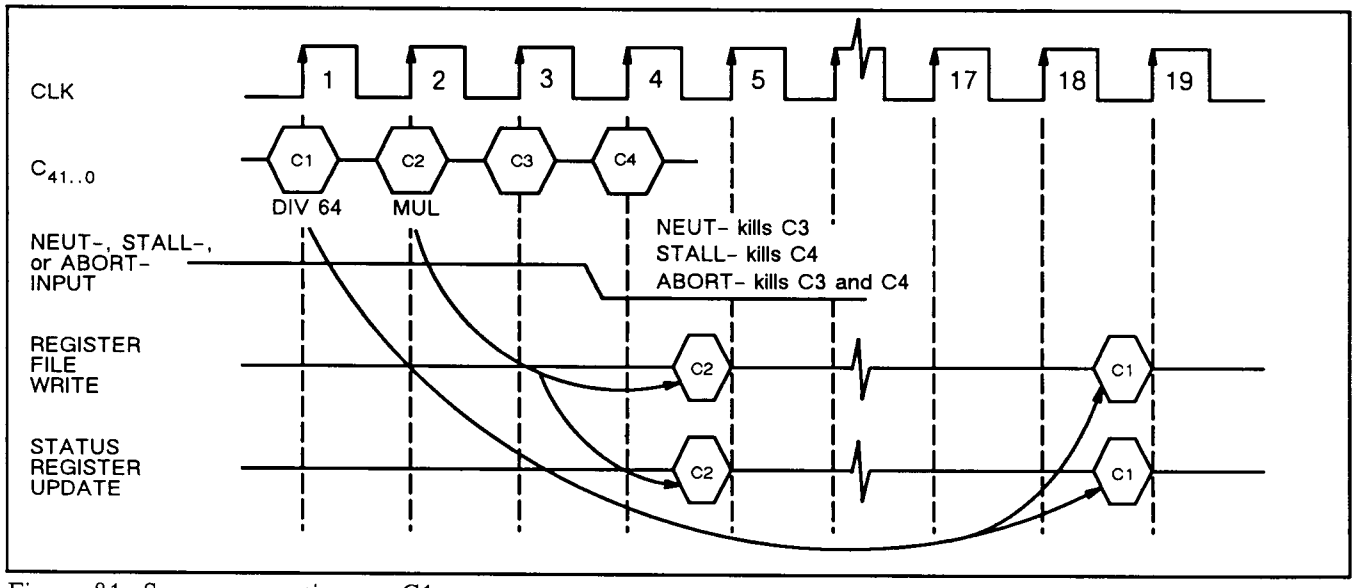


Figure 81. Source exception on C1

14. Code Register

The purpose of the code register is to "remember" the last instruction issued before an enabled exception occurred, so that it can be re-executed as part of an interrupt service routine that would deal with the cause of the interrupt. See figure 82.

14.1. Organization

The 42-bit code register consists of five 8-bit registers and one 2-bit register. See figure 83. The re-execute decoder controls the multiplexer that selects the instruction to be executed: the one on the code input or the current contents of the code register. There is an opcode for the re-execute function, see section 17.15.2.

14.2. Operation

The code register is written on every cycle unless:

- STALL–, ABORT–, or NEUT– has been asserted. See sections 14.3 and 15
- the instruction is a load/store code/status register or re-execute instruction
- FPEX TAKEN bit is set ($SR11_7 = 1$)

14.2. Operation, continued

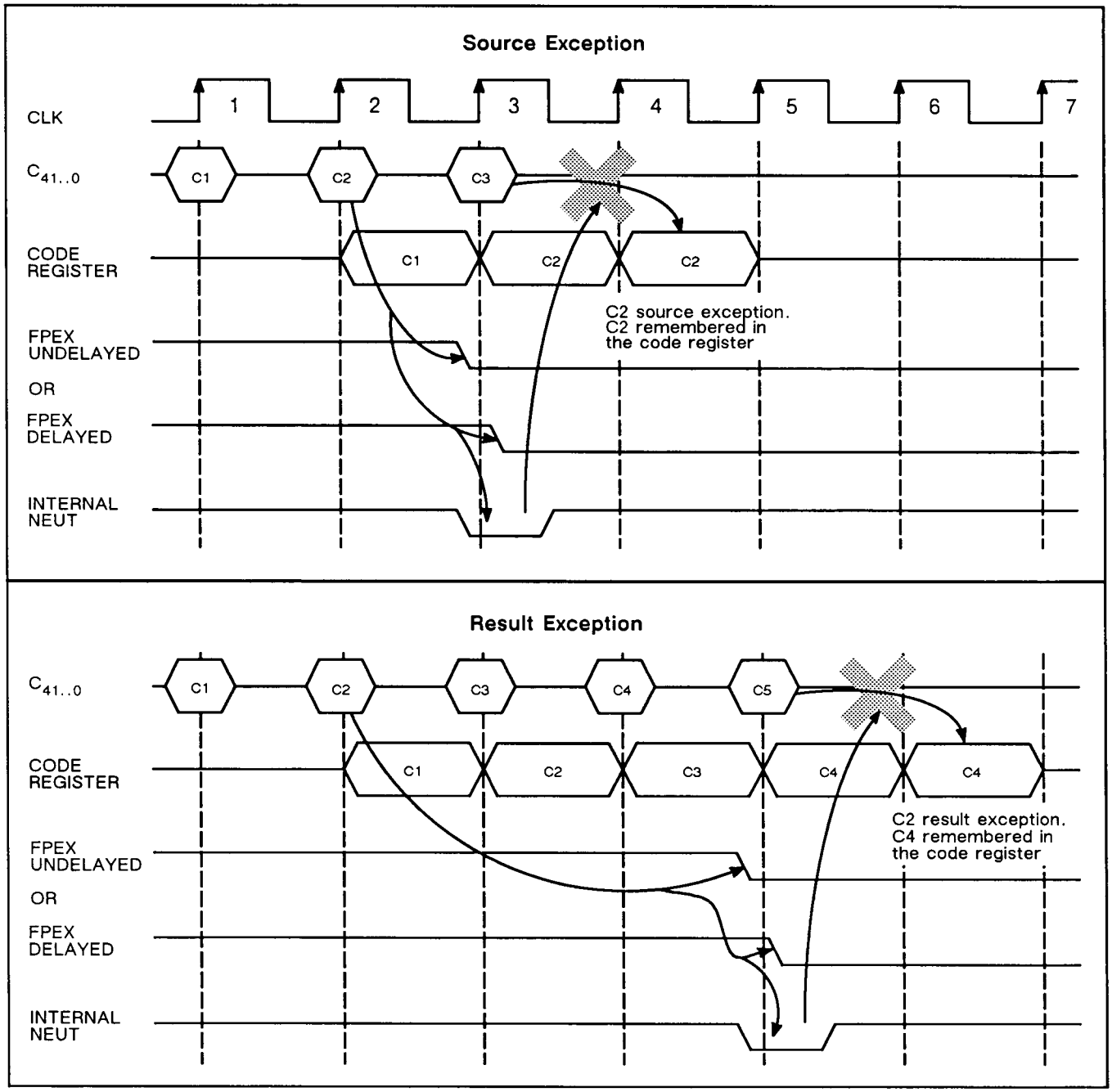


Figure 82. The purpose of the code register is to remember the last instruction clocked in before an exception occurred (INTERNAL NEUT is on)

14.2. Operation, continued

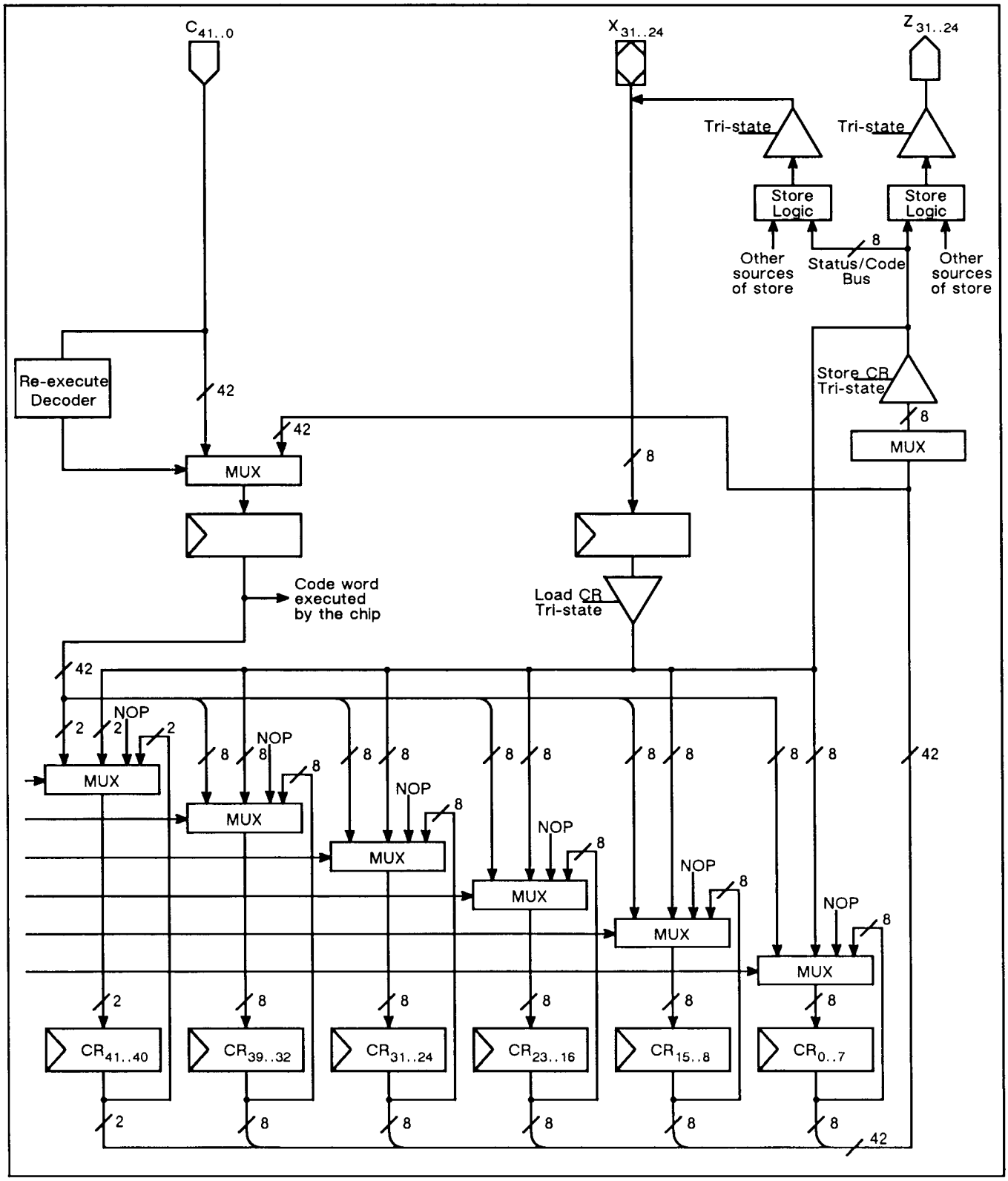


Figure 83. Code register

14.3. The Effects of STALL-, NEUT-, and ABORT- on the Code Register

Internally, the loading of the code register with the instruction to be remembered is delayed by one cycle relative to the clocking-in of this instruction at the code port. This provides the ability to automatically replace the code register with a NOP if the instruction to be remembered happens to be stalled, neutralized, or aborted.

14.3.1. SOURCE EXCEPTIONS

If it turns out that there is a STALL- or ABORT- input at the rising edge of the instruction that would cause a source exception, or if there is an ABORT- or NEUT- input at the next rising edge, then the code register retains its previous value and the FPEX output remains unasserted. See figure 84. The reason for this behavior is that the purpose of these control inputs is to "kill" C2. This has two implications. First, the fact that C2 may have had any source exceptions is irrelevant; therefore FPEX remains unasserted. Second, since C2 was not meant to be executed in the first place, it is not meant to be re-executed, and therefore need not be remembered in the code register.

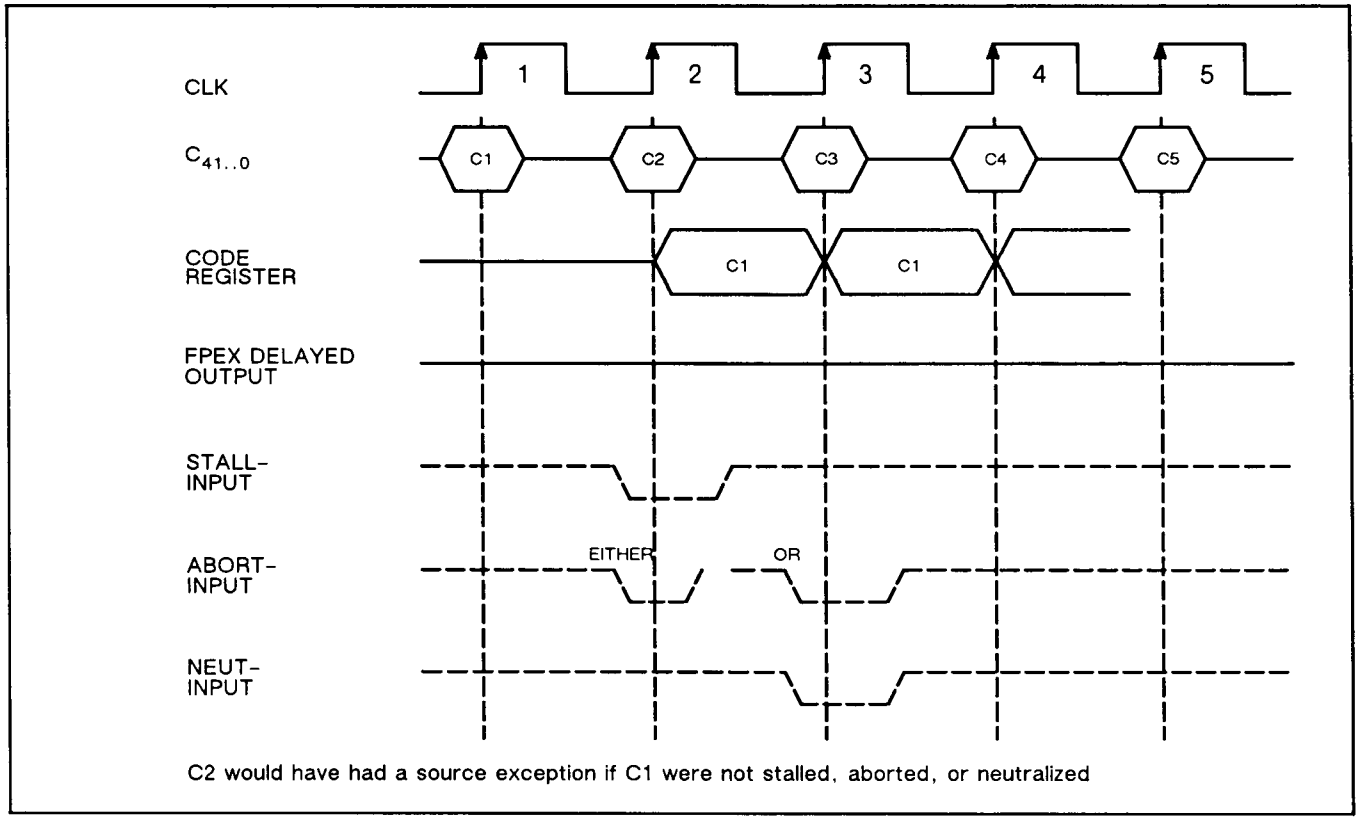


Figure 84. Source exception on C2 in conjunction with one of the STALL-, ABORT-, or NEUT- inputs

14.3. The Effect of STALL-, NEUT-, and ABORT- on the Code Register, continued

14.3.2. RESULT EXCEPTIONS

Refer to figure 85. If it turns out that there is a STALL- or ABORT- input at the rising edge of cycle 3, or a NEUT- or ABORT- at the rising edge of cycle 4, then the code register gets written with a nop. FPEX output, however, is asserted, because in this case the control inputs are meant to "kill" instruction C3, and FPEX output is the result of an exception on C1. The destination address and status of the result of C1 have been saved in the status register to be "fixed up" (in a trap handler); when the handler re-executes the instruction in the code register, it gets to re-execute a nop).

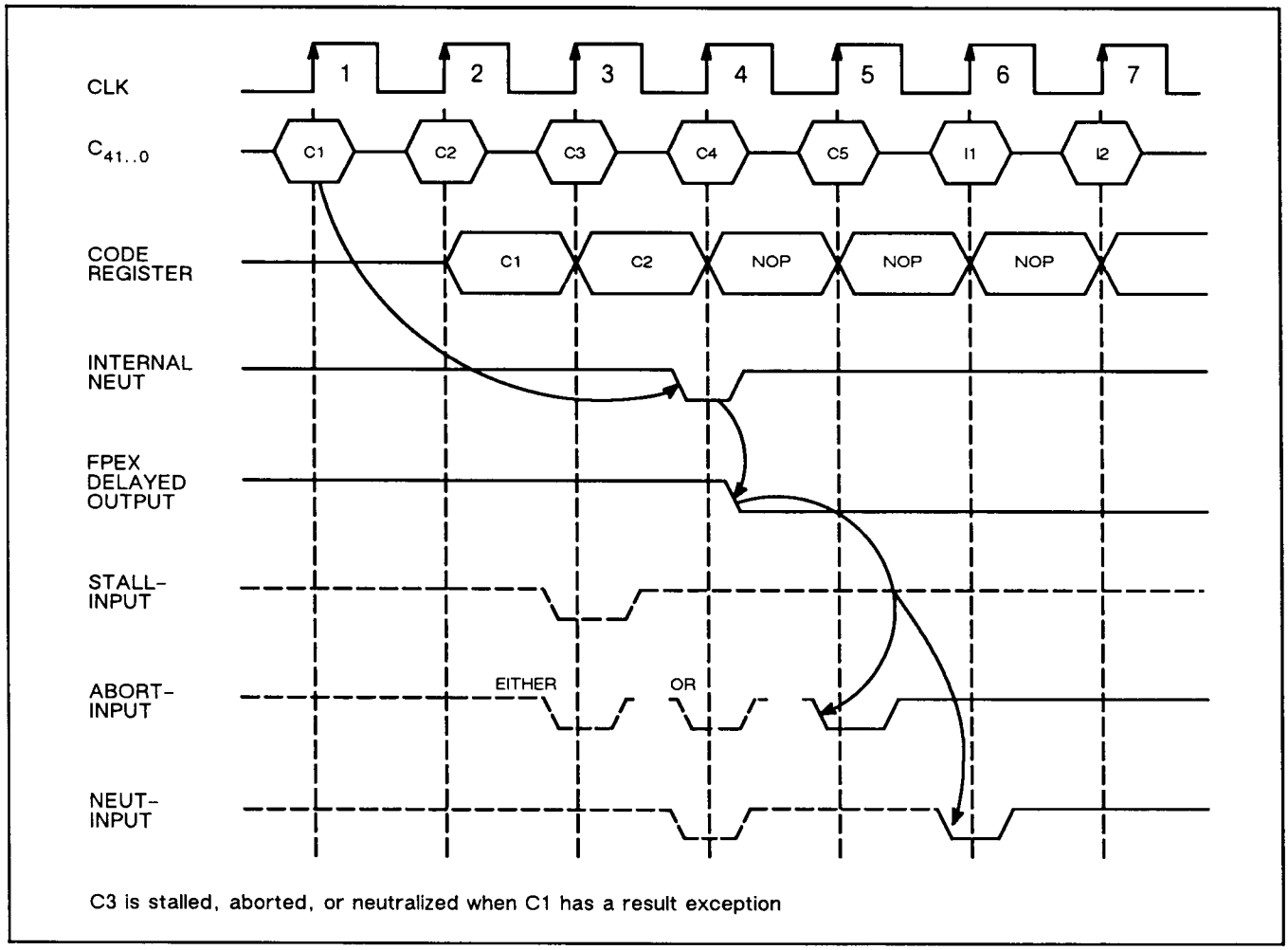


Figure 85. Result exception on C1 in conjunction with one of the STALL-, ABORT-, or NEUT- inputs

14.4. The Effect of FPEX TAKEN on the Code Register

Once FPEX TAKEN bit ($SR11_7$) is set by an exception, further INTERNAL NEUTs and writing to the code register are inhibited, until this bit is cleared. See figure 86. Subsequent exceptions — and there can be one more in the multiplier, one more in the ALU, and one in the divide/square root unit — do set sticky bits and update the status register with destination addresses and status information, just as the instruction that caused the exception, but they do not update the code register or generate INTERNAL NEUT. The relevant state information should be saved before resetting the FPEX TAKEN bit.

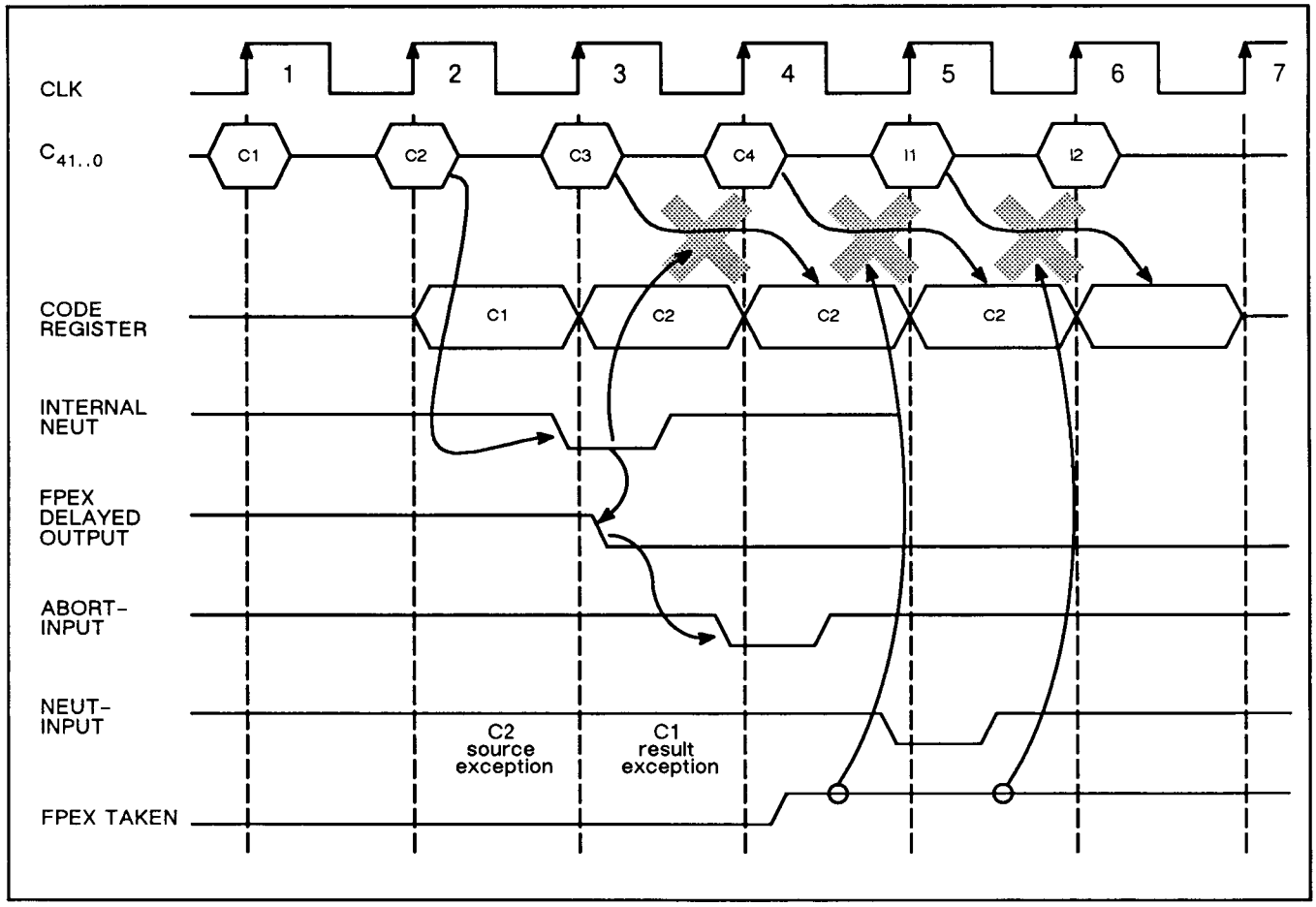


Figure 86. Several exceptions in a row

14.5. Code Register Load/Stores

Code register loads occur through the X port. The load code register instruction, like other loads, supersedes any other attempts to write to the code register. The timing of the load instruction corresponds to whatever load mode is in effect provided that the data is presented on the rising edge of the clock.

Stores occur simultaneously through both the X and the Z ports. The timing of the store code register instruction corresponds to whatever store mode is in effect.

For code register loads, the most-significant byte of the X port is used, $X_{31..24}$. The other 24 bits on that port must be set to zero. The contents of the code register are stored simultaneously through the most-significant byte of the X port ($X_{31..24}$) and Z port ($Z_{31..24}$), and the other bytes are set to zero. The code register is loaded and stored in 8-bit bytes. These are physical bytes, not logical bytes shown in the code word format in figures 4, 95, and 96 — see the next section. When loading the most-significant byte, its six most-significant bits are "don't cares"; when storing it, these bits are set to zero. The code register and status register load/store instructions are identical in timing, see figure 61. These instructions are detailed in section 17.16.7. Note that on code register loads and stores, bits $C_{12..0}$ (XCNT, YCNT, ZCNT) must be set to zero.

14.5.1. CODE WORD FORMAT AND THE PHYSICAL CODE REGISTER BITS

In this document, the term "code word" is used in the sense shown in figures 4, 95, and 96. These figures show that each multibit field is logically contiguous. Internally, however, the physical code register bits corresponding to some of these fields are not contiguous. Some applications — primarily IEEE trap handling routines — need to manipulate the contents of the code register using code register load/store instructions. Since these instructions operate on physical code register bytes, it is important to know the correspondence between the physical bits and bytes of the code register and the logical format of the code word, as well as device pins. Figure 87 provides this information.

14.5. Code Register Load/Stores, continued

Code register byte	Code register bit	Bits of the X and Z ports	Pin name	Code word bit
0	0	24	EFADD0	C8
	1	25	EFADD1	C9
	2	26	EFADD2	C10
	3	27	EFADD3	C11
	4	28	EFADD4	C12
	5	29	DADD0	C13
	6	30	DADD1	C14
	7	31	DADD2	C15
1	8	24	DADD3	C16
	9	25	DADD4	C17
	10	26	AADD0	C28
	11	27	AADD1	C29
	12	28	AADD2	C30
	13	29	AADD3	C31
	14	30	AADD4	C32
	15	31	CADD0	C18
2	16	24	CADD1	C19
	17	25	CADD2	C20
	18	26	CADD3	C21
	19	27	CADD4	C22
	20	28	FUNCT0	C37
	21	29	XCNT1	C5
	22	30	XCNT2	C6
	23	31	XCNT3	C7
3	24	24	XCNT0	C4
	25	25	ZCNT1*	C1
	26	26	YCNT0*	C2
	27	27	YCNT1*	C3
	28	28	FUNCT1	C38
	29	29	FUNCT2	C39
	30	30	FUNCT3	C40
	31	31	FUNCT4	C41
4	32	24	BADD0	C23
	33	25	BADD1	C24
	34	26	BADD2	C25
	35	27	BADD3	C26
	36	28	BADD4	C27
	37	29	ZCNT0*	C0
	38	30	ABIN	C35
	39	31	AAIN	C36
5	40	24	MBIN	C33
	41	25	MAIN	C34

* These pins are present in the 3364, but not in the 3164. In the 3164, they must be loaded as zero, otherwise unpredictabe results will be produced.

Figure 87. Correspondence between code word pins and code register bits

15. Exception Handling

The 3x64, in conjunction with appropriate software, provides the necessary functionality to handle the full range of IEEE exceptions in pipelined environment.

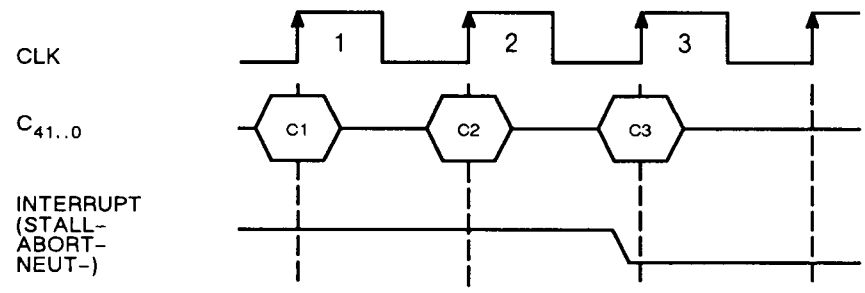
When exceptions occur it is necessary to stop the system "in time." In time means that all the information necessary to "fix up" the exception (operand addresses on source exceptions and result addresses and status on result exceptions) is available, that is, it has not been overwritten as a result of subsequent instructions.

The 3x64 is designed to handle exceptions in two types of systems: systems that can back up and re-execute an instruction, and systems that cannot. In terms of figure 88, a system can back up if both the current instruction (C2) and the next instruction (C3) can be re-executed. A system cannot back up if only the next instruction (C3) can be re-executed.

Systems that can back up can be stopped one cycle later relative to systems that cannot. These systems can therefore use delayed FPEX, which yields a faster cycle time. Exception handling with delayed FPEX is discussed in section 15.1.

In order to be stopped in time, systems that cannot back up must use undelayed FPEX, which results in longer cycle time. Exception handling with undelayed FPEX is discussed in section 15.2.

In this section, it is assumed that the INTERNAL NEUT mode bit is on ($SR0_3$ = 1) and that a floating-point instruction is initiated on every cycle.



Systems that can back up: both C2 and C3 can be seamlessly re-executed. Delayed FPEX.

Systems that cannot back up: only C3 can be re-executed. Undelayed FPEX.

Figure 88. System types

15.1. Delayed FPEX ($SR1_5$ = 1)

Systems that can back up use this mode, which yields a faster cycle time.

15.1.1. SOURCE EXCEPTIONS

If the INTERNAL NEUT mode bit is on and an enabled source exception occurs on instruction C1 (see figure 89), the INTERNAL NEUT signal inhibits writing of the code register from code port, so that C1 remains in the code register.

The detection of the exception also sets the FPEX TAKEN bit in the status register ($SR11_7$) and asserts the FPEX output pin on the next cycle.

The FPEX output should be ORed (for one cycle only) into a signal that can back up the system by one cycle. In this case, the ABORT- input is an OR of the FPEX output and all other causes of abort. This function is implemented externally. The ABORT- input eliminates the effects of the current and the next instructions; C2 and C3, respectively. Figure 89 also shows how to use the re-execute code register instruction to restore normal code flow following an interrupt.

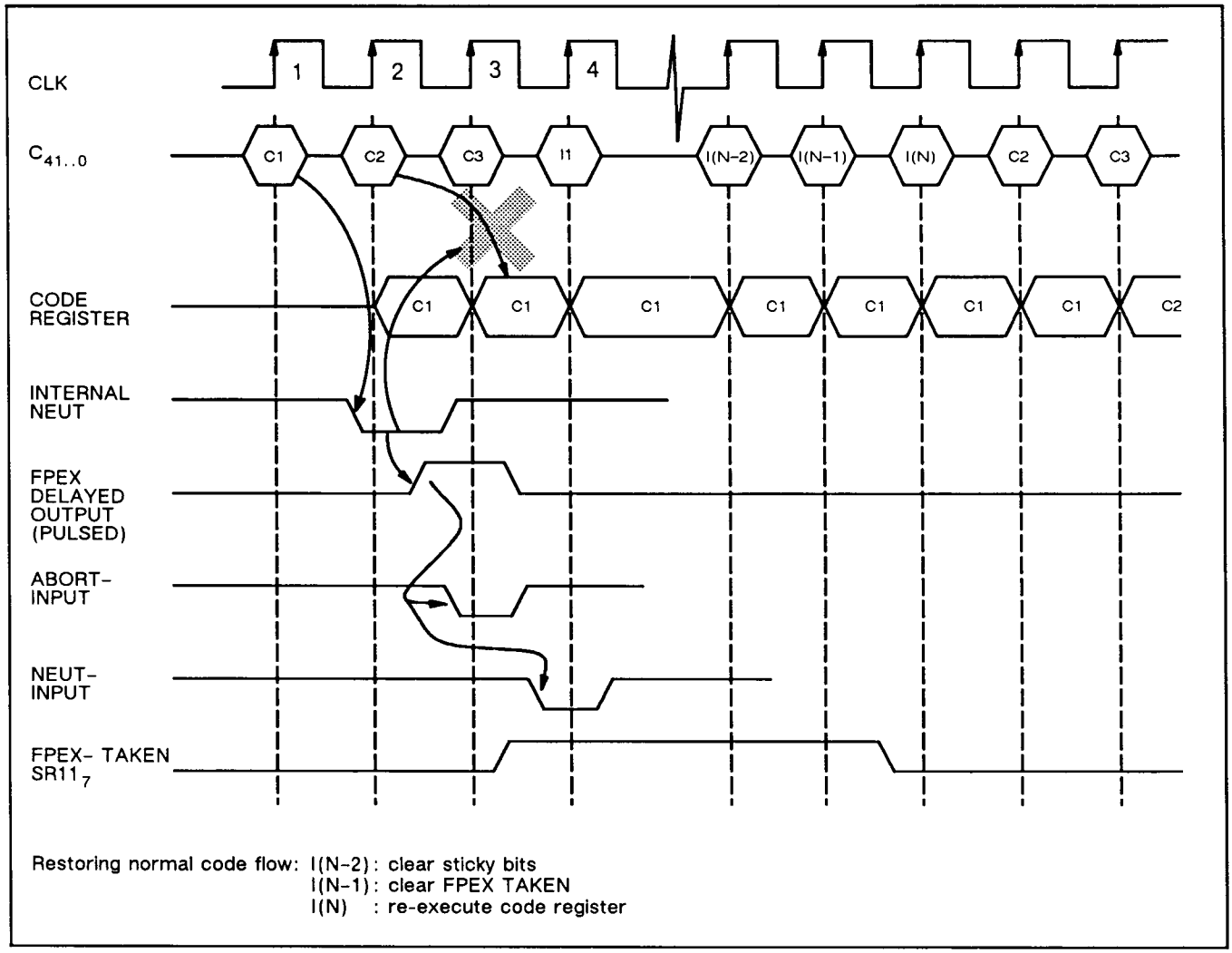


Figure 89. Source exception on C1

15.1. Delayed FPEX (SR1$_5$ = 1), continued

15.1.2. RESULT EXCEPTIONS

This case is similar to the source exception case, with INTERNAL NEUT signal generated at the time of the exception, in cycle 3. See figure 90.

In this case, the exception information is stored in the 3x64 status registers, allowing the handler to correct the exception on C1 and any potential exception on C2.

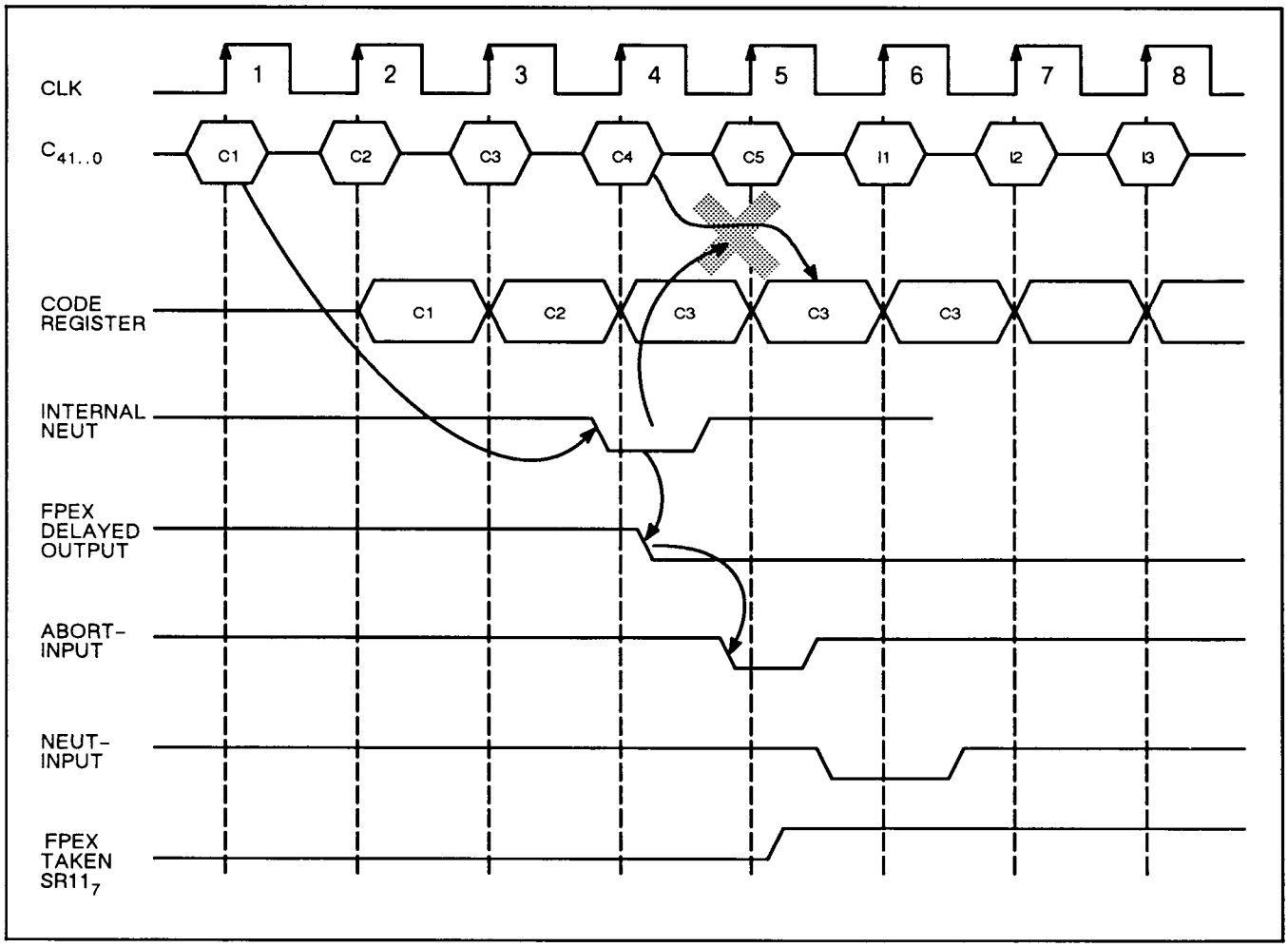


Figure 90. Result exception on C1

15.2. Undelayed FPEX ($SR1_5$ = 0)

Systems that cannot back up use this mode.

In this mode, when an enabled exception is detected, the FPEX output is asserted on the same cycle. In other respects, this case is similar to the delayed FPEX case.

Since FPEX output has to be generated on the cycle of the exception, the cycle will be longer than in the delayed FPEX case; it will be determined by FPEX output delay, set-up time, and propagation delay of an outside device to which FPEX is an input. The FPEX output should be used to generate the STALL- input by ORing it with other causes of stall. See figures 91 and 92. The STALL- input will "kill" subsequent instructions but it cannot back up the system.

Undelayed FPEX results in simpler design and it does not require a sequencer that can back up. This is achieved at the cost of potentially longer cycle time, compared to the delayed FPEX case.

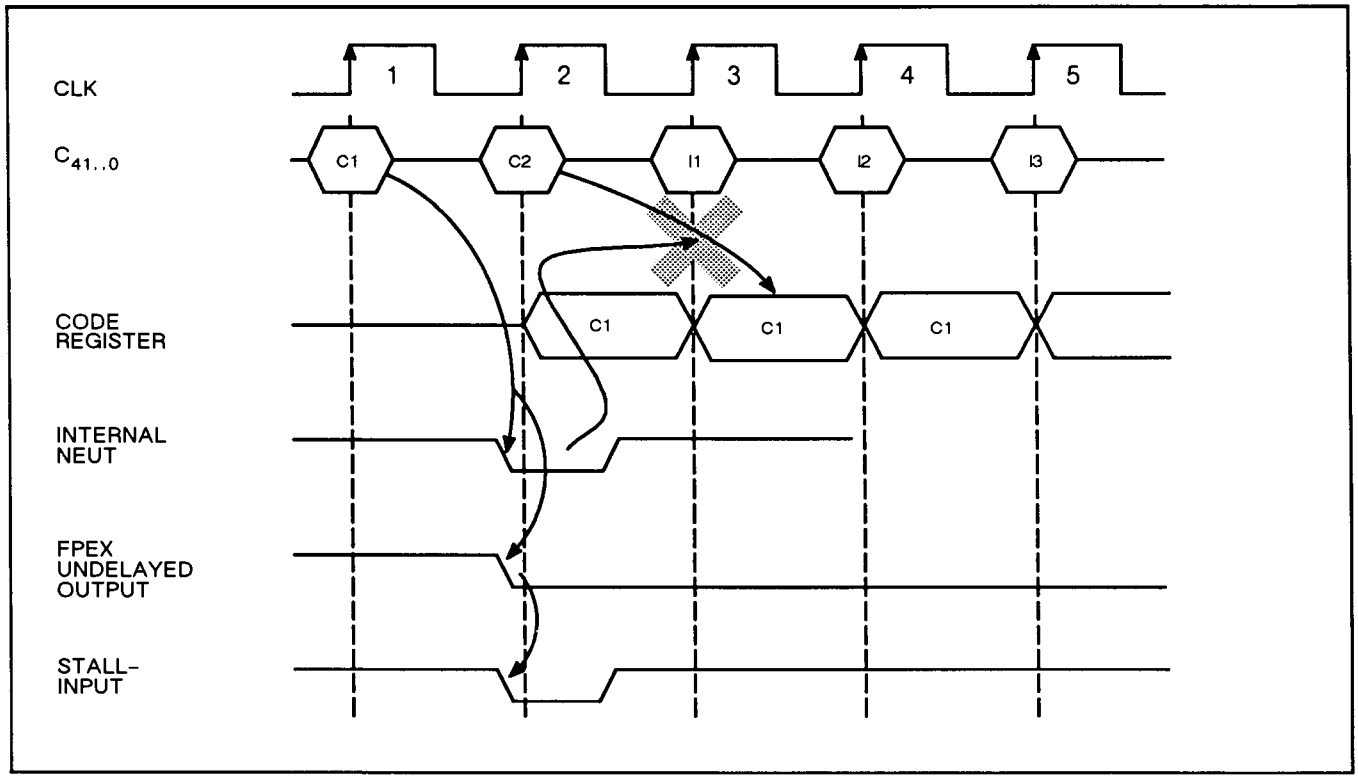


Figure 91. Source exception on C1, undelayed FPEX

15.2. Undelayed FPEX ($SR1_5$ = 0), continued

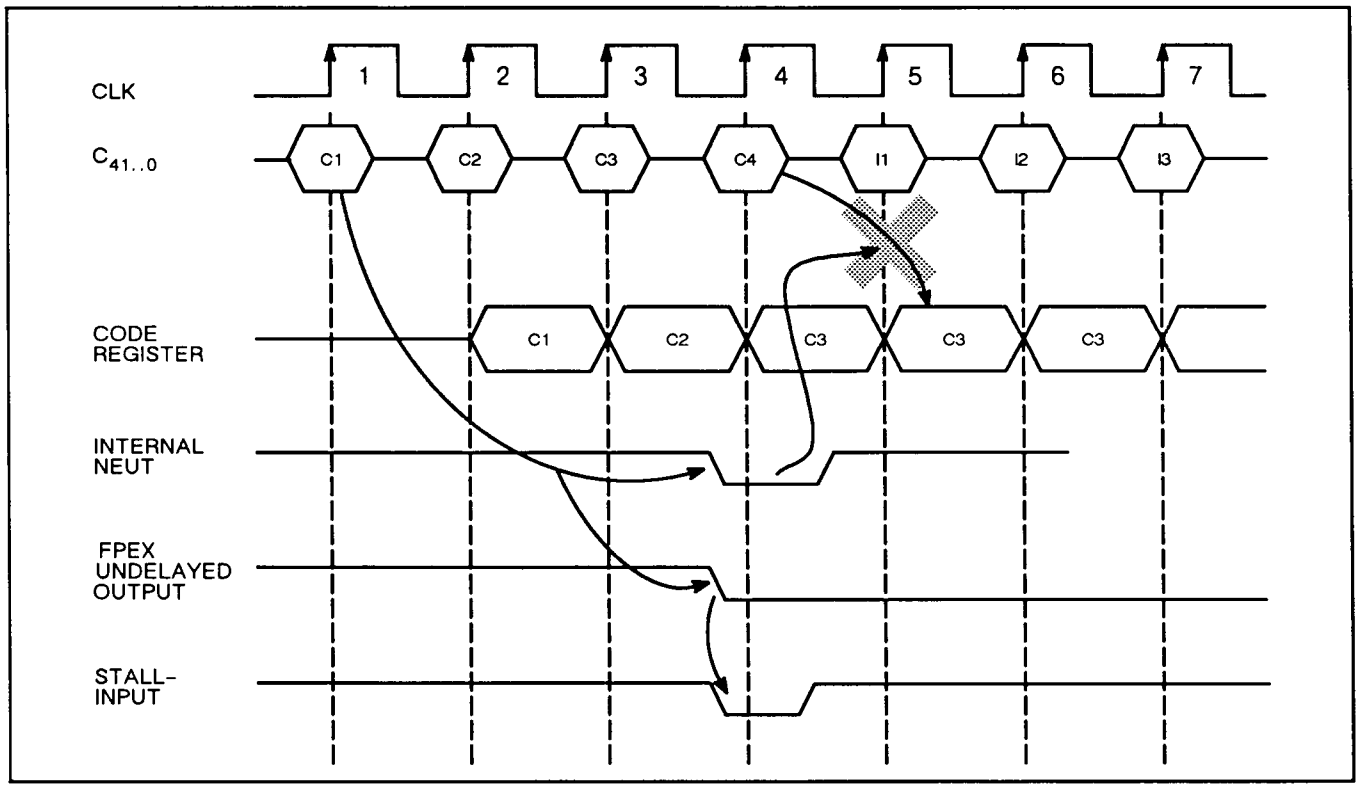


Figure 92. Result exception on C1, undelayed FPEX

16. Interruptibility and Disallowed Code Sequences

The definition of interruptibility is the ability to suspend the execution of the current program, execute a sequence of unrelated instructions (known as an interrupt handler), and to seamlessly restore execution of the current program without disturbing the current functioning of the program. Because the 3x64 uses a time-pushed pipeline, an interrupt will cause the pipeline to drain, rendering certain code sequences uninterruptible.

Certain code sequences written for the 3x64 may produce wrong results and are therefore disallowed. Other code sequences are legitimate but may be uninterruptible in the general sense. Still other code sequences may be interruptible but not necessarily IEEE-interruptible. This section deals with all three cases (sections 16.2–16.4). In the examples, it is assumed that bypassing is enabled; OP denotes any two-cycle-latency instruction. All the instructions are numbered, and it is assumed that instruction N + 1 follows instruction N on the next cycle.

16.1. Definitions of Interruptibility

If there are two instructions, with perhaps other instructions between them, and the same result is obtained whether or not there was one or more arbitrary-length interrupts between them, then the code between the two instructions (inclusive of these instructions) is interruptible. If the result depends on whether there was an interrupt, the code is uninterruptible.

For IEEE interruptibility, there is a further requirement of being able to "fix up" exceptions, which means that for source exceptions it is necessary to have all the operands, and for result exceptions it is necessary to have the result that produced the exception and its status. In the 3x64, it is possible to have up to five result exceptions. All five result destinations and the associated status are saved in the status register. See section 12 for details.

16.2. Disallowed Code Sequences

16.2.1. CONFLICTING USE OF THE SAME HARDWARE

When two successive instructions require the use of the same hardware, both instructions produce unpredictable results.

Example 1

1 R1 × R2 → R3
2 R4 × R6 → R7

If the first multiply is an integer or a double-precision floating-point multiply in three-cycle latency mode, it requires two passes through the second stage each of which takes a cycle. Since the second multiply will attempt to use the second stage of the multiplier while it is being used by the previous multiply, the second instruction will be ignored, producing unpredictable results. See section 7.3 and figure 49.

Example 2

1 Divide or SQRT → R1
2 Divide or SQRT → R2

When the DSR unit is occupied with an instruction, another instruction may not be initiated until the prior one has completed. If this condition is violated, the second instruction will be ignored, producing unpredictable results. See section 9.1 and figure 50 for more details.

Both the code register and the status register use the same 8-bit internal bus for loads and store. (This bus is identified in figure 83 as status/code bus.) Any code which results in an attempt to use this bus simultaneously for loads and stores is illegal, as in the following examples:

Example 3

1	Load SR1	Delayed load
2	Store SR4	Any store mode

Example 4

1	Load CR2	Delayed load
2	Store CR4	Any store mode

Example 5

1	Load SR1	Delayed load
2	Store CR2	Any store mode

On all stores, the status/code bus is used on the cycle which clocks in the store instruction. This is true even in the case of delayed stores or delayed-data stores: the bus is used during the same cycle and the actual delay is done at the output pins. Thus in these examples the load and the following store instructions will attempt to use the status/code bus at the same time, which is illegal. Since loads override stores, in this case a store will produce wrong data.

16.2. Disallowed Code Sequences, continued

16.2.2. BYPASSING ACROSS DIFFERENT DATA TYPES

Bypassing across different data types can occur in register-to-register operations. For example, if instruction N is a *double-precision* floating-point multiply, and an attempt is made to use the result of this instruction as a *single-precision* floating-point operand in instruction N + 2 (N + 3 in three-cycle latency mode), bypassing across data types (in this case, double- to single-) has occurred. The results produced in such a situation are, in general, unpredictable.

Since there are four different data types supported in the 3x64, there are 16 possible bypass combinations. The following table lists them and specifies whether or not they are legal. Illegal combinations produce unpredictable results.

Example 1

1	R1 OP R2 → R3	Single-precision floating-point
2	NOP	
3	R3 + R4 → R5	Integer

In this example, an attempt is made to use a single-precision floating-point result of instruction 1 as an integer operand of instruction 3, which is illegal.

Abbreviation	Data Type
L	64-bit logical
S	Single-precision floating-point
D	Double-precision floating-point
I	32-bit integer

Data Type Crossing	Legality
L → L	OK
L → D	Illegal
L → S	Illegal
L → I	Illegal
D → D	OK
D → L	OK
D → S	Illegal
D → I	Illegal
I → I	OK
I → L	OK
I → D	Illegal
I → S	Illegal
S → S	OK
S → D	Illegal
S → L	Illegal
S → I	Illegal

Figure 93.

16.3. Allowed But Uninterruptible Code

16.3.1. USE OF RESERVED OPCODES

The use of reserved opcodes is illegal as they are not guaranteed to produce predictable results.

This section contains examples to illustrate code, which, though legal, would be uninterruptible. In the absence of an interrupt, old values of registers are used. If an interrupt occurs, new values would be used, giving a different answer. Therefore, such code is uninterruptible.

16.3.2. DATA NOT READY

Example 1

```
1       R1 OP R2 → R3
2       R3 OP R4 → R5
```

Instruction 2 specifies R3 which would not be ready until cycle 3. This code is uninterruptible.

Example 2

```
1       R3 × R4 → R5    Double-precision floating-
                        point or integer multiply
                        (three-cycle latency mode)
2       R5 OP R6 → R7
```

The result of the multiply will not be ready until cycle 4, yet an attempt is made to use it in cycle 2. This code is uninterruptible.

Example 3

```
1       R3 × R4 → R5    Double-precision
                        floating-point or integer
                        multiply (three-cycle
                        latency mode)
2       R1 OP R2 → R3
3       R5 OP R6 → R7
```

The result of the multiply will not be ready until cycle 4, yet an attempt is made to use it in cycle 3. This code is uninterruptible.

Example 4

```
1       R1 ÷ R3 → R4
⋮
N       R4 OP R5 → R6
```

If an instruction tries to use R4 before the divide (or square root) operation has completed writing into R4, this code would be uninterruptible. See section 9.

Example 5

```
1       R2 OP R4 → R6
2       Store R6        Any store mode
```

The result of the operation will not be written into the register file until the second half of cycle 3. The result of the operation will not be available for any store instruction issued on cycle 2. Therefore, this code is uninterruptible.

Example 6

```
1       R2 OP R4 → R6
2       R1 OP R7 → R8
3       Store R6        Delayed store
```

Operation in cycle 1 will write R6 in the second half of cycle 3. Delayed store in cycle 3 will not bypass, so it will drive the old value onto the output port. Thus, this code is uninterruptible.

Example 7

Storing the status register before all operations in the multiplier, DSR unit, or ALU have completed will, in general, result in uninterruptible code.

Example 8

```
1       Load CR0
2       Load CR1
3       Load CR2
4       Load CR3
5       Load CR4
6       Load CR5
7       Re-execute CR
```

Example 9

```
1       Clear FPEX TAKEN
2       Re-execute CR
```

When FPEX TAKEN is set, incoming instructions do not update the code register. Clearing the FPEX TAKEN bit allows updating of the code register. If an interrupt occurs, the code register will be updated.

16.3. Allowed But Uninterruptible Code, continued

16.3.3. DESTINATION CONFLICT

A destination conflict arises whenever a long-latency instruction specifies a destination register that is also specified as a destination of a following shorter-latency instruction that would complete before the first instruction. The code in these examples is uninterruptible because the destination register will contain two different results depending on whether there was an interrupt.

Example 1

1	R2 OP R3 → R16	Includes integer or double-precision floating-point multiply in three-cycle latency mode
2	Load R16	Undelayed load

Example 2

1	R1 OP R4 → R16	Includes integer or double-precision floating-point multiply in three-cycle latency mode
2	R6 OP R7 → R8	
3	Load R16	Undelayed load

Example 3

1	R1 × R4 → R16	Double-precision floating-point or integer multiply (three-cycle latency mode)
2	R6 OP R7 → R8	
3	R6 OP R7 → R9	
4	Load R16	Undelayed load

Example 4

1	R2 OP R3 → R16	Includes integer or double-precision floating-point multiply in three-cycle latency mode
2	Load R16	Delayed load

Example 5

1	R2 OP R3 → R16	Includes integer or double-precision floating-point multiply in three-cycle latency mode
2	R6 OP R7 → R8	
3	Load R16	Delayed load

Example 6

1	R1 × R3 → R5	Integer or double-precision floating-point multiply in three-cycle latency mode
2	R2 OP R4 → R5	

Example 7

1	R1 ÷ R2 → R3	
.		
.		
.		
N	R1 OP R6 → R3	If this instruction completes before the divide its result will be overwritten by the subsequent divide result. Likewise for square root.

16.4. Generally Interruptible but not IEEE-Interruptible Code

Code which is interruptible in the general sense but is not IEEE-interruptible arises principally on loads and when chained multiply-add operations are used without regard to IEEE interruptibility. These cases are illustrated in examples below.

Example 1

```
1       Load R1; R1 OP R2 → R3      Delayed load
```

If in the operation R1 has a source exception it cannot be "fixed up" because R1 gets overwritten by the load.

Example 2

```
1       R1 OP R2 → R5
2       R3 OP R4 → R5
3       R6 OP R7 → R8
```

This is a case of not-very-useful code, and it should be avoided. It is included here because it may nevertheless be encountered.

This code is interruptible only if R5 from the first instruction is not going to be used on cycle 3, which effectively makes the first instruction a NOP. If the first instruction generates a result exception, by the time an exception handling routine is ready to use the destination address and associated status, this information will have been overwritten by the second instruction. To make this code IEEE-interruptible, either use different destination addresses or interleave another instruction between the two shown.

This is the same example that was used to calculate the sum of four products in section 10. What is different here is that the same register R30 is used in conjunction with T-latches. Since the T-latches cannot be read directly, the results placed in T-latches must be read from the corresponding register in the register file. Unless these registers are different, it will not be possible to fix up a T-latch result exception because it would be overwritten. Thus, to assure IEEE interruptibility, alternate file registers must be used in conjunction with alternate T-latches.

Example 3

1	$X_1 \times B_1 \rightarrow R3$;	
2	$X_2 \times B_2 \rightarrow R4$;	
3	$X_3 \times B_3 \rightarrow T0$, R30; R29 + T0 → R29;	R3 has $X_1 \times B_1$
4	$X_4 \times B_4 \rightarrow T1$, R30; R29 + T1 → R29;	R4 has $X_2 \times B_2$
5	$X_5 \times B_5 \rightarrow T0$, R30; R3 + T0 → R3;	
6	$X_6 \times B_6 \rightarrow T1$, R30; R4 + T1 → R4;	
7	$X_7 \times B_7 \rightarrow T1$, R30; R3 + T0 → R3;	R3 has $X_1 \times B_1 + X_3 \times B_3$
8	$X_8 \times B_8 \rightarrow T1$, R30; R4 + T1 → R4;	R4 has $X_2 \times B_2 + X_4 \times B_4$
9	nop	R3 has $X_1 \times B_1 + X_3 \times B_3 + X_5 \times B_5$
10	R3 + R4 → R5;	R4 has $X_2 \times B_2 + X_4 \times B_4 + X_6 \times B_6$
11	...	
12	...	R5 has $\sum X_i B_i$, i = 1 to 6

Figure 94.

17. Instruction Set

17.1. Code Word Format

The 3364 has a 42-bit code word. The 3164 does not use the two-bit YCNT and ZCNT fields and has a 38-bit code word. The format is shown in figures 95 and 96. The 3164 fields are different and are described in Appendix A.

ZCNT

The ZCNT field controls data transfers through the Z port. It is available in 3364 only. For more information, see section 5.5.

YCNT

The YCNT field controls data transfers through the Y port. It is available in 3364 only. For more information, see section 5.5.

XCNT

The XCNT field controls data transfers through the X port. For more information, see section 5.5.

EFADD

The EFADD field is the register file address field shared by the F write port and the E read port. It selects one of 32 general-purpose registers to be written to via the F port, or read from via the E port, or both. For more details, see section 6.3.

DADD

The DADD field selects one of 32 general-purpose registers to be written to via the D port of the register file. For more details, see section 6.2 and section 6.5.

CADD

The CADD field selects one of 32 general-purpose registers to be written to via the C port of the register file. For more details, see section 6.2 and section 6.5.

BADD

The BADD field selects one of 32 general-purpose registers to be read from via the B port of the register file. For some operations, this field is used as part of the opcode, in conjunction with the FUNC field. For more details, see section 6.2 and section 6.5.

AADD

The AADD field selects one of 32 general-purpose registers to be read from via the A port of the register file. For more details, see section 6.2 and section 6.5.

MBIN

The MBIN field selects either the Y bus or the B port of the register file as the multiplier B input. For more details, see section 7.2. For some operations, this field is used as part of the opcode, in conjunction with the FUNC field.

MAIN

The MAIN field selects either the X bus or the A port of the register file as the multiplier A input. In the case of chained multiply-add operations, MAIN is used to select between the X and the Y buses for one of the multiply operands. For more details, see section 7.2. For some operations, this field is used as part of the opcode, in conjunction with the FUNC field.

ABIN

The ABIN field selects either the Y register or the B port of the register file as the ALU B input. For more details, see section 8.2. For some operations, this field is used as part of the opcode, in conjunction with the FUNC field.

AAIN

The AAIN field selects either the X register or the A port of the register file as the ALU A input. For more details, see section 8.2. For some operations, this field is used as part of the opcode, in conjunction with the FUNC field.

FUNC

The FUNC field, sometimes in conjunction with MAIN, MBIN, AAIN, ABIN, BADD fields, selects the operation to be executed.

17.1. Code Word Format, continued

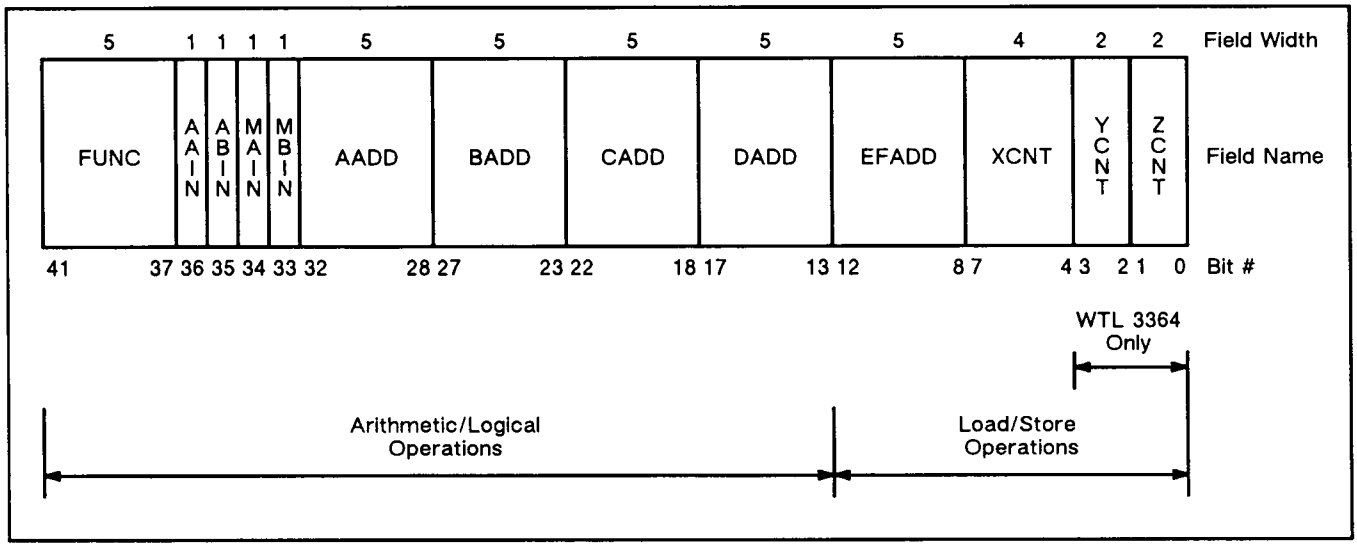


Figure 95. Code word format

CODE WORD FIELD	WIDTH (BITS)	NAME	DESCRIPTION
$C_{1..0}$	2	ZCNT	Z port control
$C_{3..2}$	2	YCNT	Y port control
$C_{7..4}$	4	XCNT	X port control
$C_{12..8}$	5	EFADD	E and/or F port register address
$C_{17..13}$	5	DADD	D write port register address
$C_{22..18}$	5	CADD	C write port register address
$C_{27..23}$	5	BADD	B read port register address
$C_{32..28}$	5	AADD	A read port register address
C_{33}	1	MBIN	Multiplier B input select
C_{34}	1	MAIN	Multiplier A input select
C_{35}	1	ABIN	ALU B input select
C_{36}	1	AAIN	ALU A input select
$C_{41..37}$	5	FUNC	Function code

Figure 96. Code word fields

17.2. Independence of the I/O and Operation Portions of the Code Word

The XCNT, YCNT, ZCNT and EFADD fields comprise the I/O portion of the code word; together they span bits $C_{12..0}$. The remaining fields, in bits $C_{41..13}$, specify the operation to be performed, operand sources and result destinations. Note that everything that needs to be known about a register-to-register operation is specified in the operation portion of the instruction at the time the instruction is issued.

It is possible to specify an instruction so that a single code word specifies all actions related to this one instruction: both the load and the operation.

It is also possible that the operation bits $C_{41..13}$ specify an operation that is independent from data transfers on the I/O ports, which are controlled by the $C_{12..0}$ portion of the code word. For example, an operation may use the register file both as a source of operands and as a destination for the results, while the X port is independently used to load the X register with a value that is unrelated to the operation.

The I/O portion of the code word is independent from the operation portion with one exception: if the operation portion specifies load or store of a code or status register.

17.3. Mnemonics

The mnemonics and notation used in this document are those used to control the 3164 in the XL programming environment. They are given here to simplify understanding of the programming model and to provide a syntax in which to present programming examples.

In the following sections, binary numbers are suffixed with "b". A range of values is indicated with "-". Don't care fields are indicated with x's.

Instructions have the form:

fop, op1, op2, ..., opn

where fop is the operation, the mnemonics for which are defined in the following sections;

op1, op2, ..., opn

are source/destination operands, with the destination on the right.

The notation for operand sources and result destinations is:

Symbol	Definition
.f0-.f31	General purpose registers in the register file
.x, .y	Input registers X and Y
.t0, .t1	Temporary latches
.sr0-.sr11	Status registers
.cr0-.cr5	Code register bytes

Figure 97.

For convenience in code examples, operands are combined into several overlapping groups:

Symbol	Registers
rreg	.f0-.f31
xreg	.f0-.f31, .x
yreg	.f0-.f31, .y
xyreg	.x, .y
tlat	.t0, .t1
sreg	.sr0-.sr11
creg	.cr0-.cr5

Figure 98.

The symbol for each group below indicates the choice of one of the available registers. Chained multiply-add instructions allow the output of the multiplier or ALU to be simultaneously written to the register file and a temporary latch. A syntax for these operations is defined in section 17.9. Load/store instructions allow simultaneous loading of data into both the register file and the X or Y register. A syntax for this operation is defined in section 17.16.

Note that since load/store and operation portions of the code word are independent, they are described separately.

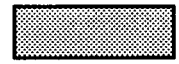
The following sections explain each instruction in detail; figures 183 and 184 provide an instruction set summary.

17.4. Instruction Interaction with the Status Register

If an instruction affects, or its result is affected by certain bits of the Status Register, then these bits are identified on a map of the Status Register.

Bits which *affect* the result of an instruction are shaded in the map as shown:

Bits which *are affected* by an instruction are shaded in the map as shown:



In some cases, a bit may *both* affect the result of the instruction, *and* it may be affected by the instruction. The shading used to identify these bits is:

17.5. Multiply and Divide Instructions

INSTRUCTIONS

Instruction		Comment
fmul	xreg, yreg, rreg	r = x × y
fdiv	xreg, yreg, rreg	r = x ÷ y
dfmul	xreg, yreg, rreg	r = x × y
dfdiv	xreg, yreg, rreg	r = x ÷ y

Figure 99.

EXAMPLES

```
fmul    .x, .y, .f0
fdiv    .x, f1, .f2
dfmul   .f0, .y, .f4
dfdiv   .f0, .f3, .f31
```

Figure 100.

ENCODING

Field	Value	Operation	Comment
FUNC	00011b	fmul	32-bit multiply
	00111b	fdiv	32-bit divide
	01011b	dfmul	64-bit multiply
	01111b	dfdiv	64-bit divide
AAIN	0	AAIN = ABIN = 0	
ABIN	0	AAIN = ABIN = 0	
MAIN	0	MUL A input = .x	
	1	MUL A input = AADD	
MBIN	0	MUL B input = .y	
	1	MUL B input = BADD	
AADD	0–31	rreg = .f0–.f31	
BADD	0–31	rreg = .f0–.f31	
CADD	xxxxxb		No ALU output
DADD	0–31	rreg = .f0–.f31	

Figure 101.

In these instructions, if DNRM CONTROL bit SR2$_4$=0, denormalized inputs are treated as zero. Single-precision instructions (fmul, fdiv) clear bits 31-0 in the destination register.

17.5 Multiply and Divide Instructions, continued

SR#	BIT#							
	7	6	5	4	3	2	1	0
SR0	Multiplier Latency	FPEX Sticky	0	0	Internal NEUT ON	Rounding Mode		Fast Mode
SR1	IEEE Software Underflow	Bypass on	FPEX Delay	I/O Mode				
SR2	NaN EN	INV EN	DVZ EN	DNRM Control	OVF EN	UNF Control	INX EN	IOVF EN
SR3	NaN	INV	DVZ	DNRM	OVF	UNF	INX	IOVF
SR4	0	TDEST0		MDEST0				
SR5	0	0	0	ADEST0				
SR6	ASTAT0			MSTAT0				
SR7	0	0	0	DIVDEST				
SR8	0	TDEST1		MDEST1				
SR9	0	0	0	ADEST1				
SR10	ASTAT1			MSTAT1				
SR11	FPEX Taken	DSR in progress	FPCN	Carry	DIVSTAT			

Figure 102. Status register map; instructions fmul, dfmul (note: $SR0_7$ affects only dfmul)

SR#	BIT#							
	7	6	5	4	3	2	1	0
SR0	Multiplier Latency	FPEX Sticky	0	0	Internal NEUT ON	Rounding Mode		Fast Mode
SR1	IEEE Software Underflow	Bypass on	FPEX Delay	I/O Mode				
SR2	NaN EN	INV EN	DVZ EN	DNRM Control	OVF EN	UNF Control	INX EN	IOVF EN
SR3	NaN	INV	DVZ	DNRM	OVF	UNF	INX	IOVF
SR4	0	TDEST0		MDEST0				
SR5	0	0	0	ADEST0				
SR6	ASTAT0			MSTAT0				
SR7	0	0	0	DIVDEST				
SR8	0	TDEST1		MDEST1				
SR9	0	0	0	ADEST1				
SR10	ASTAT1			MSTAT1				
SR11	FPEX Taken	DSR in progress	FPCN	Carry	DIVSTAT			

Figure 103. Status register map; instructions fdiv, dfdiv

17.6. Add Instructions

INSTRUCTIONS

Instruction		Comment
fadd	xreg, yreg, rreg	r = x + y
fsub	xreg, yreg, rreg	r = x – y
fsubr	xreg, yreg, rreg	r = –x + y
dfadd	xreg, yreg, rreg	r = x + y
dfsub	xreg, yreg, rreg	r = x – y
dfsubr	xreg, yreg, rreg	r = –x + y

Figure 104.

EXAMPLES

```
fadd    .x, .y, .f0
fsub    .x, .f1, .f2
dfsubr  .f0, .y, .f4
dfadd   .f0, .f3, .f31
```

Figure 105.

ENCODING

Field	Value	MAIN,MBIN	Operation	Comment
FUNC	00100b	0,0	fadd	32-bit add
		0,1	fsubr	32-bit reverse subtract
		1,0	fsub	32-bit subtract
	01100b	0,0	dfadd	64-bit add
		0,1	dfsubr	64-bit reverse subtract
		1,0	dfsub	64-bit subtract
AAIN	0		ALU A input = .x	
	1		ALU A input = AADD	
ABIN	0		ALU B input = .y	
	1		ALU B input = BADD	
AADD	0–31		rreg = .f0–.f31	
BADD	0–31		rreg = .f0–.f31	
CADD	0–31		rreg = .f0–.f31	
DADD	xxxxxb			No MUL output

Figure 106.

In these instructions, if DNRM CONTROL bit $SR2_4$=0, denormalized inputs are treated as zero.

Single-precision instructions (fadd, fsub, fsubr) clear bits 31-0 in the destination register.

17.6. Add Instructions, continued

SR#	BIT#							
	7	6	5	4	3	2	1	0
SR0	Multiplier Latency	FPEX Sticky	0	0	Internal NEUT ON	Rounding Mode		Fast Mode
SR1	IEEE Software Underflow	Bypass on	FPEX Delay	I/O Mode				
SR2	NaN EN	INV EN	DVZ EN	DNRM Control	OVF EN	UNF Control	INX EN	IOVF EN
SR3	NaN	INV	DVZ	DNRM	OVF	UNF	INX	IOVF
SR4	0	TDEST0		MDEST0				
SR5	0	0	0	ADEST0				
SR6	ASTAT0				MSTAT0			
SR7	0	0	0	DIVDEST				
SR8	0	TDEST1		MDEST1				
SR9	0	0	0	ADEST1				
SR10	ASTAT1				MSTAT1			
SR11	FPEX Taken	DSR in progress	FPCN	Carry	DIVSTAT			

Figure 107. Status register map; instructions fadd, fsub, fsubr, dfadd, dfsub, dfsubr

17.7. Compare Instructions

INSTRUCTIONS

Instruction	Comment
fcmp xreg, yreg, cond	update FPCN output and status register bit $SR11_5$
dfcmp xreg, yreg, cond	update FPCN output and status register bit $SR11_5$

Figure 108.

In figure 108 "cond" stands for "condition code" and is interpreted as follows:

Cond	Result of (x-y)	Symbol
.eq	equal to zero	(x = y)
.gt	greater than zero	(x > y)
.lt	less than zero	(x < y)
.uord	unordered	(x ? y)

Figure 109.

The timing of the FPCN output and status register $SR11_5$ bit update is shown in figure 79.

EXAMPLES

```
fcmp   .x, .y, .eq
fcmp   .x, .f1, .uord
dfcmp  .f0, .y, .gt
dfcmp  .f0, .f3, .lt
```

Figure 110.

17.7. Compare Instructions, continued

ENCODING

Field	Value	AAIN,ABIN	Operation	Comment	
FUNC	00101b	0,0	if x = y, FPCN ← 1, else FPCN ← 0	fcmp, no write	32-bit compare
		0,1	If x > y, FPCN ← 1, else FPCN ← 0	fcmp, no write	32-bit compare
		1,0	If x < y, FPCN ← 1, else FPCN ← 0	fcmp, no write	32-bit compare
		1,1	If x ? y, FPCN ← 1, else FPCN ← 0	fcmp, no write	32-bit compare
	01101b	0,0	If x = y, FPCN ← 1, else FPCN ← 0	dfcmp, no write	64-bit compare
		0,1	If x > y, FPCN ← 1, else FPCN ← 0	dfcmp, no write	64-bit compare
		1,0	If x < y, FPCN ← 1, else FPCN ← 0	dfcmp, no write	64-bit compare
		1,1	If x ? y, FPCN ← 1, else FPCN ← 0	dfcmp, no write	64-bit compare
MAIN	0		MUL A input = .y		
	1		MUL A input = AADD		
MBIN	0		MUL B input = .x		
	1		MUL B input = BADD		
AADD	0–31		rreg = .f0–.f31		
BADD	0–31		rreg = .f0–.f31		
CADD	xxxxxb			No ALU output	
DADD	xxxxxb			No MUL output	

Figure 111.

In these instructions, if DNRM CONTROL bit $SR2_4 = 0$, denormalized inputs are treated as zero. Therefore a comparison of zero with any denormalized number will yield equality.

17.7. Compare Instructions, continued

SR#	BIT#							
	7	6	5	4	3	2	1	0
SR0	Multiplier Latency	FPEX Sticky	0	0	Internal NEUT ON	Rounding Mode		Fast Mode
SR1	IEEE Software Underflow	Bypass on	FPEX Delay	I/O Mode				
SR2	NaN EN	INV EN	DVZ EN	DNRM Control	OVF EN	UNF Control	INX EN	IOVF EN
SR3	NaN	INV	DVZ	DNRM	OVF	UNF	INX	IOVF
SR4	0	TDEST0		MDEST0				
SR5	0	0	0	ADEST0				
SR6	ASTAT0			MSTAT0				
SR7	0	0	0	DIVDEST				
SR8	0	TDEST1		MDEST1				
SR9	0	0	0	ADEST1				
SR10	ASTAT1			MSTAT1				
SR11	FPEX Taken	DSR in progress	FPCN	Carry	DIVSTAT			

Figure 112. Status register map; instructions fcmp, dfcmp

Note: instructions

```
fcmp .f0, .f1, .uord
```

or

```
dfcmp .f0, .f1, .uord
```

i.e., comparisons that specify the unordered condition, will never set $SR3_7$ (NaN sticky flag).

17.8. Concurrent Multiply-Add Instructions

Concurrent multiply-add instructions allow the programmer to obtain maximum throughput from the 3x64 by simultaneously operating both the multiplier and the ALU. In order to supply both the multiplier and ALU with operands, four input values are required. The on-chip register file can provide two of these inputs, selected by the AADD and BADD fields. The X and Y registers are used to provide the other two input operands. They can be used to provide both input operands to either of the arithmetic units, or the X register can be used as an input to one arithmetic unit, and the Y register as an input to the other.

These instructions are especially useful in Configuration A. They can be used in Configurations B or C, but some problems that could make best use of these instructions will be I/O-bound on these architectures.

If xreg selects a file register on both sides of an instruction (as opposed to a file register on one side and the X register on the other), then it must be the same register. The reason for this is that only the AADD field is available to select the file register for the xreg operand. The same applies to the yreg, except the BADD field selects the yreg operand from the register file. See examples.

INSTRUCTIONS

Instruction	Comment
fmul xreg, yreg, rreg ; fadd xreg, yreg, rreg	r1 = x × y; r2 = x + y
fmul xreg, yreg, rreg ; fsubr xreg, yreg, rreg	r1 = x × y; r2 = −x + y
fmul xreg, yreg, rreg ; fsub xreg, yreg, rreg	r1 = x × y; r2 = x − y
dfmul xreg, yreg, rreg ; dfadd xreg, yreg, rreg	r1 = x × y; r2 = x + y
dfmul xreg, yreg, rreg ; dfsubr xreg, yreg, rreg	r1 = x × y; r2 = −x + y
dfmul xreg, yreg, rreg ; dfsub xreg, yreg, rreg	r1 = x × y; r2 = x − y

Figure 113.

EXAMPLES

```
fmul    .x, .y, .f0       ; fadd .f1, .f2, .f3
fmul    .f4, .f5, .f0     ; fadd .x, .y, .f3
fmul    .x, .f5, .f20     ; fadd .f31, .y, .f30
fmul    .f13, .y, .f31    ; fadd .x, .f2, .f30
```

Figure 114.

It is possible to use the same register as an input to both the multiplier and the ALU, as in the following examples:

```
fmul .x, .f2, .f0         ; fadd .f1, .f2, .f3
fmul .f1, .f2, .f0        ; fadd .f1, .y, .f3
```

Figure 115.

17.8. Concurrent Multiply-Add Instructions, continued

ENCODING

Field	Value	Operation	Comment
FUNC	00000b	fmul...; fadd...	32-bit multiply; 32-bit add
	00001b	fmul...; fsubr...	32-bit multiply; 32-bit reverse subtract
	00010b	fmul...; fsub...	32-bit multiply; 32-bit subtract
	01000b	dfmul...; dfadd...	64-bit multiply; 64-bit add
	01001b	dfmul...; dfsubr...	64-bit multiply; 64-bit reverse subtract
	01010b	dfmul...; dfsub...	64-bit multiply; 64-bit subtract
AAIN	0	ALU A input = .x	
	1	ALU A input = AADD	
ABIN	0	ALU B input = .y	
	1	ALU B input = BADD	
MAIN	0	MUL A input = .x	
	1	MUL A input = AADD	
MBIN	0	MUL B input = .y	
	1	MUL B input = BADD	
AADD	0–31	rreg = .f0–.f31	
BADD	0–31	rreg = .f0–.f31	
CADD	0–31	rreg = .f0–.f31	
DADD	0–31	rreg = .f0–.f31	

Figure 116.

In these instructions, if DNRM CONTROL bit $SR2_4$=0, denormalized inputs are treated as zero.

Single-precision instructions clear bits 31-0 in the destination register.

17.8. Concurrent Multiply-Add Instructions, continued

SR#	BIT#							
	7	6	5	4	3	2	1	0
SR0	Multiplier Latency	FPEX Sticky	0	0	Internal NEUT ON	Rounding Mode		Fast Mode
SR1	IEEE Software Underflow	Bypass on	FPEX Delay	I/O Mode				
SR2	NaN EN	INV EN	DVZ EN	DNRM Control	OVF EN	UNF Control	INX EN	IOVF EN
SR3	NaN	INV	DVZ	DNRM	OVF	UNF	INX	IOVF
SR4	0	TDEST0		MDEST0				
SR5	0	0	0	ADEST0				
SR6	ASTAT0			MSTAT0				
SR7	0	0	0	DIVDEST				
SR8	0	TDEST1		MDEST1				
SR9	0	0	0	ADEST1				
SR10	ASTAT1			MSTAT1				
SR11	FPEX Taken	DSR in progress	FPCN	Carry	DIVSTAT			

Figure 117. Status register map; concurrent multiply-add instructions

17.9. Chained Multiply-Add Instructions

The chained multiply-add instructions are the other set of instructions which result in the maximum throughput available from the 3x64. They can be used in pipelined polynomial algorithms, matrix transforms, and other applications where there is a need to calculate sum-of-products.

In order to feed the multiplier and the ALU simultaneously, four input operands are required. The on-chip register file provides two of the four inputs, selected by the AADD and BADD fields. The chained multiply-add instructions use one of the X or Y registers and one of the T-latches to provide the other two. The multiplier uses either the X or the Y register as one of its inputs; the ALU uses one of the T-latches as one of its inputs. The other inputs to the multiplier and ALU are supplied by the register file. See section 10 also.

In chained multiply-add instructions, it is possible to specify a floating-point pass operation instead of a multiply.

Chained multiply-add instructions must be used in a very specific way, as described below.

The multiplier must write its output to two destinations: a file register and a T-latch. The ALU must use the same T-latch as its input. The value presented to the ALU in the T-latch is the result of the most recently completed multiply which has this T-latch as one of its destinations; depending on the multiplier latency mode, this multiply began two, three, or more cycles earlier.

The chained multiply-add instructions are the only ones that use the T-latches. There are no instructions which directly read or write the T-latches. There are no instructions which allow the multiplier to write into a T-latch without having to perform an ALU operation. There are no instructions that allow one to perform an ALU instruction using a T-latch as an input without starting a multiply on the same cycle that will overwrite the *same* T-latch two or three cycles later, depending on the latency mode.

17.9. Chained Multiply-Add Instructions, continued

INSTRUCTIONS

In the following, TLAT stands for (rreg and .t0) or (rreg and .t1). Both TLAT and tlat must select the same latch.

Instruction				Comment
Single-precision multiply, single-precision add, single-precision result:				
fmul	xyreg, rreg, TLAT ;	fadd	rreg, tlat, rreg	r, t = x × b, r = a + t
fmul	xyreg, rreg, TLAT ;	fsub	rreg, tlat, rreg	r, t = x × b, r = a − t
fmul	xyreg, rreg, TLAT ;	fsubr	rreg, tlat, rreg	r, t = x × b, r =−a + t
Double-precision multiply, double-precision add, double-precision result:				
dfmul	xyreg, rreg, TLAT ;	dfadd	rreg, tlat, rreg	r, t = x × b, r = a + t
dfmul	xyreg, rreg, TLAT ;	dfsub	rreg, tlat, rreg	r, t = x × b, r = a − t
dfmul	xyreg, rreg, TLAT ;	dfsubr	rreg, tlat, rreg	r, t = x × b, r =−a + t
F32 × F64 → F64, double-precision add, double-precision result:				
fddmul	xyreg, rreg, TLAT ;	dfadd	reg, tlat, rreg	r, t = x × b, r = a + t
F32 × F32 → F64, double-precision add, double-precision result:				
ffdmul	xyreg, rreg, TLAT ;	dfadd	rreg, tlat, rreg	r, t = x × b, r = a + t
fpass	xyreg, TLAT ;	fadd	rreg, tlat, rreg	r, t = x; r = a + t
fpass	xyreg, TLAT ;	fsub	rreg, tlat, rreg	r, t = x; r = a − t
fpass	xyreg, TLAT ;	fsubr	rreg, tlat, rreg	r, t = x; r =−a + t
dfpass	xyreg, TLAT ;	dfadd	rreg, tlat, rreg	r, t = x; r = a + t
dfpass	xyreg, TLAT ;	dfsub	rreg, tlat, rreg	r, t = x; r = a − t
dfpass	xyreg, TLAT ;	dfsubr	rreg, tlat, rreg	r, t = x; r =−a + t

Figure 118.

17.9. Chained Multiply-Add Instructions, continued

ENCODING

Field	Value	Operation	Comment
FUNC	10000b	fmul...; fadd...	32-bit mul, add
	10001b	fmul...; fsubr...	32-bit mul, subr
	10010b	fmul...; fsub...	32-bit mul, sub
	10011b	ffdmul...; dfadd...	$F_{32} \times F_{32} \rightarrow F_{64}$, $F_{64} + F_{64} \rightarrow F_{64}$
	10100b	dfmul...; dfadd...	64-bit mul, add
	10101b	dfmul...; dfsubr...	64-bit mul, subr
	10110b	dfmul...; dfsub...	64-bit mul, sub
	10111b	fddmul...; dfadd...	$F_{32} \times F_{64} \rightarrow F_{64}$, $F_{64} + F_{64} \rightarrow F_{64}$
AAIN, ABIN	0,0	Select .t0 for MUL output and ALU input	ALU A input always uses AADD
	1,0	Select .t1 for MUL output and ALU input	
	0,1	Do not load either T-latch from multiplier output; select BADD for ALU input.	
MAIN	0	MUL A input = .x	
	1	MUL A input = .y	
MBIN	0	Pass A	
	1	MUL B input = BADD	
AADD	0-31	.f0-.f31	
BADD	0-31	.f0-.f31	
CADD	0-31	.f0-.f31	
DADD	0-31	.f0-.f31	

Figure 119.

In these instructions, if DNRM CONTROL bit $SR2_4$=0, denormalized inputs are treated as zero. Single-precision instructions clear bits 31-0 in the destination register.

EXAMPLES

```
fmul .x, .f1, .f2, .t0    ; fadd .f3, .t0, .f4
fmul .y, .f1, .f31, .t1   ; fadd .f30, .t1, .f6
dfmul .y, .f1, .f11, .t1 ; dfsub .f3, .t1, .f6
fddmul .x, .f5, .f6, .t0 ; dfadd .f4, .t0, .f7
```

Figure 120.

17.9. Chained Multiply-Add Instructions, continued

SR#	BIT#							
	7	6	5	4	3	2	1	0
SR0	Multiplier Latency	FPEX Sticky	0	0	Internal NEUT ON	Rounding Mode		Fast Mode
SR1	IEEE Software Underflow	Bypass on	FPEX Delay	I/O Mode				
SR2	NaN EN	INV EN	DVZ EN	DNRM Control	OVF EN	UNF Control	INX EN	IOVF EN
SR3	NaN	INV	DVZ	DNRM	OVF	UNF	INX	IOVF
SR4	0	TDEST0		MDEST0				
SR5	0	0	0	ADEST0				
SR6	ASTAT0			MSTAT0				
SR7	0	0	0	DIVDEST				
SR8	0	TDEST1		MDEST1				
SR9	0	0	0	ADEST1				
SR10	ASTAT1			MSTAT1				
SR11	FPEX Taken	DSR in progress	FPCN	Carry	DIVSTAT			

Figure 121. Status register map; chained multiply-add instructions

17.10. Integer Instructions

17.10.1. INTEGER ADD AND SUBTRACT

INSTRUCTIONS

Instruction		Comment
faddi	xreg, yreg, rreg	r = x + y
fsubir	xreg, yreg, rreg	r = −x + y
faddic	xreg, yreg, rreg	r = x + y + carry
fsubirc	xreg, yreg, rreg	r = −x + y −1 + carry

Figure 122.

EXAMPLES

```
faddi   .x, .y, .f0
fsubir  .x, .f1, .f2
faddic  .f0, .y, .f4
fsubirc .f0, .f3, .f31
```

Figure 123.

ENCODING

Field	Value	Operation	Comment
FUNC	11000b	faddi	32-bit integer add
		faddic	32-bit integer add with carry
		fsubir	32-bit integer subr
		fsubirc	32-bit integer subr with borrow
AAIN	0	no carry (borrow)	
	1	use carry (borrow)	
ABIN	0	operation is addition	
	1	operation is reverse subtraction	
MAIN	0	MUL A input = .x	
	1	MUL A input = AADD	
MBIN	0	MUL B input = .y	
	1	MUL B input = BADD	
AADD	0–31	.f0–.f31	
BADD	0–31	.f0–.f31	
CADD	xxxxxb		No ALU output
DADD	0–31	.f0–.f31	

Figure 124.

17.10. Integer Instructions, continued

SR#	BIT#							
	7	6	5	4	3	2	1	0
SR0	Multiplier Latency	FPEX Sticky	0	0	Internal NEUT ON	Rounding Mode		Fast Mode
SR1	IEEE Software Underflow	Bypass on	FPEX Delay	I/O Mode				
SR2	NaN EN	INV EN	DVZ EN	DNRM Control	OVF EN	UNF Control	INX EN	IOVF EN
SR3	NaN	INV	DVZ	DNRM	OVF	UNF	INX	IOVF
SR4	0	TDEST0		MDEST0				
SR5	0	0	0	ADEST0				
SR6	ASTAT0			MSTAT0				
SR7	0	0	0	DIVDEST				
SR8	0	TDEST1		MDEST1				
SR9	0	0	0	ADEST1				
SR10	ASTAT1			MSTAT1				
SR11	FPEX Taken	DSR in progress	FPCN	Carry	DIVSTAT			

Figure 125. Status register map; instructions fmulil, fmulim, faddi, fsubir

SR#	BIT#							
	7	6	5	4	3	2	1	0
SR0	Multiplier Latency	FPEX Sticky	0	0	Internal NEUT ON	Rounding Mode		Fast Mode
SR1	IEEE Software Underflow	Bypass on	FPEX Delay	I/O Mode				
SR2	NaN EN	INV EN	DVZ EN	DNRM Control	OVF EN	UNF Control	INX EN	IOVF EN
SR3	NaN	INV	DVZ	DNRM	OVF	UNF	INX	IOVF
SR4	0	TDEST0		MDEST0				
SR5	0	0	0	ADEST0				
SR6	ASTAT0			MSTAT0				
SR7	0	0	0	DIVDEST				
SR8	0	TDEST1		MDEST1				
SR9	0	0	0	ADEST1				
SR10	ASTAT1			MSTAT1				
SR11	FPEX Taken	DSR in progress	FPCN	Carry	DIVSTAT			

Figure 126. Status register map; instructions faddic, fsubirc

17.10. Integer Instructions, continued

17.10.2. INTEGER MULTIPLY

INSTRUCTIONS

Instruction	Comment
fmulil xreg, yreg, rreg	r = (x × y)LS
fmulim xreg, yreg, rreg	r = (x × y)MS

Figure 127.

EXAMPLES

```
fmulil  .x, .y, .f0
fmulim  .x, .f1, .f2
fmulil  .f0, .y, .f4
fmulim  .f0, .f3, .f31
```

Figure 128.

ENCODING

Field	Value	Operation	Comment
FUNC	11010b	fmulil	32-bit integer multiply, return LS
	11011b	fmulim	32-bit integer multiply, return MS
AAIN	0	AAIN = ABIN = 0	
ABIN	0	AAIN = ABIN = 0	
MAIN	0	MUL A input = .x	
	1	MUL A input = AADD	
MBIN	0	MUL B input = .y	
	1	MUL B input = BADD	
AADD	0-31	.f0-.f31	
BADD	0-31	.f0-.f31	
CADD	xxxxxb		No ALU output
DADD	0-31	.f0-.f31	

Figure 129.

All integer instructions clear bits 63-53 and 20-0 in the destination register.

17.11. 64-bit Logical Operations

INSTRUCTIONS

Instruction		Comment
and	xreg, yreg, rreg	r = x AND y
or	xreg, yreg, rreg	r = x OR y
xor	xreg, yreg, rreg	r = x XOR y
mov*	yreg, rreg	r = x

* "mov .x, .f0" and "mov .f0, .f31" are monadic logical multiplier instructions. See section 17.13.

Figure 130.

EXAMPLES

```
and     .x, .y, .f0
or      .x, .f1, .f2
xor     .f0, .y, .f4
mov     .y, .f31
```

Figure 131.

ENCODING

Field	Value	Operation	Comment
FUNC	11001b		64-bit logical operation
AAIN, ABIN	0,0	fop = and	
	0,1	fop = or	
	1,0	fop = xor	
	1,1	mov	Logical pass: mov .y, .fn
MAIN	0	MUL A input = .x	
	1	MUL A input = AADD	
MBIN	0	MUL B input = .y	
	1	MUL B input = BADD	
AADD	0-31	rreg = .f0-.f31	
BADD	0-31	rreg = .f0-.f31	
CADD	xxxxxb		No ALU output
DADD	0-31	rreg = .f0-.f31	

Figure 132.

Note that results of logical operations may not be bypassed to floating-point operations.

17.11. 64-bit Logical Operations, continued

SR#	BIT#							
	7	6	5	4	3	2	1	0
SR0	Multiplier Latency	FPEX Sticky	0	0	Internal NEUT ON	Rounding Mode		Fast Mode
SR1	IEEE Software Underflow	Bypass on	FPEX Delay	I/O Mode				
SR2	NaN EN	INV EN	DVZ EN	DNRM Control	OVF EN	UNF Control	INX EN	IOVF EN
SR3	NaN	INV	DVZ	DNRM	OVF	UNF	INX	IOVF
SR4	0	TDEST0		MDEST0				
SR5	0	0	0	ADEST0				
SR6	ASTAT0			MSTAT0				
SR7	0	0	0	DIVDEST				
SR8	0	TDEST1		MDEST1				
SR9	0	0	0	ADEST1				
SR10	ASTAT1			MSTAT1				
SR11	FPEX Taken	DSR in progress	FPCN	Carry	DIVSTAT			

Figure 133. Status register map; instructions and, or, xor, mov

17.12. Min and Max Instructions

Min and max are defined for single- and double-precision floating-point operands. They are also defined for integer operands if these integers are unsigned. (Everywhere else on the chip, integers are treated as two's complement signed). Both operands must be of the same data type. These instructions are logical and do not signal exceptions; therefore NaN handling will not be correct. They cannot be used in programs which must meet the *IEEE Standard for Binary Floating-Point Arithmetic*.

INSTRUCTIONS

Instruction		Comment
max	xreg, yreg, rreg	r = max(x,y)
min	xreg, yreg, rreg	r = min(x,y)

Figure 134.

EXAMPLES

```
min     .x, .y, .f0
max     .x, .f1, .f2
min     .f0, .y, .f4
max     .f0, .f3, .f31
```

Figure 135.

ENCODING

Field	Value	Operation	Comment
FUNC	11101b	min/max	
AAIN, ABIN	0,0	min	
	0,1	max	
MAIN	0	MUL A input = .x	
	1	MUL A input = AADD	
MBIN	0	MUL B input = .y	
	1	MUL B input = BADD	
AADD	0–31	rreg = .f0–.f31	
BADD	0–31	rreg = .f0–.f31	
CADD	xxxxxb		No ALU output
DADD	0–31	rreg = .f0–.f31	

Figure 136.

These instructions never treat denormalized numbers as zero, even if DNRM CONTROL bit $SR2_4$=0.

17.12. Min and Max Instructions, continued

SR#	BIT#							
	7	6	5	4	3	2	1	0
SR0	Multiplier Latency	FPEX Sticky	0	0	Internal NEUT ON	Rounding Mode		Fast Mode
SR1	IEEE Software Underflow	Bypass on	FPEX Delay	I/O Mode				
SR2	NaN EN	INV EN	DVZ EN	DNRM Control	OVF EN	UNF Control	INX EN	IOVF EN
SR3	NaN	INV	DVZ	DNRM	OVF	UNF	INX	IOVF
SR4	0	TDEST0		MDEST0				
SR5	0	0	0	ADEST0				
SR6	ASTAT0			MSTAT0				
SR7	0	0	0	DIVDEST				
SR8	0	TDEST1		MDEST1				
SR9	0	0	0	ADEST1				
SR10	ASTAT1			MSTAT1				
SR11	FPEX Taken	DSR in progress	FPCN	Carry	DIVSTAT			

Figure 137. Status register map; instructions min, max

17.13. Monadic ALU Instructions

INSTRUCTIONS

Instruction		Function	Comment
fixr	xreg, rreg	r_I = round (F_{32})	convert single float to integer and round
fix	xreg, rreg	r_I = trunc (F_{32})	convert single float to integer and truncate (round to zero)
dfixr	xreg, rreg	r_I = round (F_{64})	convert double float to integer and round
dfix	xreg, rreg	r_I = trunc(F_{64})	convert double float to integer and truncate (round to zero)
float*	xreg, rreg	r_{32} = float (I_{32})	convert integer to single float and truncate (round to zero)
dfloat	xreg, rreg	r_{64} = float (I_{32})	convert integer to double float
dfcnvt	xreg, rreg	r_{32} = round (F_{64})	convert double float to single float and round
fdcnvt	xreg, rreg	r_{64} = F_{32}	convert single float to double float
sll**	xreg, rreg	r = x << 1	shift, logical, left (64-bit data), 1 bit, 0 → lsb
scl**	xreg, rreg	r = msb(x) +(x <<1)	shift, circular, left (64-bit data), 1 bit
slr**	xreg, rreg	r = x >> 1	shift, logical, right (64-bit data), 1 bit, 0 → msb
sar**	xreg, rreg	r = (x >> 1) and sign(r) = sign(x)	shift, arithmetic, right (64-bit data), 1 bit
dfdw	xreg, rreg		F_{64D} → F_{64W}
dfude	xreg, rreg		F_{64U} → F_{64D} (exact)
dfudi	xreg, rreg		F_{64U} → F_{64D} (inexact)
fdw	xreg, rreg		F_{32D} → F_{64W}
fude	xreg, rreg		F_{32U} → F_{32D} (exact)
fudi	xreg, rreg		F_{32U} → F_{32D} (inexact)

* Not IEEE: for IEEE float operations, use dfloat followed by dfcnvt.
** These are logical instructions; bypassing their results to floating-point operations is illegal.

Figure 138.

EXAMPLES

```
fixr .x, .f0
dfloat .x, .f1
dfcnvt .f0, .f4
dfdw .f0, .f3
```

Figure 139.

17.13. Monadic ALU Instructions, continued

ENCODING

Field	Value	ABIN,MBIN	Operation	Comment
FUNC	11111b			Monadic operation
AAIN	0		ALU A input = .x	
	1		ALU A input = AADD	
MAIN	0			
AADD	0-31		rreg = .f0-.f31	
BADD	01000b	0,0	dfixr	$F_{64} \rightarrow I_{32}$ (round)
		0,1	dfix	$F_{64} \rightarrow I_{32}$ (truncate)
		1,0	fixr	$F_{32} \rightarrow I_{32}$ (round)
		1,1	fix	$F_{32} \rightarrow I_{32}$ (truncate)
	01001b	1,0	dfloat	$I_{32} \rightarrow F_{64}$
		1,1	float	$I_{32} \rightarrow F_{32}$ (truncate)
	01010b	0,0	dfcnvt	$F_{64} \rightarrow F_{32}$
		1,0	fdcnvt	$F_{32} \rightarrow F_{64}$
	01100b	0,0	sll	64-bit data logical left shift 1, 0 → lsb
		0,1	scl	64-bit data circular left shift 1
		1,0	slr	64-bit data logical right shift 1, 0 → msb
		1,1	sar	64-bit data arithmetic right shift 1
	01110b	0,0	dfude	$F_{64U} \rightarrow F_{64D}$ (exact)
		0,1	dfudi	$F_{64U} \rightarrow F_{64D}$ (inexact)
		1,0	fude	$F_{32U} \rightarrow F_{32D}$ (exact)
		1,1	fudi	$F_{32U} \rightarrow F_{32D}$ (inexact)
	01111b	0,0	dfdw	$F_{64D} \rightarrow F_{64W}$
		1,0	fdw	$F_{32D} \rightarrow F_{32W}$
CADD	0 -31		rreg = .f0-.f31	
DADD	xxxxxb			No MUL output

Figure 140.

If DNRM CONTROL bit $SR2_4$ = 0, instructions dfcnvt, fdcnvt, and fix treat denormalized inputs as zero.

17.13. Monadic ALU Instructions, continued

SR#	BIT#							
	7	6	5	4	3	2	1	0
SR0	Multiplier Latency	FPEX Sticky	0	0	Internal NEUT ON	Rounding Mode		Fast Mode
SR1	IEEE Software Underflow	Bypass on	FPEX Delay	I/O Mode				
SR2	NaN EN	INV EN	DVZ EN	DNRM Control	OVF EN	UNF Control	INX EN	IOVF EN
SR3	NaN	INV	DVZ	DNRM	OVF	UNF	INX	IOVF
SR4	0	TDEST0		MDEST0				
SR5	0	0	0	ADEST0				
SR6	ASTAT0			MSTAT0				
SR7	0	0	0	DIVDEST				
SR8	0	TDEST1		MDEST1				
SR9	0	0	0	ADEST1				
SR10	ASTAT1			MSTAT1				
SR11	FPEX Taken	DSR in progress	FPCN	Carry	DIVSTAT			

Figure 141. Status register map; instructions fixr, dfixr

SR#	BIT#							
	7	6	5	4	3	2	1	0
SR0	Multiplier Latency	FPEX Sticky	0	0	Internal NEUT ON	Rounding Mode		Fast Mode
SR1	IEEE Software Underflow	Bypass on	FPEX Delay	I/O Mode				
SR2	NaN EN	INV EN	DVZ EN	DNRM Control	OVF EN	UNF Control	INX EN	IOVF EN
SR3	NaN	INV	DVZ	DNRM	OVF	UNF	INX	IOVF
SR4	0	TDEST0		MDEST0				
SR5	0	0	0	ADEST0				
SR6	ASTAT0			MSTAT0				
SR7	0	0	0	DIVDEST				
SR8	0	TDEST1		MDEST1				
SR9	0	0	0	ADEST1				
SR10	ASTAT1			MSTAT1				
SR11	FPEX Taken	DSR in progress	FPCN	Carry	DIVSTAT			

Figure 142. Status register map; instructions fix, dfix

17.13. Monadic ALU Instructions, continued

SR#	BIT#							
	7	6	5	4	3	2	1	0
SR0	Multiplier Latency	FPEX Sticky	0	0	Internal NEUT ON	Rounding Mode		Fast Mode
SR1	IEEE Software Underflow	Bypass on	FPEX Delay	I/O Mode				
SR2	NaN EN	INV EN	DVZ EN	DNRM Control	OVF EN	UNF Control	INX EN	IOVF EN
SR3	NaN	INV	DVZ	DNRM	OVF	UNF	INX	IOVF
SR4	0	TDEST0		MDEST0				
SR5	0	0	0	ADEST0				
SR6	ASTAT0			MSTAT0				
SR7	0	0	0	DIVDEST				
SR8	0	TDEST1		MDEST1				
SR9	0	0	0	ADEST1				
SR10	ASTAT1			MSTAT1				
SR11	FPEX Taken	DSR in progress	FPCN	Carry	DIVSTAT			

Figure 143. Status register map; instruction float

SR#	BIT#							
	7	6	5	4	3	2	1	0
SR0	Multiplier Latency	FPEX Sticky	0	0	Internal NEUT ON	Rounding Mode		Fast Mode
SR1	IEEE Software Underflow	Bypass on	FPEX Delay	I/O Mode				
SR2	NaN EN	INV EN	DVZ EN	DNRM Control	OVF EN	UNF Control	INX EN	IOVF EN
SR3	NaN	INV	DVZ	DNRM	OVF	UNF	INX	IOVF
SR4	0	TDEST0		MDEST0				
SR5	0	0	0	ADEST0				
SR6	ASTAT0			MSTAT0				
SR7	0	0	0	DIVDEST				
SR8	0	TDEST1		MDEST1				
SR9	0	0	0	ADEST1				
SR10	ASTAT1			MSTAT1				
SR11	FPEX Taken	DSR in progress	FPCN	Carry	DIVSTAT			

Figure 144. Status register map; instruction dfloat

17.13. Monadic ALU Instructions, continued

SR#	BIT#							
	7	6	5	4	3	2	1	0
SR0	Multiplier Latency	FPEX Sticky	0	0	Internal NEUT ON	Rounding Mode		Fast Mode
SR1	IEEE Software Underflow	Bypass on	FPEX Delay	I/O Mode				
SR2	NaN EN	INV EN	DVZ EN	DNRM Control	OVF EN	UNF Control	INX EN	IOVF EN
SR3	NaN	INV	DVZ	DNRM	OVF	UNF	INX	IOVF
SR4	0	TDEST0		MDEST0				
SR5	0	0	0	ADEST0				
SR6	ASTAT0			MSTAT0				
SR7	0	0	0	DIVDEST				
SR8	0	TDEST1		MDEST1				
SR9	0	0	0	ADEST1				
SR10	ASTAT1			MSTAT1				
SR11	FPEX Taken	DSR in progress	FPCN	Carry	DIVSTAT			

Figure 145. Status register map; instruction dfcnvt

SR#	BIT#							
	7	6	5	4	3	2	1	0
SR0	Multiplier Latency	FPEX Sticky	0	0	Internal NEUT ON	Rounding Mode		Fast Mode
SR1	IEEE Software Underflow	Bypass on	FPEX Delay	I/O Mode				
SR2	NaN EN	INV EN	DVZ EN	DNRM Control	OVF EN	UNF Control	INX EN	IOVF EN
SR3	NaN	INV	DVZ	DNRM	OVF	UNF	INX	IOVF
SR4	0	TDEST0		MDEST0				
SR5	0	0	0	ADEST0				
SR6	ASTAT0			MSTAT0				
SR7	0	0	0	DIVDEST				
SR8	0	TDEST1		MDEST1				
SR9	0	0	0	ADEST1				
SR10	ASTAT1			MSTAT1				
SR11	FPEX Taken	DSR in progress	FPCN	Carry	DIVSTAT			

Figure 146. Status register map; instruction fdcnvt

17.13. Monadic ALU Instructions, continued

SR#	BIT#							
	7	6	5	4	3	2	1	0
SR0	Multiplier Latency	FPEX Sticky	0	0	Internal NEUT ON	Rounding Mode		Fast Mode
SR1	IEEE Software Underflow	Bypass on	FPEX Delay	I/O Mode				
SR2	NaN EN	INV EN	DVZ EN	DNRM Control	OVF EN	UNF Control	INX EN	IOVF EN
SR3	NaN	INV	DVZ	DNRM	OVF	UNF	INX	IOVF
SR4	0	TDEST0		MDEST0				
SR5	0	0	0	ADEST0				
SR6	ASTAT0			MSTAT0				
SR7	0	0	0	DIVDEST				
SR8	0	TDEST1		MDEST1				
SR9	0	0	0	ADEST1				
SR10	ASTAT1			MSTAT1				
SR11	FPEX Taken	DSR in progress	FPCN	Carry	DIVSTAT			

Figure 147. Status register map; instructions sll, scl, slr, sar

SR#	BIT#							
	7	6	5	4	3	2	1	0
SR0	Multiplier Latency	FPEX Sticky	0	0	Internal NEUT ON	Rounding Mode		Fast Mode
SR1	IEEE Software Underflow	Bypass on	FPEX Delay	I/O Mode				
SR2	NaN EN	INV EN	DVZ EN	DNRM Control	OVF EN	UNF Control	INX EN	IOVF EN
SR3	NaN	INV	DVZ	DNRM	OVF	UNF	INX	IOVF
SR4	0	TDEST0		MDEST0				
SR5	0	0	0	ADEST0				
SR6	ASTAT0			MSTAT0				
SR7	0	0	0	DIVDEST				
SR8	0	TDEST1		MDEST1				
SR9	0	0	0	ADEST1				
SR10	ASTAT1			MSTAT1				
SR11	FPEX Taken	DSR in progress	FPCN	Carry	DIVSTAT			

Figure 148. Status register map; instructions dfdw, dfude, dfudi, fdw, fude, fudi

17.14. Monadic Multiplier Instructions

INSTRUCTIONS

Instruction		Comment
fsqrt	xreg, rreg	$r = \sqrt{x}$
dfsqrt	xreg, rreg	$r = \sqrt{x}$
fabs	xreg, rreg	$r = \|x\|$
dfabs	xreg, rreg	$r = \|x\|$
fabsi	xreg, rreg	$r_i = \|x_i\|$
fnegi	xreg, rreg	$r = -x$
fnegic	xreg, rreg	$r = -x - 1 + carry$
faddic*	xreg, rreg	$r = x + carry$
fpass	xreg, rreg	$r = x$
dfpass	xreg, rreg	$r = x$
mov	xreg, rreg	$r = x$
clr**	rreg	$r = 0$
neg**	xreg, rreg	$r = -x$
dneg**	xreg, rreg	$r = -x$
not	xreg, rreg	$r = x'$

*This is the same assembler mnemonic as the dyadic faddic. The assembler selects the appropriate function code by examining the number of registers involved.

**These instructions are logical; bypassing of their results to floating-point operations is illegal.

Figure 149.

EXAMPLES

```
fsqrt   .x, .f0
dfsqrt  .x, .f1
dfabs   .f0, .f4
mov     .f0, .f3
```

Figure 150.

17.14. Monadic Multiplier Instructions, continued

ENCODING

Field	Value	ABIN,MBIN	Operation	Comment
FUNC	11111b			
AAIN	0			
MAIN	0		MUL A input = .x	
	1		MUL A input = AADD	
AADD	0–31		rreg = .f0–.f31	
BADD	00100b	0,0	dfsqrt	$\sqrt{F_{64}}$
		1,0	fsqrt	$\sqrt{F_{32}}$
	00101b	0,0	fabsi	$\lvert I_{32} \rvert$
		0,1	fnegi	I_{32} arithmetic negate
		1,0	faddic	I_{32} add carry
		1,1	fnegib	I_{32} arith. negate with borrow
	00110b	0,0	dfpass	F_{64} pass*
		0,1	dfabs	$\lvert F_{64} \rvert$
		1,0	fpass	F_{32} pass*
		1,1	fabs	$\lvert F_{32} \rvert$
	00111b	0,0	clr	
		0,1	neg, dneg	
		1,0	not	
		1,1	mov	logical pass
CADD	xxxxxb		No ALU output	
DADD	0–31		rreg = .f0–.f31	

* The F_{64} and F_{32} pass operations also set the status code, both on the $S_{3..0}$ pins and in the status register. By contrast, the mov operation (logical pass) does not affect status. Both operations pass NaNs unchanged.

Figure 151.

If DNRM CONTROL bit $SR2_4$ = 0, instructions dfsqrt, fsqrt, fabs, dfabs, fpass, dfpass treat denormalized inputs as zero.

Integer instructions (fabsi, fnegi, faddic, fnegib) clear bits 63–53 and 20–0 in the destination register.

Single-precision instructions (fpass, fabs, fsqrt) clear bits 31–0 in the destination register.

17.14. Monadic Multiplier Instructions, continued

SR#	BIT#							
	7	6	5	4	3	2	1	0
SR0	Multiplier Latency	FPEX Sticky	0	0	Internal NEUT ON	Rounding Mode		Fast Mode
SR1	IEEE Software Underflow	Bypass on	FPEX Delay	I/O Mode				
SR2	NaN EN	INV EN	DVZ EN	DNRM Control	OVF EN	UNF Control	INX EN	IOVF EN
SR3	NaN	INV	DVZ	DNRM	OVF	UNF	INX	IOVF
SR4	0	TDEST0		MDEST0				
SR5	0	0	0	ADEST0				
SR6	ASTAT0			MSTAT0				
SR7	0	0	0	DIVDEST				
SR8	0	TDEST1		MDEST1				
SR9	0	0	0	ADEST1				
SR10	ASTAT1			MSTAT1				
SR11	FPEX Taken	DSR in progress	FPCN	Carry	DIVSTAT			

Figure 152. Status register map; instructions fsqrt, dfsqrt

SR#	BIT#							
	7	6	5	4	3	2	1	0
SR0	Multiplier Latency	FPEX Sticky	0	0	Internal NEUT ON	Rounding Mode		Fast Mode
SR1	IEEE Software Underflow	Bypass on	FPEX Delay	I/O Mode				
SR2	NaN EN	INV EN	DVZ EN	DNRM Control	OVF EN	UNF Control	INX EN	IOVF EN
SR3	NaN	INV	DVZ	DNRM	OVF	UNF	INX	IOVF
SR4	0	TDEST0		MDEST0				
SR5	0	0	0	ADEST0				
SR6	ASTAT0			MSTAT0				
SR7	0	0	0	DIVDEST				
SR8	0	TDEST1		MDEST1				
SR9	0	0	0	ADEST1				
SR10	ASTAT1			MSTAT1				
SR11	FPEX Taken	DSR in progress	FPCN	Carry	DIVSTAT			

Figure 153. Status register mapl; instructions fabs, dfabs, fpass, dfpass

17.14. Monadic Multiplier Instructions, continued

SR#	BIT#							
	7	6	5	4	3	2	1	0
SR0	Multiplier Latency	FPEX Sticky	0	0	Internal NEUT ON	Rounding Mode		Fast Mode
SR1	IEEE Software Underflow	Bypass on	FPEX Delay	I/O Mode				
SR2	NaN EN	INV EN	DVZ EN	DNRM Control	OVF EN	UNF Control	INX EN	IOVF EN
SR3	NaN	INV	DVZ	DNRM	OVF	UNF	INX	IOVF
SR4	0	TDEST0		MDEST0				
SR5	0	0	0	ADEST0				
SR6	ASTAT0			MSTAT0				
SR7	0	0	0	DIVDEST				
SR8	0	TDEST1		MDEST1				
SR9	0	0	0	ADEST1				
SR10	ASTAT1			MSTAT1				
SR11	FPEX Taken	DSR in progress	FPCN	Carry	DIVSTAT			

Figure 154. Status register map; instructions fabsi, fnegi, faddic

SR#	BIT#							
	7	6	5	4	3	2	1	0
SR0	Multiplier Latency	FPEX Sticky	0	0	Internal NEUT ON	Rounding Mode		Fast Mode
SR1	IEEE Software Underflow	Bypass on	FPEX Delay	I/O Mode				
SR2	NaN EN	INV EN	DVZ EN	DNRM Control	OVF EN	UNF Control	INX EN	IOVF EN
SR3	NaN	INV	DVZ	DNRM	OVF	UNF	INX	IOVF
SR4	0	TDEST0		MDEST0				
SR5	0	0	0	ADEST0				
SR6	ASTAT0			MSTAT0				
SR7	0	0	0	DIVDEST				
SR8	0	TDEST1		MDEST1				
SR9	0	0	0	ADEST1				
SR10	ASTAT1			MSTAT1				
SR11	FPEX Taken	DSR in progress	FPCN	Carry	DIVSTAT			

Figure 155. Status register map; instructions clr, neg, dneg, not, mov

17.15. Miscellaneous Instructions

17.15.1. NOP INSTRUCTION

The nop instruction preserves the state of the 3x64. Note: The nop instruction is not affected by, and has no effect upon, the status register.

INSTRUCTION

Instruction	Comment
nop	

Figure 156.

ENCODING

Field	Value
FUNC	11111b
AAIN	0
ABIN	0
MAIN	0
MBIN	0
BADD	0

Figure 157.

17.15.2. RE-EXECUTE CODE REGISTER INSTRUCTION

INSTRUCTION

This instruction is used to execute the instruction stored in the code register. It re-executes both the arithmetic/ logical and I/O portions of the instruction. The load/ store operation can be suppressed by clearing the I/O portion of the code register (i.e. making the load/store operation a nop).

Instruction	Comment
excr	Execute the instruction in the code register

Figure 158.

ENCODING

Field	Value
FUNC	11111b
AAIN	0
ABIN	1
MAIN	0
MBIN	0
BADD	0

Figure 159.

17.15.3. RESET INSTRUCTION

The RESET instruction sets all status registers to zero.

INSTRUCTION

Instruction	Comment
reset	0 → SRN (N = 0, ..., 11)

Figure 160.

ENCODING

Field	Value
FUNC	11111b
AAIN	0
ABIN	0
MAIN	0
MBIN	1
BADD	0

Figure 161.

17.16. Load and Store Instructions

Since load/store portion of the code word is completely independent from the operation portion (except for load/store code/status registers), the fields that comprise the operation portion are not specified.

In the following, REG stands for (rreg, xyreg) or (rreg) or (xyreg). Also if "load" is prefixed by "y", the Y port is being used; if "store" is prefixed by "z", the Z port is being used; otherwise the specified load or store occurs through the X port.

Instruction		Comment
dload	REG	Load LS, MS of a double-precision operand*
dloadl	REG	Load LS of a double-precision operand
dloadm	REG	Load MS of a double-precision operand
fload	REG	Load single-precision openand (does not affect bits 31-0)
load	REG	Load integer (clears bits 63-53 and 20-0)
dstore	rreg	Store LS, MS of a double-precision operand*
dstorel	rreg	Store LS of a double-precision word
dstorem	rreg	Store MS of a double-precision word
fstore	rreg	Store single-precision operand
store	rreg	Store integer

Figure 162.

17.16. Load and Store Instructions, continued

The following tables summarize the effects of the I/O instructions. The columns in the tables have the following meanings:

int	Marked with an x if the data type of the operation is integer
float	Marked with an x if the data type of the operation is floating-point
m	Marked with an x if the move is the most-significant 32-bit word
l	Marked with an x if the move is the least-significant 32-bit word
.x	Marked with an x if input to the X register
.y	Marked with an x if input to the Y register
EFADD	Marked with an x if input to, or output from, the register file

Figure 163.

SR#	BIT#							
	7	6	5	4	3	2	1	0
SR0	Multiplier Latency	FPEX Sticky	0	0	Internal NEUT ON	Rounding Mode		Fast Mode
SR1	IEEE Software Underflow	Bypass on	FPEX Delay	I/O Mode				
SR2	NaN EN	INV EN	DVZ EN	DNRM Control	OVF EN	UNF Control	INX EN	IOVF EN
SR3	NaN	INV	DVZ	DNRM	OVF	UNF	INX	IOVF
SR4	0	TDEST0		MDEST0				
SR5	0	0	0	ADEST0				
SR6	ASTAT0			MSTAT0				
SR7	0	0	0	DIVDEST				
SR8	0	TDEST1		MDEST1				
SR9	0	0	0	ADEST1				
SR10	ASTAT1			MSTAT1				
SR11	FPEX Taken	DSR in progress	FPCN	Carry	DIVSTAT			

Figure 164. Status register map; instructions dload, dloadl, dloadm, fload, load, dstore, dstorel, dstorem, fstore, store

17.16. Load and Store Instructions, continued

17.16.1. CONFIGURATION A (THREE 32-BIT BUSES), SINGLE-PUMP

xcnt	int	float	m	l	.x	.y	EFADD	Instruction
0								nop
1		x		x			x	dstorel .fn
2		x	x				x	dstorem .fn
		x	x				x	fstore .fn
3	x						x	store .fn
4		x		x		x		dloadl .y
5		x		x	x		x	dloadl .fn, .x
6		x	x		x		x	dloadm .fn, .x
		x	x		x		x	fload .fn, .x
7	x				x		x	load .fn, .x
8		x	x			x		dloadm .y
		x	x			x		fload .y
9		x		x			x	floadl .fn
10		x	x				x	dloadm .fn
		x	x				x	fload .fn
11	x						x	load .fn
12	x					x		load .y
13		x		x	x			dloadl .x
14		x	x		x			dloadm .x
		x	x		x			fload .x
15	x				x			load .x
ycnt								
0								nop
1		x		x		x		dyloadl .y
2		x	x			x		dyloadm .y
		x	x			x		fyload .y
3	x					x		yload .y
zcnt								
0								nop
1		x		x			x	dzstorel .fn
2		x	x				x	dzstorem .fn
		x	x				x	fzstore .fn
3	x						x	zstore .fn

Note: loads occur through X or Y ports. Stores occur through the X or Z ports.

Figure 165.

EXAMPLES

```
dstorel .f0; dzstorem .f0
fstore  .f1
dloadm .y; yloadl .y
dloadm .f2, .x
```

Figure 166.

17.16. Load and Store Instructions, continued

17.16.2. CONFIGURATION A (THREE 32-BIT BUSES), DOUBLE-PUMP

xcnt	int	float	m	l	.x	.y	EFADD	Instruction
0								nop
1		x	x	x			x	dstore .fn
2		x	x				x	dstorem .fn
		x	x				x	fstore .fn
3	x						x	store .fn
4		x	x	x		x	x	dload .y
5		x	x	x	x		x	dload .fn, .x
6		x	x		x		x	dloadm .fn, .x
		x	x		x		x	fload .fn, .x
7	x				x		x	load .fn, .x
8		x	x			x		dloadm .y
		x	x			x		fload .y
9		x	x	x			x	fload .fn
10		x	x				x	dloadm .fn
		x	x				x	fload .fn
11	x						x	load .fn
12	x					x		load .y
13		x	x	x	x			dload .x
14		x	x		x			dloadm .x
		x	x		x			fload .x
15	x				x			load .x
ycnt								
0								nop
1		x	x	x		x		dyload .y
2		x	x			x		dyloadm .y
		x	x			x		yfload .y
3	x					x		yload .y
zcnt								
0								nop
1		x	x	x			x	dzstore .fn
2		x	x				x	dzstorem .fn
		x	x				x	fzstore .fn
3	x						x	zstore .fn

Note: loads occur through X or Y ports. Stores occur through the X or Z ports.

Figure 167.

EXAMPLES

```
dstore      .f3
dstorem     .f5
dload       .f9, .x
load        .x
```

Figure 168.

17.16. Load and Store Instructions, continued

17.16.3. CONFIGURATION B (SINGLE 64-BIT I/O BUS), SINGLE-PUMP ONLY

xcnt	ycnt	zcnt	int	float	m	l	.x	.y	EFADD	Instruction
0	0	0								nop
1	0	0		x	x	x			x	dstore .fn
2	0	0		x	x				x	dstorem .fn
	0	0		x	x				x	fstore .fn
3	0	0	x						x	store .fn
4	0	0		x	x	x		x		dload .y
5	0	0		x	x	x	x		x	dload .fn, .x
6	0	0		x	x		x		x	dloadm .fn, .x
	0	0		x	x		x		x	fload .fn, .x
7	0	0	x				x		x	load .fn, .x
8	0	0		x	x			x		dloadm .y
	0	0		x	x			x		fload .y
9	0	0		x	x	x			x	dload .fn
10	0	0		x	x				x	dloadm .fn
	0	0		x	x				x	fload .fn
11	0	0	x						x	load .fn
12	0	0	x					x		load .y
13	0	0		x	x	x	x			dload .x
14	0	0		x	x		x			dloadm .x
	0	0		x	x		x			fload .x
15	0	0	x				x			load .x

Note: stores occur through the X and Z, loads through the X and/or Y ports.

Figure 169.

EXAMPLES

```
dload   .f1, .x
fload   .x
dstore  .f0
```

Figure 170.

17.16. Load and Store Instructions, continued

17.16.4. CONFIGURATION C (SINGLE 32-BIT I/O BUS), SINGLE-PUMP

xcnt	int	float	m	l	.x	.y	EFADD	Instruction
0								nop
1		x		x			x	dstorel .fn
2		x	x				x	dstorem .fn
		x	x				x	fstore .fn
3	x						x	store .fn
4		x		x		x	x	dloadl .y
5		x		x	x		x	dloadl .fn, .x
6		x	x		x		x	dloadm .fn, .x
		x	x		x		x	fload .fn, .x
7	x				x		x	load .fn, .x
8		x	x			x		dloadm .y
		x	x			x		fload .y
9		x		x			x	dloadl .fn
10		x	x				x	dloadm .fn
		x	x				x	fload .fn
11	x						x	load .fn
12	x					x		load .y
13		x		x	x			dloadl .x
14		x	x		x			dloadm .x
		x	x	x	x			fload .x
15	x				x			load .x

Note: loads and stores occur through the X port.

Figure 171.

EXAMPLES

```
dloadl  .f0
dloadm .x
dloadm .f0, .x
fstore  .f0
```

Figure 172.

17.16. Load and Store Instructions, continued

17.16.5. CONFIGURATION C (SINGLE 32-BIT I/O BUS), DOUBLE-PUMP

xcnt	int	float	m	l	.x	.y	EFADD	Operation
0								nop
1		x	x	x			x	dstore .fn
2		x	x				x	dstorem .fn
	x	x					x	fstore .fn
3	x						x	store .fn
4		x	x	x		x		dload .y
5		x	x	x	x		x	dload .fn, .x
6		x	x		x		x	dloadm .fn, .x
		x	x		x		x	fload .fn, .x
7	x				x		x	load .fn, .x
8		x	x			x		dloadm .y
		x				x		fload .y
9		x	x	x			x	dload .fn
10		x	x				x	dloadm .fn
		x	x				x	fload .fn
11	x						x	load .fn
12	x					x		load .y
13		x	x	x	x			dload .x
14		x	x		x			dloadm .x
		x	x		x			fload .x
15	x				x			load .x

Note: loads and stores occur through the X port.

Figure 173.

EXAMPLES

```
dload   .f0, .x
dstore  .f0
fload   .y
```

Figure 174.

17.16. Load and Store Instructions, continued

17.16.6. LOAD/STORE STATUS REGISTERS

INSTRUCTIONS

Instruction		Comment
fsrsr	sreg	store status register
fldsr	sreg	load status register

Figure 175.

EXAMPLES

```
fstsr   .sr0
fldsr   .sr11
```

Figure 176.

ENCODING

FUNC	BADD	ABIN	MBIN	AAIN	MAIN	Operation	Comment
11111b	2	0	0	0	0	fstsr .sr0	$SR0_{7..0}$ → X/Z ports (31..24)
		0	0	0	1	fstsr .sr1	$SR1_{7..0}$ → X/Z ports (31..24)
		0	0	1	0	fstsr .sr2	$SR2_{7..0}$ → X/Z ports (31..24)
		0	0	1	1	fstsr .sr3	$SR3_{7..0}$ → X/Z ports (31..24)
		0	1	0	0	fstsr .sr4	$SR4_{7..0}$ → X/Z ports (31..24)
		0	1	0	1	fstsr .sr5	$SR5_{7..0}$ → X/Z ports (31..24)
		0	1	1	0	fstsr .sr6	$SR6_{7..0}$ → X/Z ports (31..24)
		0	1	1	1	fstsr .sr7	$SR7_{7..0}$ → X/Z ports (31..24)
		1	0	0	0	fstsr .sr8	$SR8_{7..0}$ → X/Z ports (31..24)
		1	0	0	1	fstsr .sr9	$SR9_{7..0}$ → X/Z ports (31..24)
		1	0	1	0	fstsr .sr10	$SR10_{7..0}$ → X/Z ports (31..24)
		1	0	1	1	fstsr .sr11	$SR11_{7..0}$ → X/Z ports (31..24)
	3	0	0	0	0	fldsr .sr0	X Port (31..24) → $SR0_{7..0}$
		0	0	0	1	fldsr .sr1	X Port (31..24) → $SR1_{7..0}$
		0	0	1	0	fldsr .sr2	X Port (31..24) → $SR2_{7..0}$
		0	0	1	1	fldsr .sr3	X Port (31..24) → $SR3_{7..0}$
		0	1	0	0	fldsr .sr4	X Port (31..24) → $SR4_{7..0}$
		0	1	0	1	fldsr .sr5	X Port (31..24) → $SR5_{7..0}$
		0	1	1	0	fldsr .sr6	X Port (31..24) → $SR6_{7..0}$
		0	1	1	1	fldsr .sr7	X Port (31..24) → $SR7_{7..0}$
		1	0	0	0	fldsr .sr8	X Port (31..24) → $SR8_{7..0}$
		1	0	0	1	fldsr .sr9	X Port (31..24) → $SR9_{7..0}$
		1	0	1	0	fldsr .sr10	X Port (31..24) → $SR10_{7..0}$
		1	0	1	1	fldsr .sr11	X Port (31..24) → $SR11_{7..0}$

Note: Bits $C_{12..0}$ must be set to zero.

Figure 177.

The instruction fldsr loads bits 31-24 from the data bus into the specified status register. The instruction fstsr writes the contents of the specified status register to bits 31-24 of the data bus. Neither instruction affects the code register.

17.16. Load and Store Instructions, continued

SR#	BIT#							
	7	6	5	4	3	2	1	0
SR0	Multiplier Latency	FPEX Sticky	0	0	Internal NEUT ON	Rounding Mode		Fast Mode
SR1	IEEE Software Underflow	Bypass on	FPEX Delay	I/O Mode				
SR2	NaN EN	INV EN	DVZ EN	DNRM Control	OVF EN	UNF Control	INX EN	IOVF EN
SR3	NaN	INV	DVZ	DNRM	OVF	UNF	INX	IOVF
SR4	0	TDEST0		MDEST0				
SR5	0	0	0	ADEST0				
SR6	ASTAT0			MSTAT0				
SR7	0	0	0	DIVDEST				
SR8	0	TDEST1		MDEST1				
SR9	0	0	0	ADEST1				
SR10	ASTAT1			MSTAT1				
SR11	FPEX Taken	DSR In progress	FPCN	Carry	DIVSTAT			

Figure 178. Status register map; instructions fldsr

17.16. Load and Store Instructions, continued

17.16.7. LOAD/STORE CODE REGISTERS

INSTRUCTIONS

Instruction		Comment
fstcr	creg	store code register
fldcr	creg	load code register

Figure 179.

EXAMPLES

```
fstcr .cr0
fldcr .cr5
```

Figure 180.

ENCODING

FUNC	BADD	ABIN	MBIN	AAIN	MAIN	Operation
11111b	0	0	0	0	0	nop
	1	1	0	0	0	fldcr .cr0
	1	1	0	0	1	fldcr .cr1
	1	1	0	1	0	fldcr .cr2
	1	1	0	1	1	fldcr .cr3
	1	1	1	0	0	fldcr .cr4
	1	1	1	0	1	fldcr .cr5
	1	0	0	0	0	fstcr .cr0
	1	0	0	0	1	fstcr .cr1
	1	0	0	1	0	fstcr .cr2
	1	0	0	1	1	fstcr .cr3
	1	0	1	0	0	fstcr .cr4
	1	0	1	0	1	fstcr .cr5

Note: Bits $C_{12..0}$ must be set to zero.

Figure 181.

Note: the code register should be stored before FPEX allows the code register to be overwritten by the next instruction. TAKEN bit ($SR11_7$) is cleared, because clearing this bit

17.16. Load and Store Instructions, continued

SR#	BIT#							
	7	6	5	4	3	2	1	0
SR0	Multiplier Latency	FPEX Sticky	0	0	Internal NEUT ON	Rounding Mode		Fast Mode
SR1	IEEE Software Underflow	Bypass on	FPEX Delay	I/O Mode				
SR2	NaN EN	INV EN	DVZ EN	DNRM Control	OVF EN	UNF Control	INX EN	IOVF EN
SR3	NaN	INV	DVZ	DNRM	OVF	UNF	INX	IOVF
SR4	0	TDEST0		MDEST0				
SR5	0	0	0	ADEST0				
SR6	ASTAT0				MSTAT0			
SR7	0	0	0	DIVDEST				
SR8	0	TDEST1		MDEST1				
SR9	0	0	0	ADEST1				
SR10	ASTAT1				MSTAT1			
SR11	FPEX Taken	DSR in progress	FPCN	Carry	DIVSTAT			

Figure 182. Status register map; instructions load/store code register instruction

17.17. Instruction Set Summary

In the following tables (figures 183 and 184), codes that are not listed are reserved and must not be used.

FUNC	AAIN*	ABIN†	MAIN*	MBIN†	Mnemonic	Operation	Description
0	.fa/.x	.fb/.y	.fa/.x	.fb/.y	fadd; fmul	.fc = x + y; .fd = x × y	32-bit floating add, mul
1	.fa/.x	.fb/.y	.fa/.x	.fb/.y	fsubr; fmul	.fc = -x + y; .fd = x × y	32-bit floating sub rev, mul
2	.fa/.x	.fb/.y	.fa/.x	.fb/.y	fsub; fmul	.fc = x - y; .fd = x × y	32-bit floating sub, mul
3	0	0	.fa/.x	.fb/.y	fmul	.fd = x × y	32-bit floating mul
4	.fa/.x	.fb/.y	0	0	fadd	.fc = x + y	32-bit floating add
			0	1	fsubr	.fc = -x + y	32-bit floating sub rev
			1	0	fsub	.fc = x - y	32-bit floating sub
5	0	0	.fa/.x	.fb/.y	fcmpeq	cond = (x = y)	32-bit floating compare =
	0	1	.fa/.x	.fb/.y	fcmp .gt	cond = (x > y)	32-bit floating compare >
	1	0	.fa/.x	.fb/.y	fcmp .lt	cond = (x < y)	32-bit floating compare <
	1	1	.fa/.x	.fb/.y	fcmp .uord	cond = (x = NaN OR y = NaN)	32-bit floating compare ?
7	0	0	.fa/.x	.fb/.y	fdiv	.fd = x ÷ y	32-bit floating divide
8	.fa/.x	.fb/.y	.fa/.x	.fb/.y	dfadd; dfmul	.fc = x + y; .fd = x × y	64-bit floating add, mul
9	.fa/.x	.fb/.y	.fa/.x	.fb/.y	dfsubr; dfmul	.fc = -x + y; .fd = x × y	64-bit floating sub rev, mul
10	.fa/.x	.fb/.y	.fa/.x	.fb/.y	dfsub; dfmul	.fc = x - y; .fd = x × y	64-bit floating sub, mul
11	0	0	.fa/.x	.fb/.y	dfmul	.fd = x × y	64-bit floating mul
12	.fa/.x	.fb/.y	0	0	dfadd	.fc = x + y	64-bit floating add
			0	1	dfsubr	.fc = -x + y	64-bit floating sub rev
			1	0	dsub	.fc = x - y	64-bit floating sub
13	0	0	.fa/.x	.fb/.y	dfcmp .eq	cond = (x = y)	64-bit floating compare =
	0	1	.fa/.x	.fb/.y	dfcmp .gt	cond = (x > y)	64-bit floating compare >
	1	0	.fa/.x	.fb/.y	dfcmp .lt	cond = (x < y)	64-bit floating compare <
	1	1	.fa/.x	.fb/.y	dfcmp .uord	cond = (x = NaN OR y = NaN)	64-bit floating compare ?
15	0	0	.fa/.x	.fb/.y	dfdiv	.fd = x ÷ y	64-bit floating divide

*AAIN and MAIN = 1 selects .fa, 0 selects .x
†ABIN and MBIN = 1 selects .fb, 0 selects .y

Note: To determine whether an instruction is executed in the ALU or in the multiplier, look at the destination address in the "operation" column. If it is .fc, the instruction is executed in the ALU; if it is .fd, the instruction is executed in the multiplier. Compare is a multiplier instruction.

Figure 183. Dyadic instructions

17.17. Instruction Set Summary, continued

FUNC	AAIN	ABIN	MAIN	MBIN	Mnemonic	Operation	Description
16*	.fa	.fb/.tn	y/x	.fb/fpass	fmul/fpass; fadd	.fd,.tn = x × .fb/fpass .fc = .fa + .fb/.tn;	32-bit floating chained mul, add
17*	.fa	.fb/.tn	y/x	.fb/fpass	fmul/fpass; fsubr	.fd, .tn = x × .fb/fpass -.fa + .fb/.tn;	32-bit floating chained mul, subr
18*	.fa	.fb/.tn	y/x	.fb/fpass	fmul/fpass; fsub	.fd, .tn = x × .fb/fpass .fc = .fa - .fb/.tn;	32-bit floating chained mul, sub
19*	.fa	.fb/.tn	y/x	.fb	ffdmul; dfadd	.fd, .tn = x × .fb .fc = .fa + .fb/.tn;	32 × 32 → 64-bit mul, 64-bit add
20*	.fa	.fb/.tn	y/x	.fb/dfpass	dfmul/dfpass; dfadd	.fd, .tn = x × .fb/dfpass .fc = .fa + .fb/.tn;	64-bit floating chained mul, add
21*	.fa	.fb/.tn	y/x	.fb/dfpass	dfmul/dfpass; dfsubr	.fd, .tn = x × .fb/dfpass .fc = -.fa + .fb/.tn;	64-bit floating chained mul, subr
22*	.fa	.fb/.tn	y/x	.fb/dfpass	dfmul/dfpass; dfsub	.fd, .tn = x × .fb/dfpass .fc = .fa - .fb/.tn;	64-bit floating chained mul, sub
23*	.fa	.fb/.tn	y/x	.fb	fddmul; dfadd	.fd, .tn = x × .fb .fc = .fa + .fb/.tn;	32 × 64 → 64-bit mul, 64-bit add
24**	0	0	.fa/x	.fb/y	faddi	.fd = x + y	32-bit integer add
	0	1	.fa/x	.fb/y	fsubir	.fd = -x + y	32-bit integer sub rev
	1	0	.fa/x	.fb/y	faddic .x .y .fd	.fd = x + y + carry	32-bit integer add with carry
	1	1	.fa/x	.fb/y	fsubirc .x .y .fd	.fd = -x + y -1 + carry	32-bit integer sub rev with carry
25**	0	0	.fa/x	.fb/y	and	.fd = x and y	64-bit logical bitwise "and"
	0	1	.fa/x	.fb/y	or	.fd = x or y	64-bit logical bitwise "or"
	1	0	.fa/x	.fb/y	xor	.fd = x xor y	64-bit logical bitwise "xor"
	1	1	0	.fb/y	mov .y	.fd = y	64-bit logical pass y operand
26**	0	0	.fa/x	.fb/y	fmulil	.fd = x × y (ls)	32-bit integer mul, low word
27**	0	0	.fa/x	.fb/y	fmulim	.fd = x × y (ms)	32-bit integer mul, high word
29**	0	0	.fa/x	.fb/y	min	.fd = min (x,y)	Return smaller of two operands
	0	1	.fa/x	.fb/y	max	.fd = max (x,y)	Return larger of two operands
31**	-	-	-	-	-	-	Monadic operations

* AAIN ABIN

AAIN	ABIN	
0	0	Select .t0 for mul output and ALU input
1	0	Select .t1 for mul output and ALU input
0	1	Don't load .t0 or .t1; select .fb for ALU input

MAIN = 1 selects .y, 0 selects .x
MBIN = 1 selects .fb, 0 selects fpass or invalid

** AAIN = 1 selects .fa, 0 selects .x
ABIN = 1 selects .fb, 0 selects .y
MAIN = 1 selects .fa, 0 selects .x
MBIN = 1 selects .fb, 0 selects .y

Figure 183. Dyadic instructions, continued

17.17. Instruction Set Summary, continued

BADD	ABIN	MBIN	AAIN	MAIN*	Mnemonic	Operation	Description
0	0	0	0	0	nop		No operation
0	0	1	0	0	reset	0 → SRN (N = 0, ..., 11)	Clear all status registers
0	1	0	0	0	excr	instr = CR	Re-execute code register
1	0	0	0	0	.fstcr .cr0	X port = CR byte 0	Store code reg byte 0
	0	0	0	1	.fstcr .cr1	X port = CR byte 1	Store code reg byte 1
	0	0	1	0	.fstcr .cr2	X port = CR byte 2	Store code reg byte 2
	0	0	1	1	.fstcr .cr3	X port = CR byte 3	Store code reg byte 3
	0	1	0	0	.fstcr .cr4	X port = CR byte 4	Store code reg byte 4
	0	1	0	1	.fstcr .cr5	X port = CR byte 5	Store code reg byte 5
	1	0	0	0	.fldcr .cr0	CR byte 0 = X port	Load code reg byte 0
	1	0	0	1	.fldcr .cr1	CR byte 1 = X port	Load code reg byte 1
	1	0	1	0	.fldcr .cr2	CR byte 2 = X port	Load code reg byte 2
	1	0	1	1	.fldcr .cr3	CR byte 3 = X port	Load code reg byte 3
	1	1	0	0	.fldcr .cr4	CR byte 4 = X port	Load code reg byte 4
	1	1	0	1	.fldcr .cr5	CR byte 5 = X port	Load code reg byte 5
2	0	0	0	0	fstsr .sr0	X port = sr0	Store status reg SR0
	0	0	0	1	fstsr .sr1	X port = sr1	Store status reg SR1
	0	0	1	0	fstsr .sr2	X port = sr2	Store status reg SR2
	0	0	1	1	fstsr .sr3	X port = sr3	Store status reg SR3
	0	1	0	0	fstsr .sr4	X port = sr4	Store status reg SR4
	0	1	0	1	fstsr .sr5	X port = sr5	Store status reg SR5
	0	1	1	0	fstsr .sr6	X port = sr6	Store status reg SR6
	0	1	1	1	fstsr .sr7	X port = sr7	Store status reg SR7
	1	0	0	0	fstsr .sr8	X port = sr8	Store status reg SR8
	1	0	0	1	fstsr .sr9	X port = sr9	Store status reg SR9
	1	0	1	0	fstsr .sr10	X port = sr10	Store status reg SR10
	1	0	1	1	fstsr .sr11	X port = sr11	Store status reg SR11
3	0	0	0	0	fldsr .sr0	sr0 byte 0 = X port	Load status reg SR0
	0	0	0	1	fldsr .sr1	sr1 byte 1 = X port	Load status reg SR1
	0	0	1	0	fldsr .sr2	sr2 byte 2 = X port	Load status reg SR2
	0	0	1	1	fldsr .sr3	sr3 byte 3 = X port	Load status reg SR3
	0	1	0	0	fldsr .sr4	sr4 byte 0 = X port	Load status reg SR4
	0	1	0	1	fldsr .sr5	sr5 byte 1 = X port	Load status reg SR5
	0	1	1	0	fldsr .sr6	sr6 byte 2 = X port	Load status reg SR6
	0	1	1	1	fldsr .sr7	sr7 byte 3 = X port	Load status reg SR7
	1	0	0	0	fldsr .sr8	sr8 byte 0 = X port	Load status reg SR8
	1	0	0	1	fldsr .sr9	sr9 byte 1 = X port	Load status reg SR9
	1	0	1	0	fldsr .sr10	sr10 byte 2 = X port	Load status reg SR10
	1	0	1	1	fldsr .sr11	sr11 byte 3 = X port	Load status reg SR11
4	0	0	0	.fa/.x	dfsqrt	.fd = dsqrt(x)	64-bit floating square root
	1	0	0	.fa/.x	fsqrt	.fd = sqrt(x)	32-bit floating square root
5	0	0	0	.fa/.x	fabsi	.fd = abs(x)	Integer absolute value
	0	1	0	.fa/.x	fnegi	.fd = -x	Integer negate
	1	0	0	.fa/.x	faddic .x .fd	.fd = x + carry	Integer add carry
	1	1	0	.fa/.x	fnegib	.fd = -x - 1 + carry	Integer negate with borrow
6	0	0	0	.fa/.x	dfpass, dfmov	.fd = x	64-bit floating pass (IEEE "class")
	0	1	0	.fa/.x	dfabs	.fd = dabs(x)	64-bit floating absolute value
	1	0	0	.fa/.x	fpass, fmov	.fd = x	32-bit floating pass (IEEE "class")
	1	1	0	.fa/.x	fabs	.fd = fabs(x)	32-bit floating absolute value

For all monadic instructions, FUNC field is 1f(hex)

*MAIN = 1 selects .fa, 0 selects .x

Figure 184. Monadic instructions

17.17. Instruction Set Summary, continued

BADD	ABIN	MBIN	AAIN*	MAIN*	Mnemonic	Operation	Description
7	0	0	0	0	clr	.fd = 0	Clear register†
	0	1	0	.fa/.x	dneg, neg	.fd =-x	32/64-bit negate (invert sign)†
	1	0	0	.fa/.x	not	.fd = x'	64-bit logical negate†
	1	1	0	.fa/.x	mov .x	.fd = x	64-bit logical pass x operand†
8	0	0	.fa/.x	0	dfixr	.fc = int(x) (round)	Convert 64-bit float to int, round
	0	1	.fa/.x	0	dfix	.fc = int(x) (truncate)	Convert 64-bit float to int, truncate
	1	0	.fa/.x	0	fixr	.fc = int(x) (round)	Convert 32-bit float to int, round
	1	1	.fa/.x	0	fix	.fc = int(x) (truncate)	Convert 32-bit float to int, truncate
9	1	0	.fa/.x	0	dfloat	.fc = double(x)	Convert int to 64-bit float
	1	1	.fa/.x	0	float	.fc = float(x) (truncate)	Convert int to 32-bit float, truncate
10	0	0	.fa/.x	0	dfcnvt, dcnvtf	.fc = float(x) (round)	Convert 64-bit float to 32-bit float
	1	0	.fa/.x	0	fdcnvt, fcnvtd	.fc = double(x)	Convert 32-bit float to 64-bit float
12	0	0	.fa/.x	0	sll	.fc = x << 1	Shift left 1 bit, 0 → lsb
	0	1	.fa/.x	0	scl	.fc = msb(x) + (x << 1)	Rotate left 1 bit
	1	0	.fa/.x	0	slr	.fc = x >> 1	Shift right 1 bit, 0 → msb
	1	1	.fa/.x	0	sar	.fc = sign(x) and (x >> 1)	Arithmetic shift right (sign extend) 1 bit
14	0	0	.fa/.x	0	dfude	.fc = unwrap(x) (exact)	Convert 64-bit Unrm to Dnrm, exact
	0	1	.fa/.x	0	dfudi	.fc = unwrap(x) (inexact)	Convert 64-bit Unrm to Dnrm, inexact
	1	0	.fa/.x	0	fude	.fc = unwrap(x) (exact)	Convert 32-bit Unrm to Dnrm, exact
	1	1	.fa/.x	0	fudi	.fc = unwrap(x) (inexact)	Convert 32-bit Unrm to Dnrm, inexact
15	0	0	.fa/.x	0	dfdw	.fc = dwrap(x)	Convert 64-bit Dnrm to Unrm
	1	0	.fa/.x	0	fdw	.fc = fwrap(x)	Convert 32-bit Dnrm to Unrm

For all monadic instructions, FUNC field is 1f(hex)

*AAIN and MAIN = 1 selects .fa, 0 selects .x

†These instructions are logical. They do not produce floating-point exceptions or set the sticky flags. They may not be bypassed into floating-point operations.

Figure 184. Monadic instructions, continued

18. Initialization

When the 3x64 is first powered up, its state is undefined. The state of the file registers, and more significantly, that of the status registers is undefined. Specifically, the I/O mode in effect at power up is undefined. In order to be able to load the register file and the status registers with known values, it is first necessary to set the desired I/O mode. The following sequence of operations is one suggested way of initializing the 3x64.

1. Clear the status registers using the reset instruction; this instruction sets all bits in all status registers to zero.
2. Set I/O modes in status register ($SR1_{4..0}$), see section 5.5.1

   a. The reset instruction above selects a known I/O mode ($SR1_{4..0} = 0$), namely single-pump undelayed load and single-pump undelayed store

   b. Given this known I/O mode, it is now possible to set the desired I/O mode (if different)
3. Set all other modes in status registers SR0, SR1, and trap enables in SR2
4. Since the pipelines may contain random values and since the DSR is being clocked by DIVCLK, while the above operations are taking place, it is important to wait for the DSR to empty and write its result to the register file and to potentially update the status register. See section 9 for latencies of DSR operations.
5. Clear temporary latches (using chained multiply-add instructions), to avoid spurious exceptions later.
6. Clear SR4 through SR1·1, to remove the effects of arithmetic units having potentially updated them with meaningless values.
7. Optionally load the register file and the X and Y registers with desired values.

19. IEEE Considerations

This chapter deals with issues arising from having to comply with the ANSI/IEEE Std 754-1985, the *IEEE Standard For Binary Floating-Point Arithmetic.* The standard defines terms used and specifies data formats, rounding options, operations, exceptions, and traps.

19.1. Definitions

In this section, if the text is enclosed in quotes, this text is a quote form the standard.

True exponent

"The component of a binary floating-point number that normally signifies the integer power to which 2 is raised in determining the value of the represented number."

Biased exponent

"The sum of the exponent and a constant (bias) chosen to make the biased exponent's range nonnegative." In the text below whenever the word "exponent" is used alone, it means "biased exponent."

Significand

The component of a binary floating-point number that consists of an explicit or implicit leading bit to the left of its implied binary point and a fraction field to the right."

Fraction

"The field of the significand that lies to the right of its implied binary point."

Binary floating-point number

"A bit string characterized by three components: a sign, a signed exponent and a significand. Its numerical value, if any, is the signed product of its significand and two raised to the power of its exponent."

Denormalized number

"A nonzero floating-point number whose exponent has a reserved value, usually the format's minimum, and whose explicit or implicit leading significand bit is zero." Denormalized numbers are also referred to as subnormalized numbers or subnormals.

NaN

"Not a number, a symbolic entity encoded in floating-point format."

19.2. Data Formats

Since the standard does not specify precisely certain parameters of the single extended or the double extended formats, leaving their definition up to a particular implementation, the 3x64 does not support these extended formats.

The 32-bit integer format supported by the 3x64 is not part of the standard.

19.2.1. SINGLE-PRECISION FLOATING-POINT FORMAT

The IEEE single-precision floating-point word is 32 bits wide and consists of three fields: a single-bit sign **s**, an eight-bit biased exponent **e**, and a 23-bit fraction **f**.

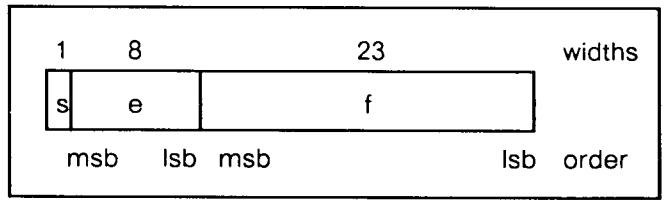


Figure 185. Single-precision floating-point format

The standard defines values of single-precision floating-point numbers according to the following conventions:

If e = 0 and f = 0,
then value V = $(-1)^s$ (0) (+0, -0)

Note that the IEEE standard has two representations of the value zero: one negative, the other positive.

If e = 0 and f ≠ 0,
then value V = DNRM Denormalized number

If 0 < e < 255,
then value V = $(-1)^s(2^{e-127})$ (1.f)
Normalized number

Note that 1.f above is the significand. The one to the left of the binary point is the so called "hidden" bit. This bit is not stored as part of the floating-point word; it is implied. For a number to be normalized it must have this one to the left of the binary point.

If e = 255 and f = 0,
then value V = $(-1)^s$ (∞) (+∞, -∞)

If e = 255 and f ≠ 0,
then value V = NaN Not-a-number

19.2.2. DOUBLE-PRECISION FLOATING-POINT NUMBER

The IEEE double-precision floating-point word is 64 bits wide an consists of three fields: a single-bit sign s, and eleven-bit biased exponent e, and a 52-bit fraction f.

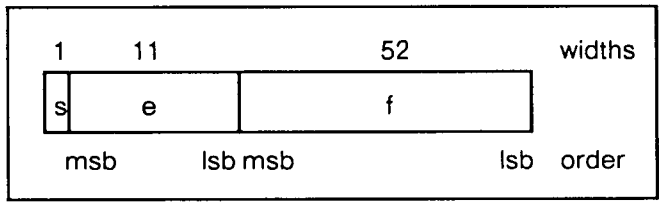


Figure 186. Double-precision floating-point format

The standard defines values of double-precision floating-point numbers according to the following conventions:

If e = 0 and f = 0,
then value V = $(-1)^s$ (0) (+0, -0)

If e = 0 and f ≠ 0,
then value V = DNRM Denormalized number

If 0 < e < 2047,
then value V = $(-1)^s(2^{e-1023})$ (1.f)
Normalized number

If e = 2047 and f = 0,
then value V = $(-1)^s$ (∞) (+∞, -∞)

If e = 2047 and f ≠ 0,
then value V = NaN Not-a-number

19.2.3. INTEGER FORMAT

The IEEE standard does not specify an integer format. The 3x64, however, does support operations on 32-bit two's complement integers. The format is given in figure 187.

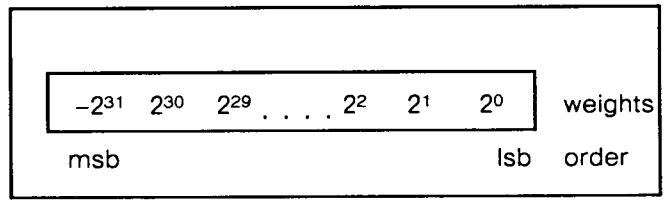


Figure 187. 32-bit integer format

19.3. Number Types Used to Implement the IEEE Standard

19.3.1. NORMALIZED NUMBERS (NRM).

Most calculations are performed on normalized numbers. For single precision, they have an exponent range from $0000\ 0001_2 = 1_{10}$ to $1111\ 1110_2 = 254_{10}$ and a normalized significand (hidden bit is 1.)

19.3.2. ZERO

The IEEE zero has all fields except the sign field—the exponent, fraction, and the hidden bit—equal to zero. The sign bit determines the sign of zero.

19.3.3. INFINITY

Infinity has a maximum exponent (255 for single precision, 2047 for double precision) and a zero fraction. The sign bit determines the sign of the infinity.

19.3.4. DENORMALIZED NUMBERS

Denormalized numbers are those numbers whose magnitude is smaller than the smallest magnitude representable in the format. They have a zero exponent and a denormalized nonzero fraction. Denormalized fraction means that the hidden bit is zero.

The functional units in the 3x64 cannot directly operate on denormalized operands. Denormalized operands have to be converted to wrapped normalized format (WNRM) before they can be used in the arithmetic units.

19.3.5. WRAPPED NORMALIZED NUMBERS (WNRM)

A wrapped number is created, using the ALU *wrap* instruction (see section 17.10), by normalizing the fraction field of a denormalized number and subtracting from the exponent the shift count. Normalization is accomplished by left-shifting the fraction until the hidden bit is a 1. The value of the wrapped exponent is equal to [1 – the shift count] and is represented in two's complement.

19.3.6. UNROUNDED NORMALIZED NUMBERS (UNRM)

A UNRM is a result of an operation that has magnitude less than the minimum representable normalized number. A UNRM has a fraction field, a wrapped exponent, and a hidden bit of one. The minimum UNRM is the result of the multiplication of two minimum denormalized numbers. UNRMs are turned into DNRMs using the ALU's *unwrap* operations (see section 17.10).

19.3.7. NOT-A-NUMBER (NaN)

NaNs do not represent numerical values but are interpreted as signals or symbols. They are used to signal invalid operations and as a way of passing status information through a series of calculations. NaNs arise in one of two ways: they can be generated by the 3x64 upon an invalid operation or they may be supplied by the user as an input operand.

The NaN generated by the 3x64 (the default NaN), whether single- or double-precision has a zero sign bit, an exponent of all 1's, and a fraction of all 1's.

Upon an *fpass* operation, an operand NaN at the input is passed to the result unchanged.

Upon an *fabs* operation, an operand NaN is preserved in the result with the sign of 0.

Logical operations treat NaNs as they would any other string, i.e. as a logical entity.

19.3.8. SINGLE-PRECISION FORMATS SUPPORTED BY THE 3x64

The table below is a summary of each number type supported by the 3x64 in order to implement the IEEE standard. A similar table for double precision can be created by extension.

19.3. Number Types Used to Implement the IEEE Standard, continued

Operand	Biased Exponent	Fraction	Hidden Bit	Value
Nan	255	$\neq 0$	x	None
Infinity	255	0	x	$(-1)^s(\infty)$
Zero	0	0	0	$(-1)^s(\infty)$
NRM.MAX	254	11...11	1	$(-1)^s(2^{127})\ (2 - 2^{-23})$
NRM.MIN	1	00...00	1	$(-1)^s(2^{-126})$
NRM	1 to 254	Any	1	$(-1)^s(2^{e-127})\ (1.f)$
DNRM.MAX	0	11...11	0	$(-1)^s(2^{-126})\ (1 - 2^{-23})$
DNRM.MIN	0	00...01	0	$(-1)^s(2^{-149})$
DNRM	0	Any	0	$(-1)^s(2^{-126})\ (0.f)$
WNRM.MAX	0	11...11	1	$(-1)^s(2^{-127})\ (2 - 2^{-22})$
WNRM.MIN	-22	00...00	1	$(-1)^s(2^{-149})$
WNRM	0 to -22	Any	1	$(-1)^s(2^{e-127})\ (1.f)$
UNRM.MAX	0	11...11	1	$(-1)^s(2^{-127})\ (2 - 2^{-23})$
UNRM.MIN	-171	00...00	1	$(-1)^s(2^{-298})$

Figure 188.

19.4. Rounding

"Rounding takes a number regarded as infinitely precise and, if necessary, modifies it to fit in the destination's format while signaling the inexact exception."

The 3x64 supports all rounding options (see section 12.3.2 for encoding). Directed rounding options—round toward plus or minus infinity and round to zero—statistically introduce a small bias in the direction of the rounding. Round to nearest introduces no statistical bias.

Round to nearest is the standard's default. The representable value nearest the infinitely precise result is delivered. If the two nearest representable values are equally near, the one with the LSB of zero (i.e. the even number) is delivered.

19.5. Exceptions

The 3x64 generates all five exceptions specified by the standard.

Invalid operation	INV
Division by zero	DVZ
Overflow, floating-point	OVF
Overflow, integer	IOVF
Underflow	UNF
Inexact	INX

Figure 189.

Exception information is signaled on the $S_{3..0}$ pins (see section 11) or it can be obtained from the Status Register (see section 12).

19.5.1. INVALID OPERATION (INV)

"The invalid operation exception is signaled if an operand is invalid for the operation to be performed." The result is a NaN," provided the destination has a floating-point format." The 3x64 will generate the invalid operaion exception for the following operations:

An operation involving a NaN operand (see section 19.3.7).	
Magnitude subtraction of infinities	$(+\infty) + (-\infty)$
Multiplication	$0 \times \infty$
Division	$0/0$ or ∞/∞
Square root of a negative argument*	
Comparison by way of predicates involving > or <, without ? (unordered) when the operands are unordered	

* Note that SQRT (-0) does not generate an exception. The result is specified by the standard as -0.

Figure 190.

19.5.2. DIVISION BY ZERO (DVZ)

If the divisor is zero and the dividend is a finite nonzero number, then the division by zero exception is signaled. The result is a correctly rounded infinity.

19.5.3. OVERFLOW (OFV, IOVF)

"The overflow exception is signaled whenever the destination format's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result were the exponent range unbounded." The result is either positive or negative infinity or the largest positive or negative finite (i.e normalized) number. This result is determined by the rounding mode and the sign of the intermediate result as follows:

1. "Round to nearest carries all overflows to ∞ with the sign of the intermediate result."
2. "Round toward zero carries all overflows to the format's largest finite number MAX.NRM with the sign of the intermediate result."
3. "Round toward $-\infty$ carries positive overflows to the format's largest positive finite number dMAX.NRMf, and carries negative overflows to $-\infty$."
4. "Round toward $+\infty$ carries negative overflows to the format's most negative finite number dMAX.NRMf, and carries positive overflows to $+\infty$."

Overflow is also generated upon a floating-point to integer conversion when the result overflows the 32-bit integer format (integer overflow, IOVF).

19.5.4. UNDERFLOW (UNF)

Underflow is generated when the magnitude of an operation's result after rounding is less than the smallest representable finite number (NRM.MIN). A result of exactly zero does not underflow.

19.5.5. INEXACT (INX)

The inexact exception is generated whenever there is a loss of accuracy (or significance) in the result. The 3x64 computes results to higher precision than the number of fraction bits in the format. If any of the fraction bits to the right of the LSB was one prior to rounding, the INX exception is signaled.

20. Glossary

This section defines potentially confusing or ambiguous terms.

CURRENT VS. NEXT CYCLE; CURRENT VS. NEXT INSTRUCTION

Where possible, instructions and corresponding clock cycles are identified by a number, and there is a one-to-one relationship between these numbers. For example, instruction C1 is clocked on the rising edge of cycle 1, instruction C3 — on the rising edge of cycle 3. Where instruction and clock cycle numbers are not used to describe instructions, this document uses the terms "current" and "next" to identify instructions at the code inputs relative to some control input as defined in the diagram below.

STICKY BITS AND STICKY OUTPUT SIGNALS

In this document, the term "sticky bits" is used to refer to any status register bit that, once set as a result of an operation, remains set until explicitly cleared. In particular, all floating-point exception bits are sticky. An output signal is sticky, if, once asserted as a result of an operation, it remains asserted until explicitly caused to be de-asserted.

LATENCY

The term "latency" is used in this document in two equivalent ways: with respect to data and with respect to instructions. In terms of data, latency is measured from the rising edge that clocks input data into a WTL 3x64 input port to the rising edge that clocks the result into a register outside the 3x64. In terms of instructions, latency is measured from the rising edge that clocks the first instruction into the code port to the rising edge that clocks in the next instruction which can use the result of the first instruction as one of its operands. The unit of measurement is either clock cycles or time (nanoseconds); the exact meaning is established by the context.

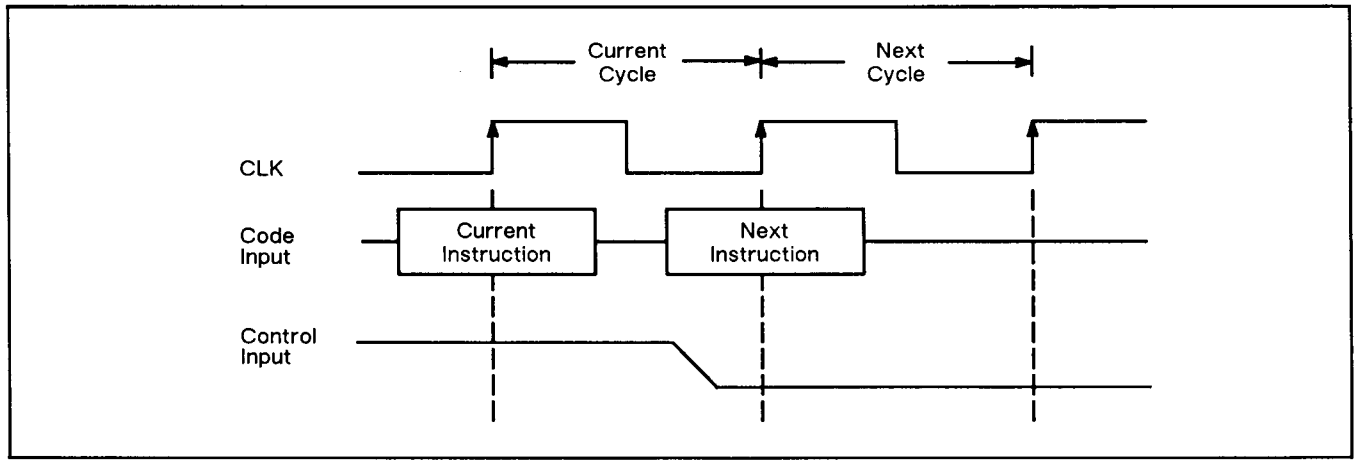


Figure 191.

November 1989

21. Specifications

Supply voltage	-0.5 to 7.0 V
Input voltage	-0.5V to VCC
Output voltage	-0.5V to VCC
Operating temperature range (T_{CASE})	0° to 85° C
Storage temperature range	-65° C to 150° C
Lead temperature (10 seconds)	300° C
Junction temperature	155° C

Figure 192. Absolute maximum ratings

PARAMETER		MIN	MAX	UNIT
V_{CC}	Supply voltage	4.75	5.25	V
I_{OH}	High-level output current		-0.4	mA
I_{OL}	Low-level output current		4.0	mA
T_{CASE}	Operating case temperature	0	85	°C

Figure 193. Operating conditions

21.1. DC Specifications

PARAMETER		TEST CONDITIONS	MIN	MAX	UNITS
V_{IH}	High-level input voltage	V_{CC} = MAX	2.0		V
V_{IL}	Low-level input voltage	V_{CC} = MIN		0.8	V
V_{IHC}	High-level clock and divide clock input voltage	V_{CC} = MAX	2.4		V
V_{ILC}	Low-level clock and divide clock input voltage	V_{CC} = MIN		0.8	V
V_{OH}	High-level output voltage	V_{CC} = MIN, I_{OH} = MAX	2.4		V
V_{OL}	Low-level output voltage	V_{CC} = MIN, I_{OL} = MAX		0.4	V
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} = 0		10	µA
I_{OZL}	Tri-state leakage current low	V_{CC} = MAX, V_{IN} = 0		10	µA
I_{OZH}	Tri-state leakage current high	V_{CC} = MAX, V_{IN} = 0		10	µA
I_{CC}	Supply current	V_{CC} = MAX, T_{CY} = MIN		400	mA
C_{IN}*	Input capacitance			15	pF
C_{CLK}*	Clock input capacitance			50	pF
C_{OUT}*	Output capacitance			15	pF

* Characterized at 1 MHz. Not tested.

Figure 194. DC specifications

21.2. AC Specifications and Timing Diagrams

DESCRIPTION		FINAL SPECIFICATIONS						ADVANCE SPECIFICATIONS			UNIT
		3164-100 3364-100		3164-75 3364-75		3164-60 3364-60		3164-50 3364-50			
		Either two- or three-cycle latency mode						Either two- or three-cycle latency mode		NOTES	
		MIN	MAX	MIN	MAX	MIN	MAX	MIN	MAX		
T_{CY}	CLK cycle time	100		75		60		50			ns
T_{CH}	CLK HIGH time	45	DC	34	DC	28	DC	21	DC		ns
T_{CL}	CLK LOW time	45	DC	34	DC	28	DC	23	DC		ns
T_{DCY}	DIVCLK cycle time	50		37.5		30		25			ns
T_{DCH}	DIVCLK HIGH time	22		16		13		11			ns
T_{DCL}	DIVCLK LOW time	22		16		13		11			ns
T_{CR}	CLK and DIVCLK rise time		4		4		4		4	See Note 8	ns
T_{CF}	CLK and DIVCLK fall time		4		4		4		4	See Note 8	ns
T_{DC}	DIVCLK to CLK skew (DIVCLK before CLK)	0	10	0	10	0	9	0	9	See Note 8	ns
T_{S1}	Setup time (from rising edge)	15		15		15		10			ns
T_{S2}	Setup time (from falling edge)	15		15		15		10			ns
T_{S3}	Setup time for NEUT-, STALL-, ABORT- inputs	15		10		10		10			ns
T_{H1}	Hold time for all inputs other than NEUT-, STALL-, ABORT-	3		3		3		3			ns
T_{H2}	Hold time for NEUT-, STALL-, ABORT- inputs	5		5		5		5			ns
Output delays (X and Z buses)											
	Single-pump										
T_{D1}	Undelayed store		85		60		55		46		ns
T_{D2}	Delayed store		85		60		55		46		ns
T_{D3}	Delayed data store		30		30		25		21		ns
	Double-pump										
$T_{D4A/B}$	Delayed data store from two successive CLK edges		85/30		60/30		55/25		Note 7	See Note 1	ns
T_{ZO}	Instruction-driven minimum output enable times (X & Z buses)	5		5		5		5		See Note 8	ns
T_{VO}	Output valid time	5		5		5		5			ns

Figure 195. AC specifications: guaranteed switching characteristics over commercial temperature range and operating conditions (continued on next page)

21.2. AC Specifications and Timing Diagrams, continued

DESCRIPTION	FINAL SPECIFICATIONS						ADVANCE SPECIFICATIONS			UNIT
	3164-100 3364-100		3164-75 3364-75		3164-60 3364-60		3164-50 3364-50			
	Either two- or three-cycle latency mode						Either two- or three-cycle latency mode		NOTES	
	MIN	MAX	MIN	MAX	MIN	MAX	MIN	MAX		
Instruction-driven minimum output disable times (X and Z buses)										
Single-pump										
T_{OZ1} Undelayed store		40		40		35		30	See Note 8	ns
T_{OZ2} Delayed store		40		40		35		30	See Note 8	ns
T_{OZ3} Delayed data store		30		30		25		21	See Note 8	ns
Double-pump										
T_{OZ4} Delayed data store		30		30		25		Note 7	See Note 8	ns
Output delays—other outputs										
T_{D5} FPEX- delayed, single- or double-pump		30		30		25		23		ns
T_{D6} FPEX- undelayed, single-pump only		85		65		54		Note 2		ns
T_{D7} FPCN		30		30		25		23		ns
$T_{D8A/B}$ Status (from two successive CLK edges)		85/30		65/30		54/25		Note 3/ Note 7	See Note 1	ns
T_{ENA} Tri-state bus enable time Output Enable (OEX-, OEZ-) transition to bus valid		20		20		20		20		ns
T_{DIS} Tri-state bus disable time Output Enable (OEX-, OEZ-) transition to bus disabled		20		20		20		20		ns

Notes:
1. Use the worst (i.e. the latest) of the T_{D4A}/T_{D8A} after the rising edge or the T_{D4B}/T_{D8B} after the falling edge of the CLK.
2. FPEX undelayed is not available in this speed grade.
3. The status comes out on the $S_{3..0}$ pins in two phases (see figures 58 and 197).
4. Worst case over time and temperature range.
5. Input levels are 0.4 V and 3.5 V unless otherwise specified.
6. Timing transitions are measured at 1.5 V unless otherwise noted.
7. Not available in this speed grade. See section 21.4.
8. This parameter is guaranteed, but is not tested.

AC TEST CONDITIONS (Notes 4, 5, and 6)		
$V_{CC} = 5V \pm 5\%$	$V_{IH} = 3.5V$ $V_{IL} = 0.4V$	$V_{OH} = 2.8V$, $I_{OH} = -1.0$ mA $V_{OL} = 0.4V$, $I_{OL} = 4.0$ mA	$T_{CASE} = 0 - 85\,^{\circ}C$

Figure 196. AC specifications: guaranteed switching characteristics over commercial temperature range and operating conditions

21.2. AC Specifications and Timing Diagrams, continued

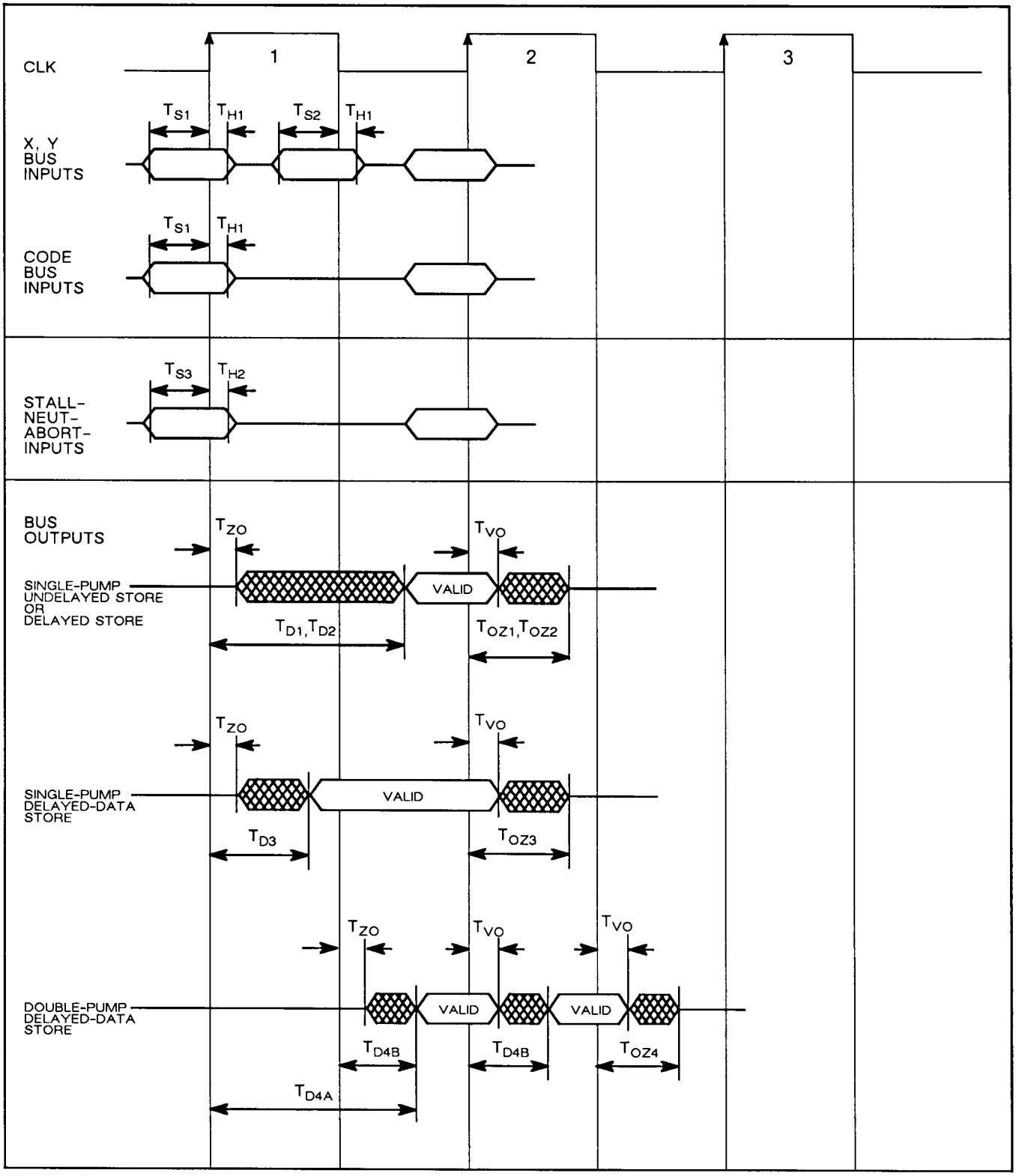


Figure 197. AC timing (continued on next page)

21.2. AC Specifications and Timing Diagrams, continued

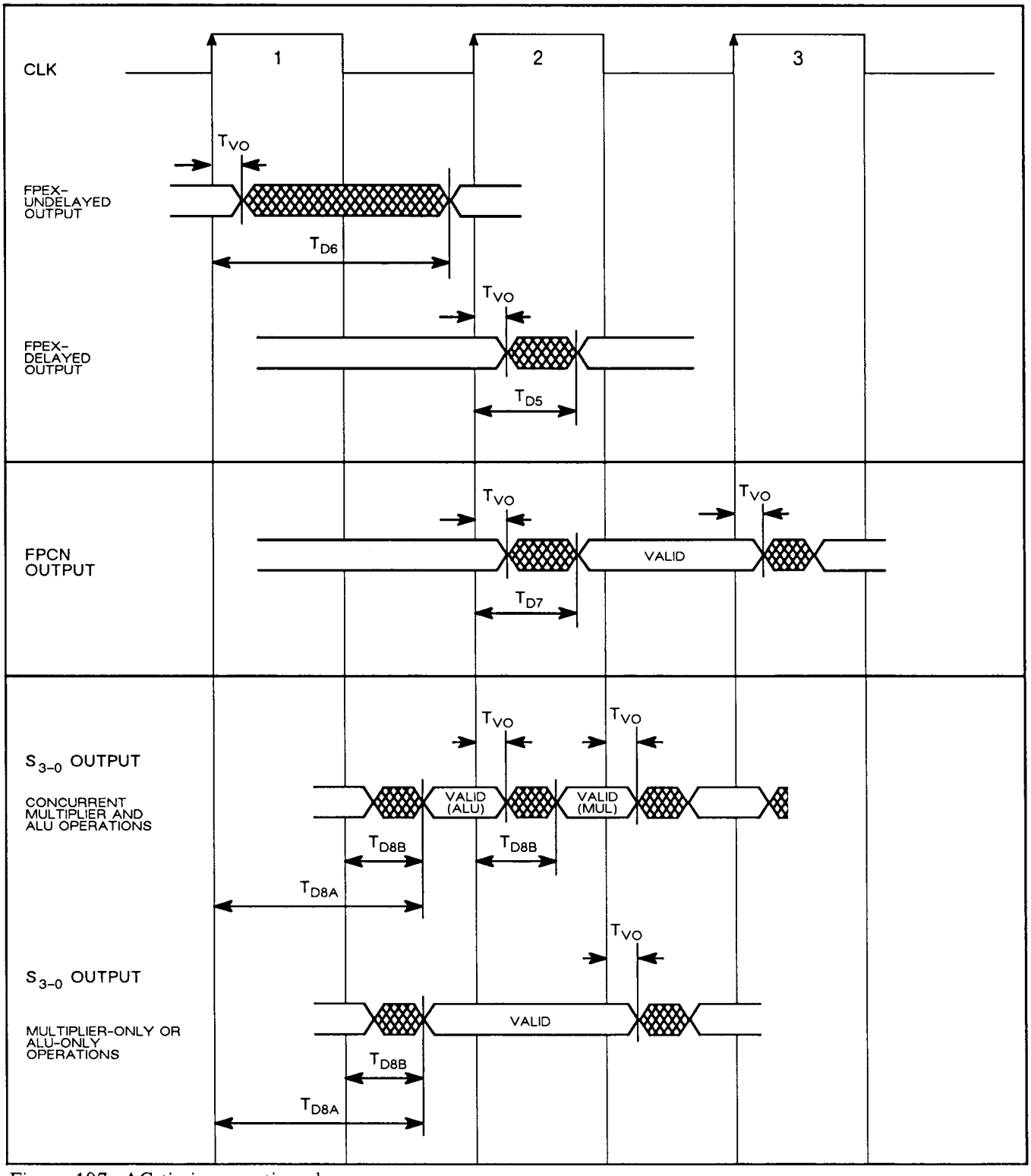


Figure 197. AC timing, continued

21.2. AC Specifications and Timing Diagrams, continued

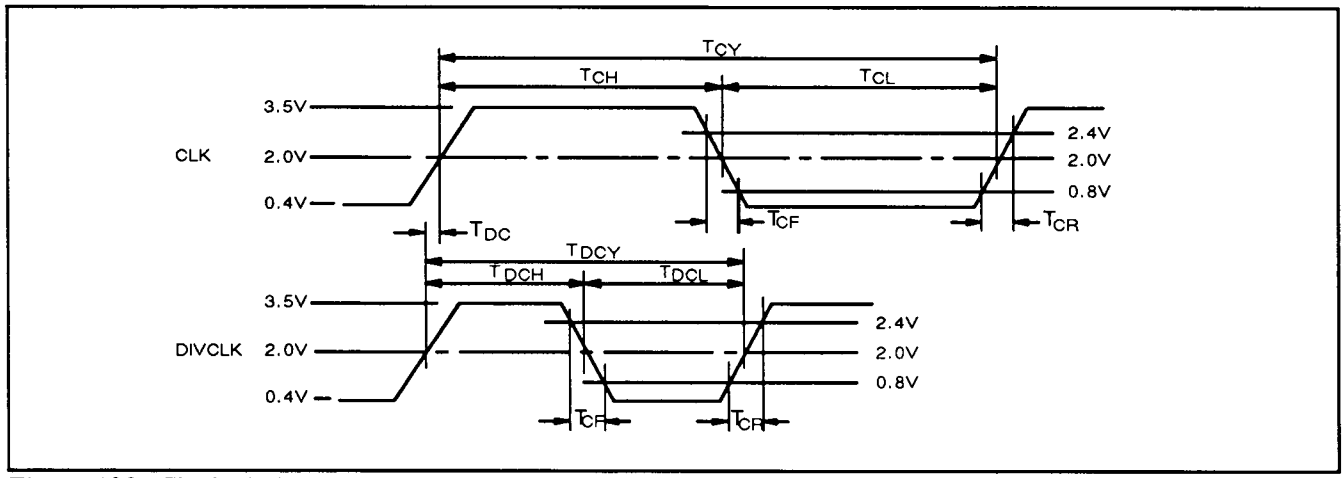


Figure 198. Clock timing

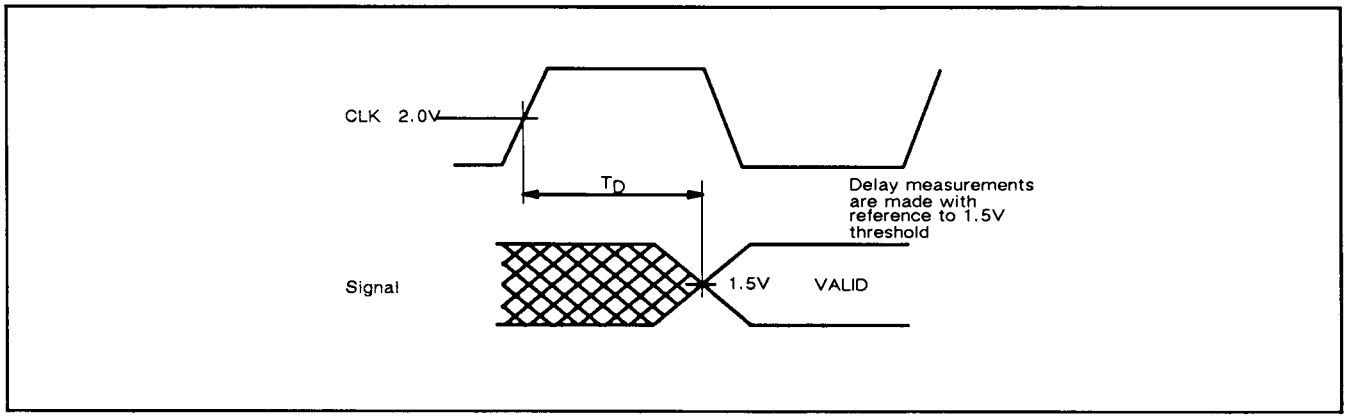


Figure 199. Reference levels in delay measurements

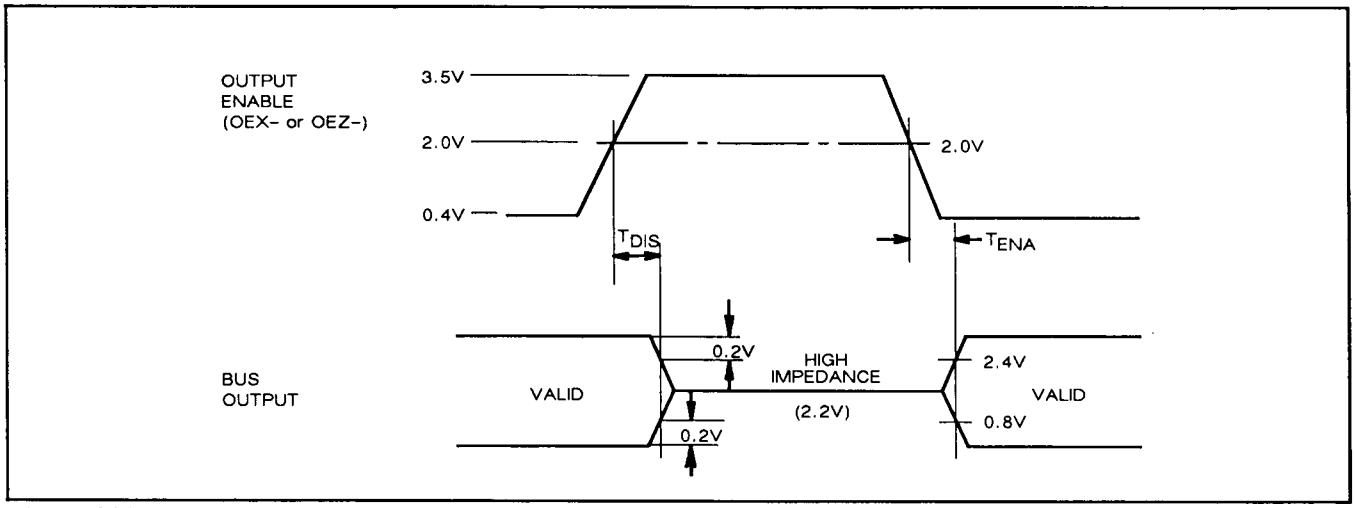


Figure 200. Tri-state timing

21.3. I/O Characteristics

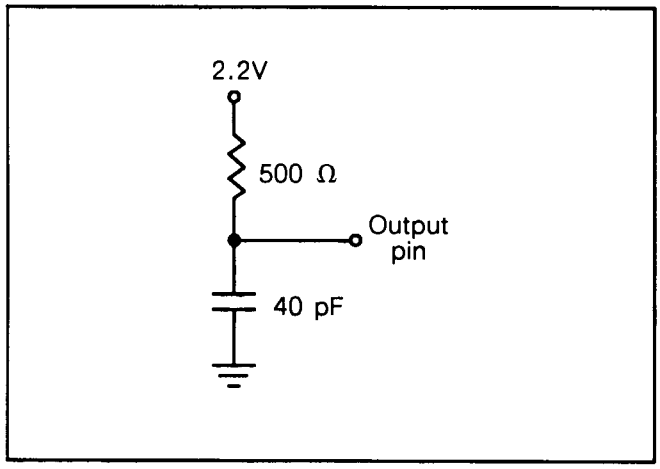


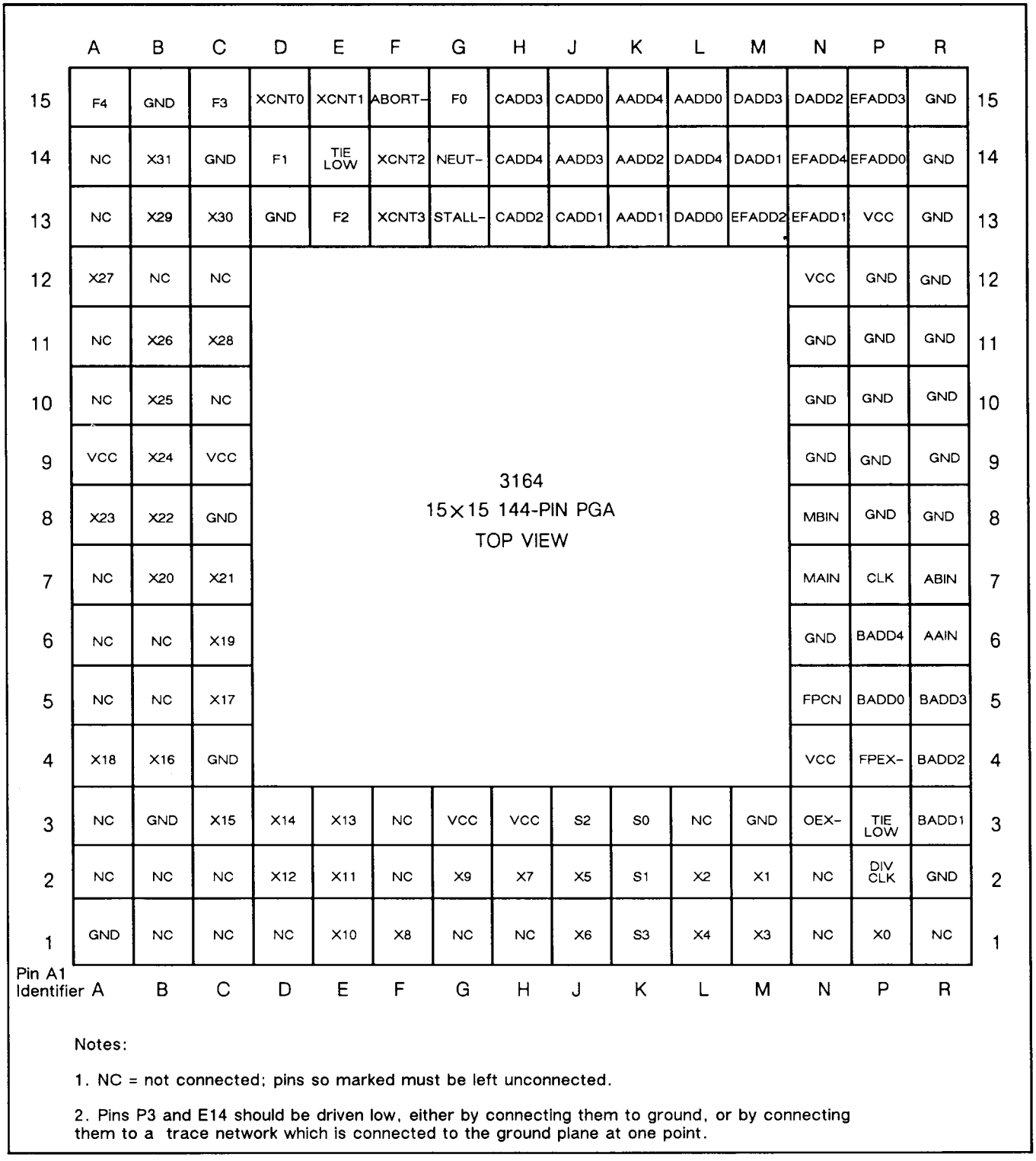
Figure 201. AC test load

21.4. The 3164-50

Double-pump delayed data store is not available in this speed grade: the timings of $T_{D4A/B}$ and T_{OZ4} are not appropriate for use at 50 ns. Double-pump data load is still available.

The status output on the S3..0 pins is not valid for concurrent operations in this speed grade. To determine which exception has occurred without using S3..0, use delayed FPEX- to signal the occurrence of an exception, then examine internal status registers (SR0..11).

21.5. Pin Configurations



	A	B	C	D	E	F	G	H	J	K	L	M	N	P	R	
15	F4	GND	F3	XCNT0	XCNT1	ABORT-	F0	CADD3	CADD0	AADD4	AADD0	DADD3	DADD2	EFADD3	GND	15
14	NC	X31	GND	F1	TIE LOW	XCNT2	NEUT-	CADD4	AADD3	AADD2	DADD4	DADD1	EFADD4	EFADD0	GND	14
13	NC	X29	X30	GND	F2	XCNT3	STALL-	CADD2	CADD1	AADD1	DADD0	EFADD2	EFADD1	VCC	GND	13
12	X27	NC	NC										VCC	GND	GND	12
11	NC	X26	X28										GND	GND	GND	11
10	NC	X25	NC										GND	GND	GND	10
9	VCC	X24	VCC										GND	GND	GND	9
8	X23	X22	GND										MBIN	GND	GND	8
7	NC	X20	X21										MAIN	CLK	ABIN	7
6	NC	NC	X19										GND	BADD4	AAIN	6
5	NC	NC	X17										FPCN	BADD0	BADD3	5
4	X18	X16	GND										VCC	FPEX-	BADD2	4
3	NC	GND	X15	X14	X13	NC	VCC	VCC	S2	S0	NC	GND	OEX-	TIE LOW	BADD1	3
2	NC	NC	NC	X12	X11	NC	X9	X7	X5	S1	X2	X1	NC	DIV CLK	GND	2
1	GND	NC	NC	NC	X10	X8	NC	NC	X6	S3	X4	X3	NC	X0	NC	1
Pin A1 Identifier	A	B	C	D	E	F	G	H	J	K	L	M	N	P	R	

Notes:

1. NC = not connected; pins so marked must be left unconnected.

2. Pins P3 and E14 should be driven low, either by connecting them to ground, or by connecting them to a trace network which is connected to the ground plane at one point.

Figure 202. 3164 pin configuration

21.5. Pin Configurations, continued

	A	B	C	D	E	F	G	H	J	K	L	M	N	P	R	T	U	
17	GND	GND	Z17	X18	Z21	X20	YCNT0	Z22	Z24	Z26	YCNT1	Z27	X26	Z29	X30	F4	GND	17
16	Z15	GND	X17	Z18	X19	X21	VCC	X22	X24	X25	VCC	X28	Z28	Z30	X31	F1	GND	16
15	X15	X14	Z16	X16	Z20	Z19	VCC	X23	Z23	Z25	VCC	X27	X29	Z31	F2	ZCNT1	F3	15
14	X13	Z13	Z14												XCNT0	XCNT3	XCNT2	14
13	Z12	Z11	X12												STALL-	XCNT1	NEUT-	13
12	X11	Z10	X10												Y0	F0	ABORT-	12
11	Z9	X8	X7												Y2	CADD4	CADD3	11
10	X9	VCC	Z8												CADD1	CADD2	Y1	10
9	Z7	VCC	Z6												CADD0	AADD3	Y3	9
8	X5	Z5	S3												AADD1	AADD2	Y4	8
7	S2	S1	X6												AADD4	AADD0	DADD4	7
6	Z4	S0	X4												DADD2	DADD3	Y5	6
5	X3	X2	Z3												Y6	DADD1	DADD0	5
4	Z2	Z1	X1												EFADD3	Y7	EFADD4	4
3	Z0	X0	GND	FPCN	BADD2	BADD4	Y28	Y30	Y26	Y25	Y27	Y20	Y17	Y12	Y9	EFADD2	EFADD1	3
2	DIV CLK	NC	GND	FPEX-	BADD3	CLK	ABIN	MAIN	Y29	Y24	Y21	Y19	Y15	Y13	Y11	VCC	EFADD0	2
1	OEX-	OEZ-	GND	BADD1	BADD0	ZCNT0	AAIN	Y31	MBIN	Y22	Y23	Y18	Y16	Y14	Y10	VCC	Y8	1
Pin A1 Identifier	A	B	C	D	E	F	G	H	J	K	L	M	N	P	R	T	U	

3364
17×17 168-PIN PGA
TOP VIEW

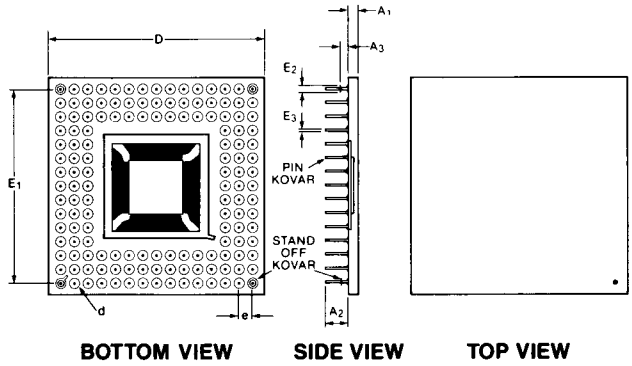
Notes:

1. NC = not connected; pins so marked must be left unconnected.

Figure 203. 3364 pin configuration

21.6. Physical Dimensions

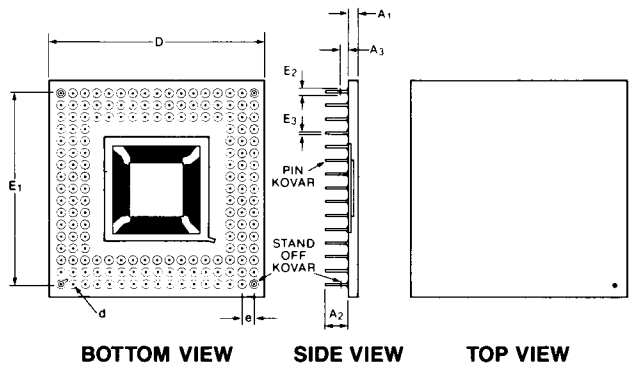
3164 144-PIN GRID ARRAY



Symbol	DIMENSIONS	
	INCHES	MM
A1	0.080 ± 0.012	2.03 ± 0.30
A2	0.180 typ.	4.57 typ.
A3	0.050	1.27
D	1.575 sq. ± 0.020	40.0 sq. ± 0.51
E1	1.400 sq. ± 0.014	35.56 sq. ± 0.36
E2	0.050 dia. typ.	1.27 dia. typ.
E3	0.018 ±0.002	.46 ± 0.05
d	0.070 dia. typ.	1.78 dia. typ.
e	0.100 typ.	2.54 typ.

Figure 204. 3164 physical dimensions

3364 168-PIN GRID ARRAY



Symbol	DIMENSIONS	
	INCHES	MM
A1	0.095 ± 0.013	2.41 ± 0.33
A2	0.180 typ.	4.57 typ.
A3	0.050 typ.	1.27 typ.
D	1.750 sq. ± 0.022	44.5 sq. ± 0.56
E1	1.600 sq. ± 0.016	40.6 sq. ± 0.41
E2	0.050 dia. typ.	1.27 dia. typ.
E3	0.018 ±0.002	.46 ± 0.05
d	0.065 dia. typ.	1.65 dia. typ.
e	0.100 typ.	2.54 typ.

Figure 205. 3364 physical dimensions

Appendix A. The 3x64 in the XL Environment

A.1 . WEITEK XL-Series

The WEITEK XL-Series is a family of three VLSI processors: the XL-8000, a high-speed 32-bit integer processor; the XL-8032, a single-precision floating-point processor, and the XL-8064, a double-precision floating-point processor.

These processors provide the performance of bit-slice components and are supported by high-level language development tools. These include optimizing C and FORTRAN compilers which produce code that is then further improved by an instruction parallelizer. The programmer remains free to create custom microcode routines for peak performance. Simulators, prototyping boards, and debugging tools are all provided.

This appendix is dedicated to the 8064 double-precision floating-point processor. Further information may be found in the *XL-Series Overview*, the *XL-Series Hardware Designer's Guide*, the *XL-Series Programmer's Reference Manual*, the *XL-8136 Data Sheet* and the *XL-8137 Data Sheet*.

The XL-Series double-precision floating-point processor is available in two versions; the XL-8164 has a 32-bit data bus and the XL-8364 has a 64-bit data bus. The XL-8164 is a simple upgrade from the single-precision XL-8032; the XL-8364 offers the enhanced performance of a full 64-bit data bus. It is possible to design a printed circuit board that will accept both the XL-8032 and XL-8164 as jumper-selectable options. Details of such a design are given in the WEITEK publication, *Supporting XL-8032/8064 Compatible Designs*.

Both the XL-8164 and XL-8364 processors consist of three interconnected VLSI components:

- XL-8136 program sequencing unit (PSU)
- XL-8137 integer processing unit (IPU)
- 3164 floating-point unit (FPU) (XL-8164 only) or 3364 floating-point unit (FPU) (XL-8364 only)

Each of these components is manufactured in high-density, low-power CMOS. The XL-8136, XL-8137 and 3164 are delivered in 144-pin grid array packages; the 3364 is delivered in a 168-pin grid array package. Unlike traditional microprocessors, the XL-8164 and XL-8364 are not constrained by the limits of circuit density or bus bandwidth imposed by a single chip in a small package. Consequently, bit-slice performance levels can be obtained both for integer and floating-point operations.

The XL-Series simplifies system design. Signals and system buses need only be connected as shown in figures 206 or 207 in order to create the XL-8164 or XL-8364, respectively.

A.1. WEITEK XL-Series, continued

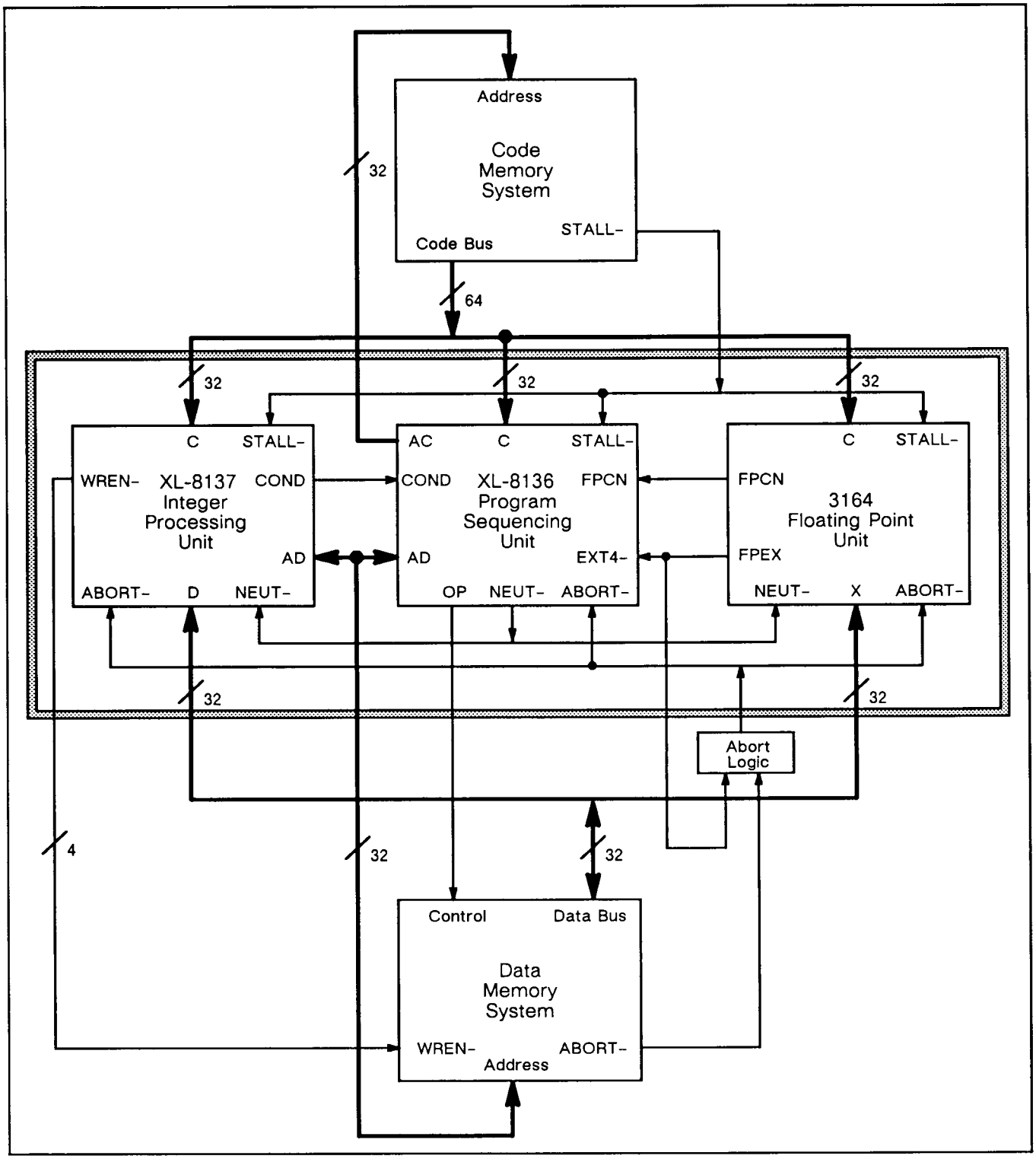


Figure 206. XL-8164 schematic

A.1. WEITEK XL-Series, continued

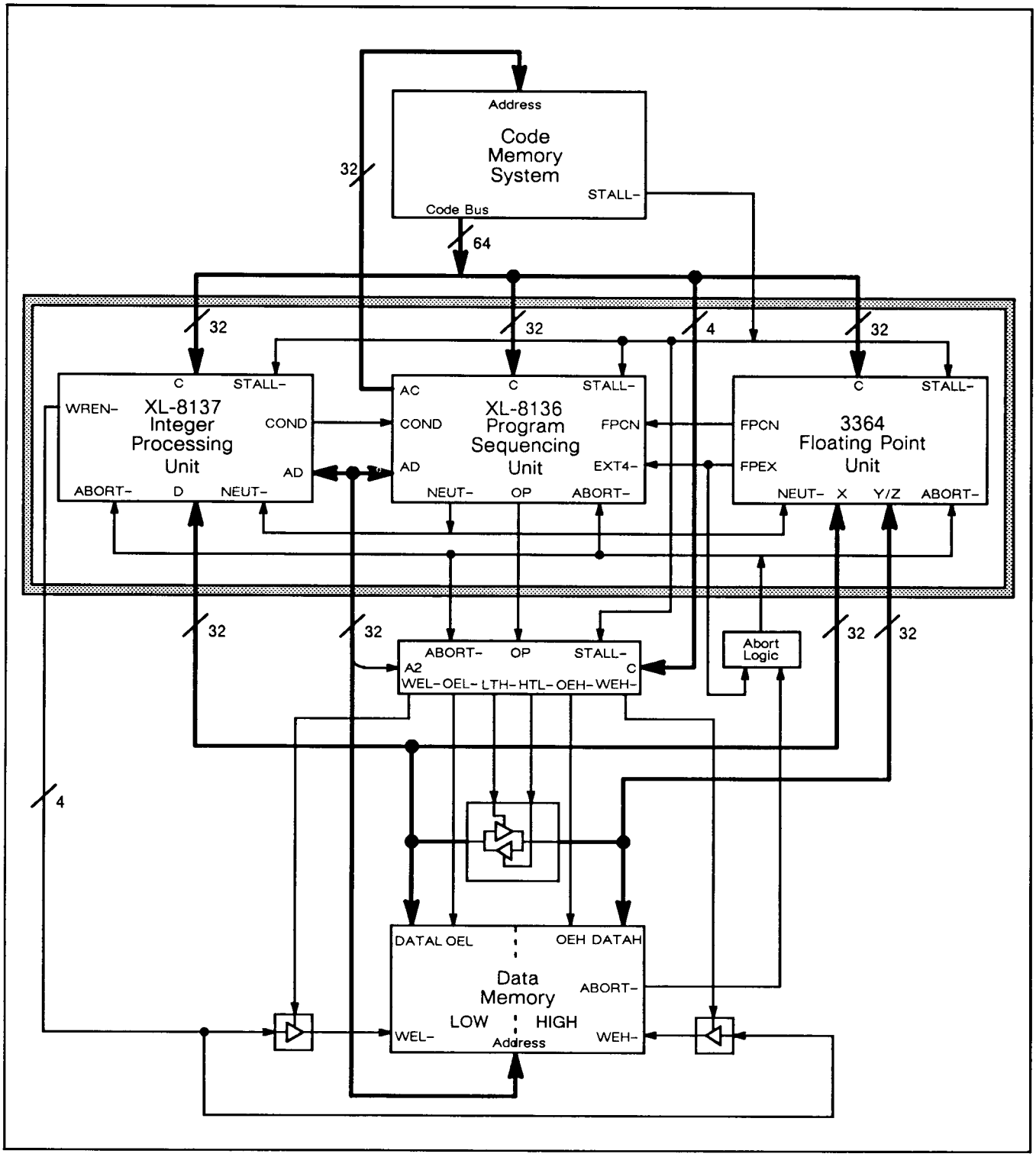


Figure 207. XL-8364 schematic

A.2. Buses

Four high-bandwidth system buses are provided by the architecture.

CODE BUS

A 64-bit code bus feeds the code input ports of the PSU, IPU and FPU with the next instruction. The PSU and IPU share 32 of the 64-bits; this half of the code word directs program control operations, address generation, loads and stores and integer arithmetic. The remainder of the code word directs floating-point operations. The designer may choose to lengthen the code word to add custom extensions to the processor architecture.

DATA BUS

If an XL-8164 design is required a 32-bit data bus is shared by the IPU and FPU. It allows bytes, 16- and 32-bit integers and 32-bit floating-point numbers to be transferred between the processing units and data memory. 64-bit floating-point numbers are handled by two successive 32-bit transfers.

Alternatively, an XL-8364 design may be employed to provide a 64-bit data bus which is shared by the IPU and FPU. It allows bytes, 16- and 32-bit integers, 32-bit floating-point numbers and 64-bit floating-point values to be transferred between the processing units and data memory. To obtain the extra performance furnished by this 64-bit bus, a few additional logic devices are required.

CODE ADDRESS BUS

A 32-bit code address bus carries the address of the next instruction from the PSU to the code memory. A word address is provided allowing up to 4 Gwords of code memory.

DATA ADDRESS BUS

A 32-bit data address bus carries the address of the next data read or write. The address is generated by the IPU and the data may be transferred to or from the IPU or FPU as required. A byte address is provided allowing up to 4 Gbytes of 32-bit-wide data memory [XL-8164] or 64-bit-wide memory [XL-8364]. Support for accessing bytes, half-words, words and double-words is provided by the IPU.

The 3x64 are designed to connect directly to the code and data buses alongside the other components of the XL-8164/XL-8364. When driven by the same system clock, the code word is sampled by all three components simultaneously and the data bus is driven or sampled at the same time in the cycle no matter which component is transferring information.

The code and data memory systems may be implemented with SRAM, static column DRAM or interleaved DRAM. Both code and data caches may be added to XL-8164/XL-8364 systems for higher performance. More details are provided in the *XL-Series Hardware Designer's Guide*.

A.3. Instruction Format

Both the XL-8164 and XL-8364 have a 64-bit microword. The bits that are directed to the 3x64 are shown in figure 208.

The lower 32-bits of the control word are shared by the IPU and the PSU. Bits 23..0 normally define the IPU operation. Bits 31..24 define the instruction flow control performed by the PSU. Five of these control bits, 28..24, are also used as the floating-point register address (EFadd) when floating-point load and store operations are performed. This saves on code bits and insures that the FPU and IPU never compete for the data bus.

The 3x64 has 42 code port inputs. When used in the XL-8164/XL-8364 configuration, the Eadd/Fadd field is tied to the appropriate bits of the PSU code input. The YCNT and ZCNT bits are held to GND as only the XCNT field controls data transfers. In order to further reduce the code word size to the 37 bits required by the XL-8164/XL-8364 format, the most-significant bit of address fields Cadd and Dadd is shared. This only has an impact when multiply and ALU functions are executed in parallel (chained instructions and mul/add) in which case the assembly language programmer has to insure that the two destination registers lie in the same half of the register file.

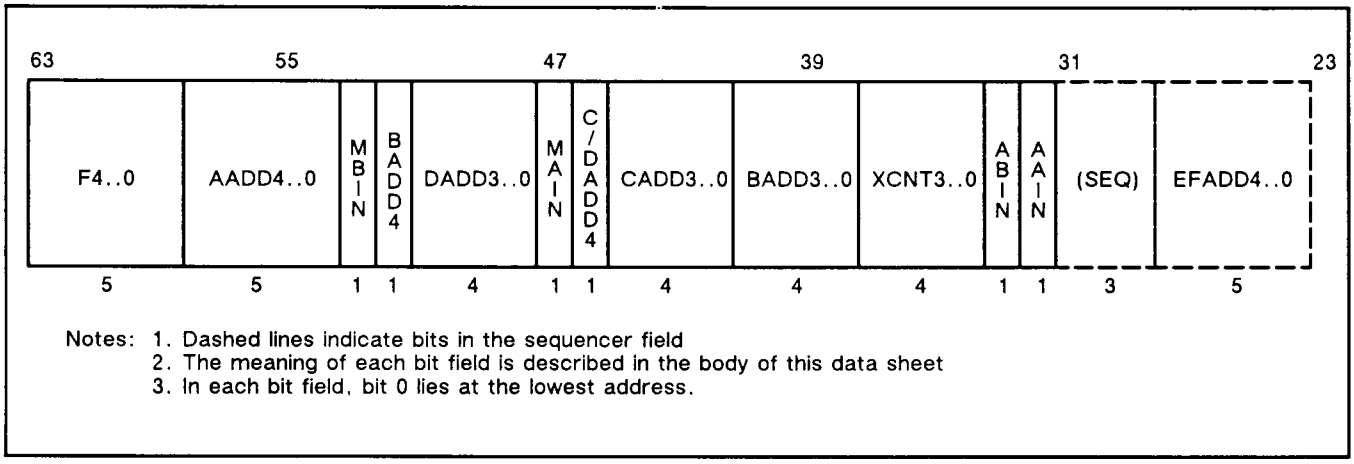


Figure 208. XL-8064 code word

A.4 . Load/Store Model

The XL-Series has a consistent load/store model regardless of processor configuration. Each processing unit has its own register file: register moves between the IPU and FPU must be made through the data memory. Each of these register files is multi-ported and each register may be the operand source or result destination of any instruction implemented by the unit. When an instruction takes more than one cycle to execute, the registers that supply its operands and receive its result cannot be modified until it has been completed. This allows any instruction to be resubmitted for execution with its original state after an interrupt.

Transactions between the register files and data memory are performed with dedicated load/store operations. The only restriction on loads and stores is that the operands of an operation must be loaded before it is executed and that it shall have completed before its result is stored back to memory. This allows the parallelizer considerable freedom to optimize register usage and I/O transactions.

An example of the normal sequence of operation is given below. This example leaves several free cycles in which other loads, stores and calculations could be performed in parallel. In this particular case, single-length floating-point values are being transferred on the 32-bit-wide XL-8164 data bus. Double-length floating-point values have the same timing on the XL-8364's 64-bit data bus and the case of such doubles on the XL-8164 is shown in the next section.

```
.....
addr .ra
fload .fx
fabs .fx, .fy
addr .rb
fstore .fy
....
```

The load and store modes of the 3x64 are selected to produce the same timing in the FPU as in the IPU. For loads, the address is presented on the AD bus at the beginning of a cycle and the data is expected to be available on the D bus at the end of that cycle.

A.4. Load/Store Model, continued

For stores, the address is presented on the AD bus at the beginning of a cycle and the data is driven onto the D bus during the next cycle.

Because the addr and fload instructions can be executed in parallel, they may be pipelined to support contiguous fload operations (one per cycle). The fstore operation, on the other hand, requires two cycles: the second cycle is filled with an I/O nop. This automatically prevents bus turnaround conflicts when floads follow immediately after fstores. It has minimal impact on overall performance because loads usually outnumber stores and because the parallelizer can organize I/O transfers efficiently. If WEITEK software tools are used, then this load/store model will be followed.

A.5 . Data Bus Implementation

XL-8164

The XL-8164 has a 32-bit data bus. It must be connected to the D port of the XL-8137 IPU, the X port of the 3164, and the data memory system bus. The 3164 is placed in configuration C (see the main body of this data sheet) by setting its status registers as shown below. The XL-8164 bus is identical in operation to the XL-8032 bus. Double-precision variables are loaded into the X port of the 3164 by two successive 32-bit floads; fstores proceed in a similar manner. Double-precision values are stored on even boundaries, with the least-significant half occupying the lowest address (A2=0).

The following code sequence will load a double-precision floating-point word to the 3164's Y register:

```
addr+ .ra, 1, .word, .ra
addr .ra, 1, .word; dloadl .y
dloadm .y
```

More details on the assembler syntax for double-precision loads and stores may be found in the *Programmer's Reference Manual.*

XL-8364

The XL-8364 has a 64-bit data bus. This bus connects the 32-bit D port of the XL-8137 integer processing unit, the 64-bit X, Y/Z port of the 3364 and the 64-bit-wide system memory. This may be seen in figure 207.

The 3364 is placed in configuration B (see the main body of this data sheet) by setting its status registers as shown below. In this mode, the X port transfers the least-significant half of double-precision floating-point values and any 32-bit integer values. The Y and Z ports are tied together to produce another 32-bit I/O port which transfers the most-significant half of double-precision floating-point values and any single-precision floating-point values.

In memory, double-precision floating-point values are aligned on 64-bit boundaries, whereas single-precision floating-point values and integers are aligned on 32-bit boundaries. The least-significant half of the double is stored at the lowest address (A2=0). The X port should, therefore, be connected to the even addresses and the Y/Z port to the odd ones. Double-precision dloads and dstores are then straightforward; the address is output and, on the next cycle, the data is transferred least- and most-significant halves together.

It is often necessary to load a single located at an even address, or an integer at an odd address, to the FPU. In these cases, transceivers that allow the data to cross over from one half of the 64-bit word to the other must be enabled. These cases may be detected by decoding the XCNT field in the instruction word with the A2 address bit. A pipeline register for the XCNT field should be provided to synchronize it to the address output. The resulting signals can then be used to control the transceivers,gate the WREN- lines and activate the OE-'s of the data memory system before the data transfer occurs.

The crossover logic is also needed to allow integers to be transferred between both even and odd addresses and the integer processing unit. The decode logic must, therefore, also sense integer loads and stores from the OP bus of the IPU. Care must betaken to insure that all loads and stores are properly qualified by the NEUT-, STALL- and ABORT- signals discussed below.

Refer to the XL-8364 section of the *Hardware Designer's Guide* for an example implementation.

A.6. Modes

The 3x64 mode register must be initialized to the values given in figure 209 when used with the XL-8164/XL-8364 processor.

The remainder of the status register bits change according to the state of the FPU and are used by the trap handler to recover from exceptions. All of the status registers are saved along with the general purpose, x,y and t registers during a context switch. Each selection is explained here:

1. Internal NEUT- cancels register writes when an exception is detected. This allows the system to recover from multiple exceptions and allows the XL-8164/XL-8364 to provide full support for the IEEE standard even for pipelined operations.
2. Two-cycle multiplier latency is selected.
3. The load/store modes are set to match the XL-Series load/store model.
4. FPEX is delayed to the beginning of the cycle after and exception occurs. Because the XL system may "back-up" a cycle, this does not impact performance and eases design constraints.
5. Register file bypassing is enabled to minimize the register-to-register latency.

Mode Bit	Logic Value	Description
$SR0_0$	0	IEEE mode selected
$SR0_1$	0	Round to nearest mode selected
$SR0_2$	0	
$SR0_3$	1	Internal NEUT- on
$SR0_6$	1	FPEX active low and sticky
$SR0_7$	0	Two-cycle multiplier latency
$SR1_0$	0	XL-8164 load/store modes Load mode = SP-D Store mode = SP-DD
$SR1_1$	0	
$SR1_2$	1	
$SR1_3$	1	
$SR1_4$	0	
OR	OR	OR
$SR1_0$	0	XL-8364 load/store modes Load mode = SP-D/CB Store mode = SP-D/CB
$SR1_1$	1	
$SR1_2$	1	
$SR1_3$	1	
$SR1_4$	0	
$SR1_5$	1	FPEX delayed
$SR1_6$	1	Register file bypass enabled

Figure 209. XL-8064 mode register

A.7. Conditions and Exceptions

The XL-8164/XL-8364 provide several signals which transfer state information from the processing units (IPU, FPU) to the sequencer (PSU). These are either conditions, upon which the PSU may decide to branch; or exceptions, which require software intervention to recover gracefully.

The FPCN output on the 3x64 should be connected to the FPCN input on the XL-8136. The FPCN signal may be asserted according to the outcome of a compare instruction. The PSU may then execute a "branch on condition" instruction to selectively transfer program control according to the outcome of the comparison. See the body of this data sheet for details of the compare instructions.

The FPEX output on the 3x64 should be connected the EXT4- interrupt input on the XL-8136. When an enabled floating-point exception is detected, the FPU activates the internal NEUT- signal and sets the FPEX taken bit in the status register. The FPEX output is activated at the beginning of the following cycle, and is detected as an interrupt on EXT4- by the PSU at the end of this cycle. In addition, the FPEX pin on the FPU should be OR'ed into the system ABORT- line for one cycle. The internal NEUT- causes the current instruction to be held in the code register; the trap handler can then read, modify, and re-execute it before returning from the interrupt. The ABORT- insures that the system begins re-execution of the next sequential instruction after the return. The source code of an example IEEE trap handler is available from WEITEK.

A.8 . NEUT-, STALL- and ABORT-

The XL-8164/XL-8364 components all use the NEUT-, STALL- and ABORT- signals. These pins should be connected directly between the three chips in the XL-8164/XL-8364 processor (see figures 206 and 207).

NEUT- cancels the effect of the current instruction. The signal is generated by the PSU. It is normally used in the shadow of a delayed branch to prevent the instruction in the pipeline from having any effect on the state of the IPU and FPU.

STALL- cancels the effect of the next instruction. It should be generated by the code memory subsystem to indicate the delay or absence of the correct code word. This prevents any invalid operation that may be present on the code input at this time from affecting the state of the processor. It allows wait states to be inserted in code fetches, perhaps to allow for DRAM refresh or a code cache miss.

ABORT- cancels the effect of both the current and the next instructions. It should be generated by the data memory subsystem to indicate the inability of the system to instantly access the required data word. This allows the cancelled instructions to be repeated when the address becomes valid and for this retry to have the correct effect. It allows the data memory to be "not ready" if, for example, a page fault occurs.

Appendix B. 3x64 Programming Examples

B.1 . 3x64 Initialization

This example initializes a 3x64 to operate within an XL-8164 system. In this configuration, the 3x64 is connected to a single 32-bit data bus.

The first task during initialization is to set the I/O mode. In the example that follows, lines 25–38 do this. Lines 25–26 establish the desired setting of status register .sr1 in bits 24–31 of register .r0. Lines 27–28 move this constant to memory in anticipation of loading it into the 3x64.

The actual movement of the constant to .sr1 is done by the loop on lines 31–38. This loop is executed three times. It is thought that the first trip through the loop may fail to accomplish the initialization due to code cache misses, and the second trip may fail due to data refresh. The third trip insures that the I/O mode is properly set.

Once the I/O mode is properly established, lines 44–49 completes the initialization of status register .sr0. Following this, lines 54–66 clear the remaining status registers, .sr2–.sr11.

If the instances of constant 0x6c000000 appearing on lines 25 and 26 were to be replaced with 0x6e000000, then the proper initialization for an XL-8364 system would be performed. This is the configuration with a 64-bit data bus.

```
 1 /*****************************************************************************
 2  * reset - initializes WTL3164 in an XL-8164 system to operate in standard  *
 3  *   XL-series mode.                                                         *
 4  *                                                                           *
 5  * input:                                                                    *
 6  *   .r31 = .sp = stack pointer                                              *
 7  *                                                                           *
 8  * output:                                                                   *
 9  *   .sr0-.sr11 properly initialized                                         *
10  *****************************************************************************/
11
12         .text
13         .globl reset
14
15 reset:
16
17 /*****************************************************************************
18  * First set the load and store modes.  This is done in a loop 3 times since  *
19  * the first trip through the loop may be aborted due to code cache misses,   *
20  * the second may be aborted due to memory refresh, and the third trip may    *
21  * therefore be required.  I/O mode is single pump, 32-bit bus, delayed load, *
22  * and delayed data store.  Also, FPEX is delayed and register file bypassing *
23  * is enabled.                                                               *
24  *****************************************************************************/
25   movi 0x6c000000,.r0
26   movih 0x6c000000>>16,.r0
27   +addr .sp,-1,.word
28   store .r0
29   movi 3,.r1
30
31   addrd .sp,0;                    loop
32   addrd .sp,0;          fldsr .sr1
33   addrd .sp,0;          fldsr .sr1
34   addrd .sp,0;          fldsr .sr1
35   addrd .sp,0;          fldsr .sr1
36                         fldsr .sr1
37   subi .r1,1,.r1;                 endloop .gtz
38   addrd .sp,0;                    revneut
```

B.1. 3x64 Initialization, continued

```
39
40 /*****************************************************************************
41  * Next set .sr0  -  2-cycle mode, FPEX sticky, internal neut on, round to     *
42  * nearest, and FAST modes.                                                    *
43  *****************************************************************************/
44   movi 0x49000000,.r0
45   movih 0x49000000>>16,.r0
46   addrd .sp,0
47   store .r0
48   addrd .sp,0
49                            fldsr .sr0
50
51 /*****************************************************************************
52  * Clear the remaining status registers                                        *
53  *****************************************************************************/
54   addrd .sp,0
55   clr .r0;  store
56   addrd .sp,0
57   addrd .sp,0;          fldsr .sr2
58   addrd .sp,0;          fldsr .sr3
59   addrd .sp,0;          fldsr .sr4
60   addrd .sp,0;          fldsr .sr5
61   addrd .sp,0;          fldsr .sr6
62   addrd .sp,0;          fldsr .sr7
63   addrd .sp,0;          fldsr .sr8
64   addrd .sp,0;          fldsr .sr9
65   addrd .sp,0;          fldsr .sr10
66                         fldsr .sr11
67
68   addi 4,.sp,.sp;                    rts
```

B.2 . Saving the State of the 3x64

This second example shows how the internal state of the 3x64 can be saved when context must be switched. This routine is written for an XL-8164 system, in which the 3x64 is connected to a single 32-bit data bus. As written, this code depends on register file bypassing being enabled. To execute with bypassing disabled would require adjusting the program to account for an additional cycle of latency for certain instructions.

Referring to the code listing that follows, the first section of the routine, lines 16–109, defines the layout of the 90-word save area that stores the context information. The layout is defined by means of .set assembly directives. These directives allow a symbol (that immediately follows the directive name) to be assigned the value of the expression that follows. Using these directives, the layout is specified in terms of symbolic names given to relative offsets for each element of the context. These symbolic offsets are then used in the routine to address the save area relative to the pointer passed to the routine in register .r0. (Register .r0 is referred to by the symbolic name sap following the definition on line 114.)

The first six words of the save area receive the code register, as shown by lines 16–21. Each byte of the code register is stored in bits 24–31 of a memory word, for this is the format in which the register in transferred to and from the 3x64. Similarly, the next twelve words receive the twelve bytes of the status register (lines 23–34). The contents of the 3x64's register file are stored in the next 64 words (lines 36–99), which are followed by eight words to save the .x, .y, and .t registers (lines 101–109).

The instruction sequence required to save the code register appears on lines 125–136. The XL-Series memory model allows a store operation to occur every other cycle; hence, the sequence shown alternates address generation instructions with code register stores. This pattern is also continued through lines 139–165 which store the status register in a similar manner.

Lines 167–297 save the 3x64's register file. Due to the 32-bit data bus, the registers must be stored by 32-bit halves. The code sequence shows each register being saved as least-significant half (dstorel) followed by most-significant half (dstorem). The same is true for saving the .x and .y registers (lines 299–310) following a move of their contents to the register file. In an XL-8364 system with a 64-bit data bus, a single address generation and double-precision store (dstore) would be used for each register.

B.2. Saving the State of the 3x64, continued

Saving the .t registers completes the context save function. This is the only part of the function that is not entirely straightforward. Within the routine, lines 319–325 first store a double-precision floating-point value of zero on the stack and then load this into the .x register on the 3x64. At the same time, lines 319 and 321 create a value of –0 (negative zero) in register .f2. Line 326 then adds the –0 to the contents of register .t0 and puts the result in register .f1. In most circumstances, this add operation transfers the contents of .t0 to the register file. However, if round to negative infinity mode is set and .t0 contains a value of zero, a value of –0 is stored in the register file. Also, NaNs and denorms will be affected. (Users concerned with full IEEE conformance should consult their design consultants for information on how this can be achieved.) Once the value in the .t0 register has been moved to the register file, it is saved in the normal manner (lines 327–330). This process is also repeated for the .t1 register (lines 330–334).

```
 1 /*****************************************************************************
 2  * save_3164  -  saves the state of the 3164.
 3  *                                                                           *
 4  * input:                                                                    *
 5  *    .r0 = address of save area                                             *
 6  *    .sp = stack pointer (2 words used)                                     *
 7  *                                                                           *
 8  * output:                                                                   *
 9  *    A copy of all registers is placed in the save area.                    *
10  *****************************************************************************/
11
12
13 /*****************************************************************************
14  * layout of the save area (90 words)                                        *
15  *****************************************************************************/
16   .set cr0,0            /* bytes of code word stored in bits 24-31 of word */
17   .set cr1,cr0+4
18   .set cr2,cr1+4
19   .set cr3,cr2+4
20   .set cr4,cr3+4
21   .set cr5,cr4+4
22
23   .set sr0,cr5+4        /* bytes of status register stored in bits 24-31 */
24   .set sr1,sr0+4
25   .set sr2,sr1+4
...
34   .set sr11,sr10+4
35
36   .set f0_l,sr11+4      /* register file as least and most significant words */
37   .set f0_m,f0_l+4
38   .set f1_l,f0_m+4
39   .set f1_m,f1_l+4
...
98   .set f31_l,f30_m+4
99   .set f31_m,f31_l+4
100
101  .set fx_l,f31_m+4     /* x register as least and most significant words */
102  .set fx_m,fx_l+4
103  .set fy_l,fx_m+4      /* y register as least and most significant words */
104  .set fy_m,fy_l+4
105
106  .set ft0_l,fy_m+4     /* t registers as least and most significant words */
107  .set ft0_m,ft0_l+4
108  .set ft1_l,ft0_m+4
109  .set ft1_m,ft1_l+4
```

B.2. Saving the State of the 3x64, continued

```
110
111 /*****************************************************************************
112  * 8137 register map                                                         *
113  *****************************************************************************/
114   .reg .r0 sap          /* save area pointer */
115   .reg .r1 rx           /* scratch */
116
117
118
119   .text
120   .globl save_3164
121 /*****************************************************************************
122  * Save the code register first                                              *
123  *****************************************************************************/
124 save_3164:
125   addrd sap,cr0
126                   fstcr .cr0
127   addrd sap,cr1
128                   fstcr .cr1
129   addrd sap,cr2
130                   fstcr .cr2
131   addrd sap,cr3
132                   fstcr .cr3
133   addrd sap,cr4
134                   fstcr .cr4
135   addrd sap,cr5
136                   fstcr .cr5
137
138
139 /*****************************************************************************
140  * Save the status register                                                  *
141  *****************************************************************************/
142   addrd sap,sr0
143                   fstsr .sr0
144   addrd sap,sr1
145                   fstsr .sr1
...
164   addrd sap,sr11
165                   fstsr .sr11
166
167 /*****************************************************************************
168  * Save the register file                                                    *
169  *****************************************************************************/
170   addrd sap,f0_l
171                   dstorel .f0
172   addrd sap,f0_m
173                   dstorem .f0
174   addrd sap,f1_l
175                   dstorel .f1
176   addrd sap,f1_m
177                   dstorem .f1
...
294   addrd sap,f31_l
295                   dstorel .f31
296   addrd sap,f31_m
297                   dstorem .f31
```

B.2. Saving the State of the 3x64, continued

```
298
299 /*************************************************************************
300  * Save the x and y registers                                            *
301  *************************************************************************/
302                                  mov .x,.f0
303   addrd sap,fx_l;                mov .y,.f1
304                    dstorel .f0
305   addrd sap,fx_m
306                    dstorem .f0
307   addrd sap,fy_l
308                    dstorel .f1
309   addrd sap,fy_m
310                    dstorem .f1
311
312 /*************************************************************************
313  * Save the .t0 and .t1 registers.  The t registers are unloaded by adding   *
314  * -0 to their contents.  In most cases the t register + (-0) yields the     *
315  * register contents.  However, if round to negative infinity mode is set    *
316  * and the t register contains 0, a -0 results.  Denorms and Nans are also   *
317  * affected.                                                                 *
318  *************************************************************************/
319   +addr .sp,-1,.word;            fclr .f2
320   clr rx;  store
321   +addr .sp,-1,.word;            dfneg .f2,.f2
322   store rx
323   addr .sp,0,.word
324   addr .sp,1,.word;  dloadl .x
325   addai 8,.sp,.sp;   dloadm .x
326                                  dfpass .x,.f1,.t0;  dfadd .f2,.t0,.f1
327   addrd sap,ft0_l
328                    dstorel .f1
329   addrd sap,ft0_m
330                    dstorem .f1;  dfpass .x,.f1,.t1;  dfadd .f2,.t1,.f3
331   addrd sap,ft1_l
332                    dstorel .f3
333   addrd sap,ft1_m
334                    dstorem .f3
335
336   rts
337
```

B.3. 3-D Graphics Example

This example illustrates how the 3364 embedded in a XL-8364 system (64-bit bus) can be used as the transformation processing element in a 3-D graphics system. As shown, the routine maps a 3-D polyline with coordinates in double-precision floating-point representation to screen-space coordinates in 32-bit integer format. The routine does a full 4×4 matrix transformation of local coordinates to clip-space, performs homogeneous clipping, does a perspective division, and finally completes the 2×2 transformation to screen-space.

This routine is written to be executed in 2-cycle multiply latency mode. It is IEEE-interruptible.

PERFORMANCE

The performance of the routine in various cases is presented in figure 210. As shown in this figure, the trivial acceptance loop requires 31 cycles to process a 3-D point. *Trivial acceptance* is said to occur when all points in the polyline are contained within the viewbox.

Within the loop, the first 15 cycles are required to perform the 4×4 transformation to clip-space and to begin the computation of $1/w_c$. The next six cycles are required to test if the transformed point lies in the viewbox, and the final ten cycles are required to perform the perspective division and the 2×2 transformation to screen-space. Additional details of these operations are presented in the next section.

Trivial rejection of a polyline occurs when each of its line segments lies on the outside of at least one plane bounding the viewbox. As shown in figure , the trivial rejection loop requires 29, 32, or 35 cycles. The first 16 cycles of this loop are required to access the point and transform it to clip-space. This is followed by six cycles of testing the point against the viewbox. For rejection to occur, 1–3 of the tests will indicate that the point lies outside a clipping plane. This event requires control to pass out of the loop so a flag can be set before testing is resumed — an action that adds three cycles to the execution path. Finally, two cycles of overhead are required to complete the loop. Hence, 29, 32, or 35 cycles are required depending on the number of clipping planes that separate a point from the viewbox interior.

The *single clip* operation listed in figure 210 refers to processing a two-point polyline whose first point lies within the viewbox and whose second point lies outside a single clipping plane. As shown in the figure, this processing requires 106–108 cycles to complete.

The example is a partial implementation of a polyline transformation routine. It does not include the code to process the other two cases requiring clipping: the case where a segment's fist endpoint is outside and its second point is inside the viewbox, and the case where both endpoints are outside the viewbox. These cases can be handled by a generalization of the method shown in the example.

THE EXAMPLE — PRELIMINARY MATERIAL

Referring to the code listing of the example, lines 1–33 contain some introductory comments. Following this on lines 34–71, a set of assembler .reg directives assign symbolic names to the 32 registers of the 3364. A similar map for the registers used in the integer processing unit appears on lines 73–85.

Operation	Cycles Required
trivial acceptance	
(2 points)	71
(*n* points)	9 + 31*n*
trivial rejection	
(2 points)	72
(*n* points)	9 + (29 or 32 or 35)*n*
single clip	
(2 points, against single plane)	106–113
additional clip	
(2 points, each additional plane)	14–15

Figure 210. Performance of the graphics routines

B.3. 3-D Graphics Example, continued

The entry point for the routine, polyline, appears on line 93. This labels a block of three instructions that perform initialization. Line 94 transfers the count of points in the polygon from a register in the integer processor to the top of stack in the sequencer. Line 96 generates an address reference to the x-coordinate of the first point of the polyline.

THE TRIVIAL ACCEPTANCE LOOP

The trivial acceptance loop follows on lines 98–155. For the first point of a polyline, execution of this loop begins on line 111. Reading from the left, this line shows first the addr instruction that generates an address one double word (.quad) beyond the location referenced by the current input pointer, *ip*. The address generated is that of the first point's y-coordinate. Continuing with line 111, the "dload .x" causes the .x register in the 3364 to be loaded with the double word referenced by the preceding address instruction — in this case the x-coordinate of the first point. The final instruction on line 111 shows a "dfixr t,ys," which for each execution of the loop beyond the first converts the previous point's screen-space coordinate from double-precision floating-point to integer representation. Thus it can be seen that the loop has been "folded" so that each trip through it performs initial calculations for a point and final calculations and storing of the previous point.

Line 112 begins with another addr instruction, this time to generate the address to store the screen-space x-coordinate for the previous point (*op* is the output pointer). On this same line, the double-precision y-coordinate for the current point is loaded into the .y register.

Line 112 also begins the dot product computation for the w-coordinate of the current point's image in clip space by multiplying $x \times a_{1,4}$. This is, of course, an example of computing the sum-of-products. The multiply instruction of line 112, "dfmul .x,a14,s,.t0," sends its result to both the register named t and the temporary register .t0. If you examine the add instruction (line 114) that will be executed two cycles later (dfadd a44,.t0,xc), you will see that the product of $x \times a_{1,4}$ is being added to $a_{4,4}$ (or $w \times a_{4,4}$ since all w's are equal to 1) — the previous product is added via the .t0 register. Line 116 computes the final product and lines 116 and 118 add two more products to complete the determination of w_c. In the same manner, lines 113, 115, 117, and 119 simultaneously compute the x-coordinate of the point. Line 120 can then begin the computation of $1.0/w_c$. Lines 119, 121, 123, and 125 compute the dot product for the y-coordinate, and lines 118, 122, 124, and 126 do it for the z-coordinate.

While the transformation of the point is being accomplished, other operations are also happening in parallel. Lines 112 and 113 perform the address generation and storing of the x-value generated for the previous point. Lines 116 and 117 do this for the previous point's y-value. The integer processor instructions on lines 113 and 117 establish a pointer in register .dp to the data area beginning at symbolic label const. The multiply instruction on line 125 begins the transformation of the clip-space x-value to screen-space by multiplying it by the x scale factor. The multiply on line 126 computes $-1 \times w_c$, so that $-w_c$ is available for the viewbox testing that follows. Additional loading operations also take place.

Testing to determine if the transformed point lies within the viewbox is done on lines 133–138 (and 145). Each compare instruction, such as the "dfcmp xc,wc,.gtz" on line 133, is followed on the next line (cycle) with a floating-point conditional branch instruction that acts on the result of the comparison. If a point fails one of these tests against the six planes of the viewbox, then control exits from the loop and enters supplementary code to clip or reject the current polyline segment.

The final portion of the loop, lines 145–152 along with lines 109–111, completes the mapping of the point to screen-space and the conversion to integer output format. Line 145 applies a scale factor to the y-value. The multiplies on lines 149 and 150 accomplish the perspective division, and the adds on lines 151 and 152 translate the clip-space origin to the screen-space origin. As previously mentioned, the fix instructions on lines 109 and 111 do the final conversion to integer format.

Because the IEEE divide begun on line 120 requires 17 cycles to complete, three floating-point operations remain available before the result of the division can be used on line 149. These three cycles are consumed by the adds shown on lines 146–148. The effect of these operations is to copy the w–, x–, and y–coordinates of the current point to a second set of registers. The z-coordinate is transferred through memory during other spare operations. If it is discovered on the next trip through the loop that clipping is required, this second set of registers contains the previous point.

B.3. 3-D Graphics Example, continued

The output pointer is incremented by the output pointer increment, the value in the opi register, on line 151. On the first trip through the loop this increment will be zero. This has the effect of discarding the garbage output that was stored earlier in the loop. Line 152 initializes this increment to eight for subsequent trips through the loop, representing the eight bytes output for each point.

CLIPPING

As explained in the comments on lines 187–203 in the example, clipping is done by evaluating a set of 2×2 determinants. When a coordinate is found to be outside a clipping plane in the accept loop, a (short) branch is made to one of the six two-line program segments on lines 167–184. Each of these segments begins the computation of the determinants and branches to a segment of code that completes these computations. Each of the two-line program segments also sets a flag in the *codes-for-point-2* register, cp2. This set of flags can be interpreted as the signs of the results of each of the clipping plane tests.

Lines 204–218 represent the segment of code that completes clipping of a line at the $xc = w_c$ plane. This segment first tests the value of the output pointer increment, *opi*. If this value is zero, then the it is the first point of a polyline segment that has been discovered to be outside the viewbox. In this case, control is transferred to the trivial rejection loop to complete testing of the point and to begin the rejection process.

The remainder of this segment of the clipping code, lines 206–215, does a straightforward evaluation of the determinants. Additionally, $-w_c$ is computed to allow further viewbox testing of the clipped point. The lines 220–317 contain five additional code segments to clip line segments at the other five bounding planes.

Each of the segments that perform clipping exit to another segment that continues the viewbox testing of the new point (lines 324–329). Testing continues and any further clipping that is required is performed. When the second endpoint is finally determined, the division $(1/w_c)$ is initiated (line 329).

Control then passes to the code segment on lines 359–380. This segment completes the mapping of the second point of the clipped segment to screen-space and outputs the results. Finally, control passes to the trivial rejection loop to begin the rejection process.

TRIVIAL REJECTION

The first portion of the trivial rejection loop, lines 393–408, is responsible for transforming input points to clip-space. It differs is only minor aspects from that described for the trivial acceptance loop.

As expected, the next portion of the loop performs the viewbox testing (lines 415–421). If a test against a plane of the viewbox fails, then control transfers to one of the six two-line instruction segments on lines 435–446. This two-line instruction segment merely sets the appropriate flag in the codes-for-point-2 register, cp2, and returns to the viewbox testing sequence.

OTHER CONSIDERATIONS

As written the polyline routine processes points whose coordinates are represented as double-precision floating-point values. What about handling single-precision coordinate lists? By substituting fdmul for dfmul in each of the *chained instructions* in the dot product calculations (lines 112–126 and 394–408) the routine will handle single-precision input. The fdmul instructions will multiply the single-precision coordinates by a double-precision matrix element and yield double-precision results. The remainder of the calculations can be completed in double-precision. Of course, the dload's used to load the coordinate values must also be changed to fload's. Doing the transforms with a double-precision matrix can facilitate the retention of precision when using concatenated transformations.

As stated earlier, doing all calculations in full double-precision with double-precision input requires that the 3364 operate in two-cycle multiply latency mode. Operating on single-precision input data with fdmul instructions, however, would allow the routine to use a 3x64 operating in either two-cycle or three-cycle latency mode. This is because it is only the full double-precision multiply that requires three cycles in three-cycle mode — the mixed mode multiplies require only two cycles. The next example shows how slightly higher performance can be realized when doing strictly single-precision calculations.

B.3. 3-D Graphics Example, continued

```
 1 /*************************************************************************
 2  * polyline -- generates a sequence of screen-space coordinates, (x,y),      *
 3  *   given a list of local coordinates.  Coordinates are 64-bit floating     *
 4  *   point and all arithmetic is done in 64-bit floating point format.       *
 5  *                                                                           *
 6  * input:                                                                    *
 7  *   .r0 = count of points in polyline                                       *
 8  *   .r1 = address of list of (x,y,z)'s (all w's assumed = 1)                *
 9  *   .r2 = address of area to receive list of screen-space (x,y)'s           *
10  *   .r3 = address of screen-space transformation                            *
11  *   .f16-f31 = 4x4 transformation from local coordinates to clip-space      *
12  *                                                                           *
13  * output:                                                                   *
14  *   .r2 = address+2 of last point in list of (x,y)'s                        *
15  *   .f16-.f31 left undisturbed                                              *
16  *                                                                           *
17  * subroutine calls:                                                         *
18  *   outpl -- to output the polyline when a pen-up is required               *
19  *                                                                           *
20  * performance:                                                              *
21  *   trivial acceptance -- 9 + 31*n cycles for n points                      *
22  *   trivial rejection -- 9 + (29|32|35)*n cycles for n points               *
23  *   single clip of 2 pt polyline -- 106-108 cycles                          *
24  *   additional clips -- 14-15 cycles per plane                              *
25  *                                                                           *
26  * note:                                                                     *
27  *   This is a partial implementation.  This code segment includes the       *
28  *   trivial acceptance loop, the trivial rejection loop, and the code to    *
29  *   clip a line segment with its first point inside and its second outside  *
30  *   the viewbox.                                                            *
31  *************************************************************************/
32
33
34 /*************************************************************************
35  * 3264 register map                                                       *
36  *************************************************************************/
37        .reg    .f0     wc      /* clip-space coordinates */
38        .reg    .f1     xc
39        .reg    .f2     yc
40        .reg    .f3     zc
41
42        .reg    .f4     rwc     /* 1/wc */
43        .reg    .f5     mwc,z2  /* -wc */
44        .reg    .f6     s       /* scratch */
45        .reg    .f7     t       /* scratch */
46        .reg    .f8     zero,e1 /* 0.0 */
47        .reg    .f9     u,e2
48        .reg    .f10    xs      /* screen-space x */
49        .reg    .f11    ys      /* screen-space y */
50
51        .reg    .f12    w1      /* last point */
52        .reg    .f13    x1
53        .reg    .f14    y1
54        .reg    .f15    z1
55
56        .reg    .f16    a11     /* local coordinates to clip-space xform */
57        .reg    .f17    a12
58        .reg    .f18    a13
59        .reg    .f19    a14
60        .reg    .f20    a21
61        .reg    .f21    a22
62        .reg    .f22    a23
63        .reg    .f23    a24
```

B.3. 3-D Graphics Example, continued

```
64          .reg    .f24    a31
65          .reg    .f25    a32
66          .reg    .f26    a33
67          .reg    .f27    a34
68          .reg    .f28    a41
69          .reg    .f29    a42
70          .reg    .f30    a43
71          .reg    .f31    a44
72
73  /*************************************************************************
74   * 8137 register map                                                     *
75   *************************************************************************/
76          .reg    .r0     n       /* count of points */
77          .reg    .r1     ip      /* input pointer */
78          .reg    .r2     op      /* output pointer */
79          .reg    .r3     tp      /* screen-space transformation pointer */
80          .reg    .r4     opi     /* output pointer increment */
81          .reg    .r5     dp      /* data pointer */
82          .reg    .r6     cp2     /* codes for current point */
83          .reg    .r7     cp1     /* codes for preceding point */
84          .reg    .r8     wait    /* wait counts in cycles */
85          .reg    .r9     rx      /* scratch */
86
87
88          .text
89          .globl  polyline
90  /*************************************************************************
91   * Initialization                                                        *
92   *************************************************************************/
93  polyline:
94    pushs n;                      fclr t
95    movi 0,opi;         shbr acc1; fclr zero
96    addr ip,0,.quad;    ovneut                                  /* ->x[0] */
97
98  /*************************************************************************
99   *                ACCEPTANCE  LOOP  START                                *
100  *                                                                       *
101  * Trivial acceptance loop start.  Transform local coordinates (with w = 1)  *
102  * to clip-space coordinates:                                            *
103  *      xc = a11*x + a21*y + a31*z + a41                                 *
104  *      yc = a12*x + a22*y + a32*z + a42                                 *
105  *      zc = a13*x + a23*y + a33*z + a43                                 *
106  *      wc = a14*x + a24*y + a34*z + a44                                 *
107  *************************************************************************/
108 accept:
109   +addr ip,3,.quad;                                 dfixr s,xs
110 acc1:
111   addr ip,1,.quad;  dload .x;                       dfixr t,ys
112   addr op,0,.word;  dload .y;  dfmul .x,a14,t,.t0;  dfadd zero,.t0,u /* a14*x */
113   movi con,dp;      store xs;  dfmul .x,a11,s,.t1;  dfadd zero,.t1,u /* a11*x */
114   addr ip,2,.quad;             dfmul .y,a24,t,.t0;  dfadd a44,.t0,wc /* a24*y */
115   addr ip,1,.quad;  dload .x;  dfmul .y,a21,s,.t1;  dfadd a41,.t1,xc /* a21*y */
116   addr op,1,.word;  dload .y;  dfmul .x,a34,t,.t0;  dfadd wc,.t0,wc  /* a34*z */
117   movih con>>16,dp; store ys;  dfmul .x,a31,s,.t1;  dfadd xc,.t1,xc  /* a31*z */
118   addr dp,0,.quad;             dfmul .y,a13,t,.t0;  dfadd wc,.t0,wc  /* a13*x */
119   addr ip,1,.quad;  dload .x;  dfmul .y,a12,s,.t1;  dfadd xc,.t1,xc  /* a12*x */
120   mov cp2,cp1;      dload .y;  dfdiv .x,wc,rwc                      /* 1/wc */
121   addr ip,2,.quad;             dfmul .y,a22,s,.t1;  dfadd a24,.t1,yc /* a22*y */
122   addr dp,2,.quad;  dload .x;  dfmul .y,a23,t,.t0;  dfadd a43,.t0,zc /* a23*y */
123   addr tp,0,.quad;  dload zero;dfmul .x,a32,s,.t1;  dfadd yc,.t1,yc  /* a32*z */
124   addr dp,1,.quad;  dload .y;  dfmul .x,a33,t,.t0;  dfadd zc,.t0,zc  /* a33*z */
125   addr dp,3,.quad;  dload .x;  dfmul .y,xc,s,.t1;   dfadd yc,.t1,yc
126   addr tp,1,.quad;  dload z1;  dfmul .x,wc,mwc,.t0; dfadd zc,.t0,zc  /* -1*wc */
```

B.3. 3-D Graphics Example, continued

```
127
128 /*************************************************************************
129  * Test if the point is within the viewbox:                              *
130  *      -wc <= xc <= +wc  &&  -wc <= yc <= +wc  &&  0 <= zc <= +wc       *
131  * The six signs are accumulated in cp2 to assist in trivial rejection.  *
132  *************************************************************************/
133   movi 0,cp2;  dload .y;  dfcmp xc,wc,.gtz                    /* xc <= wc? */
134                           dfcmp xc,mwc,.ltz;   fbr .gtz,outxp1  /* xc >= -wc? */
135                           dfcmp yc,wc,.gtz;    fbr .ltz,outxn1  /* yc <= wc? */
136                           dfcmp yc,mwc,.ltz;   fbr .gtz,outyp1  /* yc >= -wc? */
137                           dfcmp zc,zero,.ltz;  fbr .ltz,outyn1  /* zc >= 0? */
138                           dfcmp zc,wc,.gtz;    fbr .ltz,outzn1  /* zc <= wc? */
139
140 /*************************************************************************
141  * Transform clip-space coordinates to screen-space:                     *
142  *      xs = fix[(xc/wc)*vsx + vcx]                                      *
143  *      ys = fix[(yc/wc)*vsy + vcy]                                      *
144  *************************************************************************/
145                               dfmul yc,.y,t;   fbr .gtz,outzp1
146   addr dp,3,.quad;            dfadd wc,zero,w1
147                     dstore zc; dfadd xc,zero,x1                 /* unload DSR */
148   addr tp,3,.quad;            dfadd yc,zero,y1                  /* wait 1 cyc */
149   addr tp,3,.quad;  dload .x;  dfmul s,rwc,s
150                     dload .y;  dfmul t,rwc,t
151   adda op,opi,op;             dfadd .x,s,s;    shsob accept
152   movi 8,opi;                 dfadd t,.y,t;    ovneut          /* update op */
153 /*************************************************************************
154  *                   A C C E P T A N C E   L O O P   E N D               *
155  *************************************************************************/
156                               dfixr s,xs
157   addr+ op,1,.word;           dfixr t,ys
158                     store xs                                    /* store last */
159   addr+ op,1,.word
160                     store ys
161                                          rts
162
163
164 /*************************************************************************
165  * Dispatch to the appropriate clipping routine                          *
166  *************************************************************************/
167 outxp1:
168                               dfsub w1,x1,e1;  br cxp1            /* w1-x1 */
169         addi10 cp2,0x20,cp2;  dfsub wc,xc,e2;  ovneut             /* w2-x2 */
170 outxn1:
171                               dfadd w1,x1,e1;  br cxn1            /* w1+x1 */
172         addi10 cp2,0x10,cp2;  dfadd wc,xc,e2;  ovneut             /* w2+x2 */
173 outyp1:
174                               dfsub w1,y1,e1;  br cyp1            /* w1-y1 */
175         addi10 cp2,0x08,cp2;  dfsub wc,yc,e2;  ovneut             /* w2-y2 */
176 outyn1:
177                               dfadd w1,y1,e1;  br cyn1            /* w1+y1 */
178         addi10 cp2,0x04,cp2;  dfadd wc,yc,e2;  ovneut             /* w2+y2 */
179 outzn1:
180                               dfmov z1,e1;     br czn1            /* z1 */
181         addi10 cp2,0x02,cp2;  dfmov zc,e2;     ovneut             /* z2 */
182 outzp1:
183                               dfsub w1,z1,e1;  br czp1            /* w1-z1 */
184         addi10 cp2,0x01,cp2;  dfsub wc,zc,e2;  ovneut             /* w2-z2 */
```

B.3. 3-D Graphics Example, continued

```
185
186
187 /*****************************************************************************
188  * Clip exiting segment at xc = wc plane.  At the intersection,              *
189  *                                                                           *
190  *          x = x1 + t*(xc-x1) where t = (w1-x1)/((w1-x1) - (wc-xc)).        *
191  *                                                                           *
192  * This simplifies to                                                        *
193  *                                                                           *
194  *               | x1  (w1-x1) |            | x1 e1 |                        *
195  *               | xc  (wc-xc) |            | xc e2 |                        *
196  *          x = -------   or  x = ----                    *
197  *              (wc-xc) - (w1-x1)            e2 - e1                        *
198  *                                                                           *
199  * Similar expressions can be derived for y, z, and w.  Since division by the *
200  * scale factor represented by the denominator in the expressions will be    *
201  * accomplished in the normal perspective division, the coordinates can be   *
202  * computed by determinants equal to or similar to the numerator above.      *
203  *****************************************************************************/
204 cxp1:
205   subai opi,0,opi;            dfmul wc,e1,s;   br .eqz,cxp2       /* w2*e1 */
206                               dfmul w1,e2,t                       /* w1*e2 */
207                               dfmul yc,e1,yc                      /* y2*e1 */
208   addr dp,1,.quad;            dfsub s,t,wc                        /* new w2 */
209   addr dp,0,.quad;  dload mwc; dfmul y1,e2,s                       /* y1*e2 */
210                     dload rwc; dfmul wc,mwc,mwc                    /* -wc */
211                               dfsub yc,s,yc                       /* new y2 */
212   addr dp,2,.quad;            dfmul zc,e1,s                       /* z2*e1 */
213                     dload zero;dfmul z1,e2,t                      /* z1*e2 */
214                               dfmov wc,xc;     br typ1            /* new x2 */
215                               dfsub s,t,zc;    ovneut             /* new z2 */
216
217 cxp2:                                          br typ2            /* 1st pt */
218   movi 0x3F,cp1;                               ovneut
219
220 /*****************************************************************************
221  * Clip exiting segment at xc = -wc plane                                    *
222  *****************************************************************************/
223 cxn1:
224   subai opi,0,opi;            dfmul wc,e1,s;   br .eqz,cxn2       /* w2*e1 */
225                               dfmul w1,e2,t                       /* w1*e2 */
226                               dfmul yc,e1,yc                      /* y2*e1 */
227   addr dp,1,.quad;            dfsub s,t,wc                        /* new w2 */
228   addr dp,0,.quad;  dload mwc; dfmul y1,e2,s                       /* y1*e2 */
229                     dload rwc; dfmul wc,mwc,mwc                    /* -wc */
230                               dfsub yc,s,yc                       /* new y2 */
231   addr dp,2,.quad;            dfmul zc,e1,s                       /* z2*e1 */
232                     dload zero;dfmul z1,e2,t                      /* z1*e2 */
233                               dfcmp zero,zero,.uord               /* clr cc */
234                               dfneg wc,xc;     br typ1            /* new x2 */
235                               dfsub s,t,zc;    ovneut             /* new z2 */
236
237 cxn2:                                          br tyn2            /* 1st pt */
238   movi 0x3F,cp1;                               ovneut
```

B.3. 3-D Graphics Example, continued

```
239
240 /*************************************************************************
241  * Clip exiting segment at yc = wc plane                                 *
242  *************************************************************************/
243 cyp1:
244   subai opi,0,opi;            dfmul wc,e1,s;   br .eqz,cyp2       /* w2*e1 */
245                               dfmul w1,e2,t                       /* w1*e2 */
246                               dfmul xc,e1,xc                      /* x2*e1 */
247   addr dp,0,.quad;            dfsub s,t,wc                        /* new w2 */
248   addr dp,1,.quad;  dload mwc; dfmul x1,e2,s                      /* x1*e2 */
249                     dload rwc; dfmul wc,mwc,mwc                   /* -wc */
250                               dfsub xc,s,xc                       /* new x2 */
251   addr dp,2,.quad;            dfmul zc,e1,s                       /* z2*e1 */
252                     dload zero;dfmul z1,e2,t                      /* z1*e2 */
253                               dfmov wc,yc;     br tzn1            /* new y2 */
254                               dfsub s,t,zc;    ovneut             /* new z2 */
255
256 cyp2:                                        br tzn2            /* 1st pt */
257   movi 0x3F,cp1;                             ovneut
258
259 /*************************************************************************
260  * Clip exiting segment at yc = -wc plane                                *
261  *************************************************************************/
262 cyn1:
263   subai opi,0,opi;            dfmul wc,e1,s;   br .eqz,cyn2       /* w2*e1 */
264                               dfmul w1,e2,t                       /* w1*e2 */
265                               dfmul xc,e1,xc                      /* x2*e1 */
266   addr dp,1,.quad;            dfsub s,t,wc                        /* new w2 */
267   addr dp,0,.quad;  dload mwc; dfmul x1,e2,s                      /* x1*e2 */
268                     dload mwc; dfmul wc,mwc,mwc                   /* -wc */
269                               dfsub xc,s,xc                       /* new x2 */
270   addr dp,2,.quad;            dfmul zc,e1,s                       /* z2*e1 */
271                     dload zero;dfmul z1,e2,t                      /* z1*e2 */
272                               dfcmp zero,zero,.uord               /* clr cc */
273                               dfneg wc,yc;     br tzn1            /* new y2 */
274                               dfsub s,t,zc;    ovneut             /* new z2 */
275
276 cyn2:                                        br tzp2            /* 1st pt */
277   movi 0x3F,cp1;                             ovneut
278
279 /*************************************************************************
280  * Clip exiting segment at zc = wc plane                                 *
281  *************************************************************************/
282 czp1:
283   subai opi,0,opi;            dfmul wc,e1,s;   br .eqz,czp2       /* w2*e1 */
284                               dfmul w1,e2,t                       /* w1*e2 */
285                               dfmul xc,e1,xc                      /* x2*e1 */
286   addr dp,1,.quad;            dfsub s,t,wc                        /* new w2 */
287   addr dp,0,.quad;  dload mwc; dfmul x1,e2,s                      /* x1*e2 */
288                     dload rwc; dfmul wc,mwc,mwc                   /* -wc */
289                               dfsub xc,s,xc                       /* new x2 */
290   addr dp,2,.quad;            dfmul yc,e1,s                       /* y2*e1 */
291                     dload zero;dfmul y1,e2,t                      /* y1*e2 */
292                               dfcmp zero,zero,.uord               /* clr cc */
293                               dfmov wc,zc;     br ss1a            /* new z2 */
294                               dfsub s,t,yc;    ovneut             /* new y2 */
295
296 czp2:                                        br ss2r            /* 1st pt */
297   movi 0x3F,cp1;                             ovneut
```

B.3. 3-D Graphics Example, continued

```
298
299 /*****************************************************************************
300  * Clip exiting segment at zc = 0 plane                                      *
301  *****************************************************************************/
302 czn1:
303   subai opi,0,opi;              dfmul wc,e1,s;   br .eqz,czn2       /* w2*e1 */
304                                 dfmul w1,e2,t                       /* w1*e1 */
305                                 dfmul xc,e1,xc                      /* x2*e1 */
306   addr dp,1,.quad;              dfsub s,t,wc                        /* new w2 */
307   addr dp,0,.quad;  dload mwc; dfmul x1,e2,s                        /* x1*e2 */
308                     dload rwc; dfmul wc,mwc,mwc                     /* -wc */
309                                 dfsub xc,s,xc                       /* new x2 */
310   addr dp,2,.quad;              dfmul yc,e1,s                       /* y2*e1 */
311                     dload zero;dfmul y1,e2,t                        /* y1*e2 */
312                                 fclr zc;         br ss1a            /* new z2 */
313                                 dfsub s,t,yc;    ovneut             /* new yc */
314
315 czn2:                                          br ss1r            /* 1st pt */
316   movi 0x3F,cp1;                               ovneut
317
318
319 /*****************************************************************************
320  * Test if the clipped point is within the viewbox:                          *
321  *      -wc <= xc <= +wc  &&  -wc <= yc <= +wc  &&  0 <= zc <= +wc           *
322  * The six signs are accumulated in cp2 to assist in trivial rejection.      *
323  *****************************************************************************/
324 typ1:                     dfcmp yc,wc,.gtz;    fbr .ltz,outxn3  /* yc <= wc? */
325 tyn1:                     dfcmp yc,mwc,.ltz;   fbr .gtz,outyp3  /* yc >= -wc? */
326 tzn1:                     dfcmp zc,zero,.ltz;  fbr .ltz,outyn3  /* zc >= 0? */
327 tzp1:                     dfcmp zc,wc,.gtz;    fbr .ltz,outzn3  /* zc <= wc? */
328 ss1a:                                          fbr .gtz,outzp3
329 ss2a:                     dfdiv rwc,wc,rwc;    br ss3a          /* 1/wc */
330
331 /*****************************************************************************
332  * Dispatch to the appropriate clipping routine                              *
333  *****************************************************************************/
334 outxp3:
335                                 dfsub w1,x1,e1;  br cxp1            /* w1-x1 */
336         addi10 cp2,0x20,cp2;    dfsub wc,xc,e2;  ovneut             /* w2-x2 */
337 outxn3:
338                                 dfadd w1,x1,e1;  br cxn1            /* w1+x1 */
339         addi10 cp2,0x10,cp2;    dfadd wc,xc,e2;  ovneut             /* w2+x2 */
340 outyp3:
341                                 dfsub w1,y1,e1;  br cyp1            /* w1-y1 */
342         addi10 cp2,0x08,cp2;    dfsub wc,yc,e2;  ovneut             /* w2-y2 */
343 outyn3:
344                                 dfadd w1,y1,e1;  br cyn1            /* w1+y1 */
345         addi10 cp2,0x04,cp2;    dfadd wc,yc,e2;  ovneut             /* w2+y2 */
346 outzn3:
347                                 dfmov z1,e1;     br czn1            /* z1 */
348         addi10 cp2,0x02,cp2;    dfmov zc,e2;     ovneut             /* z2 */
349 outzp3:
350                                 dfsub w1,z1,e1;  br czp1            /* w1-z1 */
351         addi10 cp2,0x01,cp2;    dfsub wc,zc,e2;  ovneut             /* w2-z2 */
352
```

B.3. 3-D Graphics Example, continued

```
353
354 /*************************************************************************
355  * Transform clip-space coordinates to screen-space:                     *
356  *      xs = fix[(xc/wc)*vsx + vcx]                                      *
357  *      ys = fix[(yc/wc)*vsy + vcy]                                      *
358  *************************************************************************/
359 ss3a:
360   addr tp,0,.quad
361   addr tp,1,.quad;  dload .x
362                     dload .y;  dfmul .x,xc,s
363   movi 8,wait;                 dfmul yc,.y,t
364 ss4a: addi -2,wait,wait;                        br .gtz,ss4a    /* wait for / */
365                              dfadd wc,zero,w1
366                              dfadd xc,zero,x1
367   addr tp,2,.quad;           dfadd yc,zero,y1
368   addr tp,3,.quad;  dload .x;dfadd zc,zero,z1
369   addr op,0,.word;  dload .y;dfmul s,rwc,s
370                     store xs;dfmul t,rwc,t
371   addr op,1,.word;           dfadd .x,s,s
372   adda op,opi,op;   store ys;dfadd t,.y,t                      /* update op */
373   movi 8,opi;                dfixr s,xs
374   +addr ip,3,.quad;          dfixr t,ys                        /* ->x[i+1] */
375   addr+ op,1,.word
376                     store xs                                   /* output pt */
377   addr+ op,1,.word
378   movi 0,opi;       store ys                                 /* pipe empty */
379                                                 bsr outpl      /* put pline */
380                                                 br ss3r        /* go reject */
381
382
383
384 /*************************************************************************
385  *                REJECTION   LOOP  START                                *
386  *                                                                       *
387  * First transform local coordinates (with w = 1) to clip-space coordinates:  *
388  *      xc = a11*x + a21*y + a31*z + a41                                 *
389  *      yc = a12*x + a22*y + a32*z + a42                                 *
390  *      zc = a13*x + a23*y + a33*z + a43                                 *
391  *      wc = a14*x + a24*y + a34*z + a44                                 *
392  *************************************************************************/
393 reject: mov cp2,cp1;dload .x;  dfmov yc,y1
394   addr ip,1,.quad;             dfmul .x,a14,t,.t0;  dfadd zero,.t0,u /* a14*x */
395                     dload .y;  dfmul .x,a11,s,.t1;  dfadd zero,.t1,u /* a11*x */
396   addr ip,2,.quad;             dfmul .y,a24,t,.t0;  dfadd a44,.t0,wc /* a24*y */
397   movi.con,dp;      dload .x;  dfmul .y,a21,s,.t1;  dfadd a41,.t1,xc /* a21*y */
398   addr ip,0,.quad;             dfmul .x,a34,t,.t0;  dfadd wc,.t0,wc  /* a34*z */
399   movih con>>16,dp; dload .y;  dfmul .x,a31,s,.t1;  dfadd xc,.t1,xc  /* a31*z */
400   addr dp,0,.quad;             dfmul .y,a13,t,.t0;  dfadd wc,.t0,wc  /* a13*x */
401   addr ip,1,.quad;  dload .x;  dfmul .y,a12,s,.t1;  dfadd xc,.t1,xc  /* a12*x */
402                     dload .y;  dfdiv .x,wc,rwc                       /* 1/wc */
403   addr ip,2,.quad;             dfmul .y,a22,s,.t1;  dfadd a24,.t1,yc /* a22*y */
404   addr dp,2,.quad;  dload .x;  dfmul .y,a23,t,.t0;  dfadd a43,.t0,z2 /* a23*y */
405   addr dp,0,.quad;  dload zero;dfmul .x,a32,s,.t1;  dfadd yc,.t1,yc  /* a32*z */
406   addr dp,1,.quad;  dload .y;  dfmul .x,a33,t,.t0;  dfadd z2,.t0,z2  /* a33*z */
407   addr tp,1,.quad;  dload .x;  dfmul .y,zc,z1,.t1;  dfadd yc,.t1,yc
408   addr tp,0,.quad;  dload .y;  dfmul .x,wc,mwc,.t0; dfadd z2,.t0,zc  /* -1*wc */
```

B.3. 3-D Graphics Example, continued

```
409
410 /*************************************************************************
411  * Test if the point is within the viewbox and exit the loop if it is:    *
412  *      -wc <= xc <= +wc  &&  -wc <= yc <= +wc  &&  0 <= zc <= +wc        *
413  * The six condition codes are accumulated in cp2 to assist in rejection. *
414  *************************************************************************/
415     movi 0,cp2; dload .x; dfcmp xc,wc,.gtz                    /* xc <= wc? */
416                           dfcmp xc,mwc,.ltz;   fbr .gtz,outxp2  /* xc >= -wc? */
417 typ2:                     dfcmp yc,wc,.gtz;    fbr .ltz,outxn2  /* yc <= wc? */
418 tyn2:                     dfcmp yc,mwc,.ltz;   fbr .gtz,outyp2  /* yc >= -wc? */
419 tzn2:                     dfcmp zc,zero,.ltz;  fbr .ltz,outyn2  /* zc >= 0? */
420 tzp2:                     dfcmp zc,wc,.gtz;    fbr .ltz,outzn2  /* zc <= wc? */
421 ss1r:                     dfmul .x,xc,s;       fbr .gtz,outzp2
422 ss2r: and cp1,cp2,rx;     dfmul yc,.y,t;       br .eqz,outin    /* reject? */
423 ss3r:                     dfmov wc,w1;         shsob reject     /* loop end */
424       +addr ip,3,.quad;   dfmov xc,x1;         ovneut           /* ->x[i+1] */
425                                                rts
426 /*************************************************************************
427  *                    R E J E C T I O N    L O O P   E N D               *
428  *************************************************************************/
429
430
431 /*************************************************************************
432  * Each of the following pairs of instructions sets the appropriate bit in  *
433  * the code for the point and returns immediately to the testing sequence.  *
434  *************************************************************************/
435 outxp2:                                    br typ2
436         addi10 cp2,0x20,cp2;               ovneut
437 outxn2:                                    br tyn2
438         addi10 cp2,0x10,cp2;               ovneut
439 outyp2:                                    br tzn2
440         addi10 cp2,0x08,cp2;               ovneut
441 outyn2:                                    br tzp2
442         addi10 cp2,0x04,cp2;               ovneut
443 outzn2:                                    br ss1r
444         addi10 cp2,0x02,cp2;               ovneut
445 outzp2:                                    br ss2r
446         addi10 cp2,0x01,cp2;               ovneut
447
448
449 /*************************************************************************
450  * The current segment does not lie on the outside of a single clipping plane *
451  * (clp2 & clp1 == 0).  Hence the segment must be clipped.                 *
452  *************************************************************************/
453 outin:
454
455
456         .data
457 /*************************************************************************
458  * data area                                                              *
459  *************************************************************************/
460 con:    .dfloat 1.0
461         .dfloat -1.0
462         .dfloat 0.0
463 zx:     .dfloat 0.0
```

B.4. Single-Precision 3-D Graphics

This example is a slight variation of the previous example given for double-precision 3-D graphics. Only the trivial acceptance loop is presented. All operations have been translated to single-precision. The performance of this loop is shown in figure 211.

The performance improvement from 31 cycles to 29 cycles per point results from the reduced latency of a single-precision divide over a double-precision divide (11 cycles versus 17 cycles). This does, however, necessitate saving the previous point in memory instead of the register file of the 3x64. This causes additional overhead for initialization and for retrieving the previous point when a segment must be clipped. Also, this version of the loop expects the 2×2 screen-space transformation to be present in the 3x64's register file.

Unlike the previous example, this routine is not IEEE-interruptible. The only violation of IEEE-interruptibility rules occurs on line 107. This line loads into the .y register and also uses this register as a source operand during the same cycle.

Operation	Cycles Required
trivial acceptance	
(2 points)	71
(*n* points)	13 + 29*n*

Figure 211. Single-precision graphics performance

B.4. Single-Precision 3-D Graphics, continued

```
 1 /*****************************************************************************
 2  * polyline -- generates a sequence of screen-space coordinates, (x,y),      *
 3  *   given a list of local coordinates.  Coordinates are 32-bit floating     *
 4  *   point and all arithmetic is done in 32-bit floating-point format.       *
 5  *                                                                           *
 6  * input:                                                                    *
 7  *   .r0 = count of points in polyline                                       *
 8  *   .r1 = address of list of (x,y,z)'s (all w's assumed = 1)                *
 9  *   .r2 = address of area to receive list of screen-space (x,y)'s           *
10  *   .f12-.f15 = 2x2 transformation from clip-space to screen-space          *
11  *   .f16-.f31 = 4x4 transformation from local coordinates to clip-space     *
12  *                                                                           *
13  * output:                                                                   *
14  *   .r2 = address+2 of last point in list of (x,y)'s                        *
15  *   .f12-.f31 left undisturbed                                              *
16  *                                                                           *
17  * performance:                                                              *
18  *   trivial acceptance -- 13 + 29*n cycles for n points                     *
19  *                                                                           *
20  * note:                                                                     *
21  *   This is the trivial acceptance loop only.                               *
22  *****************************************************************************/
23
24
25 /*****************************************************************************
26  * 3264 register map                                                         *
27  *****************************************************************************/
28        .reg    .f0     wc      /* clip-space coordinates */
29        .reg    .f1     xc
30        .reg    .f2     yc
31        .reg    .f3     zc
32        .reg    .f4     rwc     /* 1/wc */
33        .reg    .f5     mwc, z2 /* -wc */
34        .reg    .f6     s       /* scratch */
35        .reg    .f7     t       /* scratch */
36        .reg    .f8     zero    /* 0.0 */
37        .reg    .f10    xs      /* screen-space x */
38        .reg    .f11    ys      /* screen-space y */
39
40        .reg    .f12    vsx     /* clip-space to screen-space xform */
41        .reg    .f13    vsy
42        .reg    .f14    vcx
43        .reg    .f15    vcy
44
45        .reg    .f16    a11     /* local coordinates to clip-space xform */
46        .reg    .f17    a12
47        .reg    .f18    a13
48        .reg    .f19    a14
49        .reg    .f20    a21
50        .reg    .f21    a22
51        .reg    .f22    a23
52        .reg    .f23    a24
53        .reg    .f24    a31
54        .reg    .f25    a32
55        .reg    .f26    a33
56        .reg    .f27    a34
57        .reg    .f28    a41
58        .reg    .f29    a42
59        .reg    .f30    a43
60        .reg    .f31    a44
```

B.4. Single-Precision 3-D Graphics, continued

```
 61
 62 /*****************************************************************************
 63  * 8137 register map                                                         *
 64  *****************************************************************************/
 65         .reg    .r0     n       /* count of points */
 66         .reg    .r1     ip      /* input pointer */
 67         .reg    .r2     op      /* output pointer */
 68         .reg    .r3     opi     /* output pointer increment */
 69         .reg    .r4     dp      /* data pointer */
 70         .reg    .r5     cp2     /* codes for current point */
 71         .reg    .r6     cp1     /* codes for preceding point */
 72         .reg    .r7     rx      /* scratch */
 73         .reg    .r8     lp      /* last point pointer */
 74         .reg    .r9     li      /* last point pointer increment */
 75
 76
 77         .text
 78         .globl  polyline
 79
 80 /*****************************************************************************
 81  * Initialization                                                            *
 82  *****************************************************************************/
 83 polyline:
 84   movi lpt,lp
 85   movih (lpt)>>16,lp
 86   pushs n
 87   movi 0,opi
 88   addai ip,-12,ip;              fclr s
 89   +addr lp,8,.word,dp;          fclr t                              /* ->1.0 */
 90
 91 /*****************************************************************************
 92  *                A C C E P T A N C E   L O O P   S T A R T                  *
 93  *                                                                           *
 94  * Trivial acceptance loop start.  Transform local coordinates (with w = 1)  *
 95  * to clip-space coordinates:                                                *
 96  *      xc = a11*x + a21*y + a31*z + a41                                     *
 97  *      yc = a12*x + a22*y + a32*z + a42                                     *
 98  *      zc = a13*x + a23*y + a33*z + a43                                     *
 99  *      wc = a14*x + a24*y + a34*z + a44                                     *
100  *****************************************************************************/
101 accept:
102   +addr ip,3,.word; fload rwc;                      fixr s,xs
103   addr ip,1,.word;  fload .x;                       fixr t,ys       /* x[i] */
104   addr op,0,.word;  fload .y;   fmul .x,a14,s,.t0;  fadd t,.t0,t    /* a14*x */
105   movi 16,li;       store xs;   fmul .x,a11,s,.t1;  fadd t,.t1,t    /* a11*x */
106   addr ip,2,.word;              fmul .y,a24,s,.t0;  fadd a44,.t0,wc /* a24*y */
107   addr op,1,.word;  fload .y;   fmul .y,a21,s,.t1;  fadd a41,.t1,xc /* a21*y */
108                     store ys;   fmul .y,a34,s,.t0;  fadd wc,.t0,wc  /* a34*z */
109   addr lp,3,.word;              fmul .y,a31,s,.t1;  fadd xc,.t1,xc  /* a31*z */
110   xor li,lp,lp;     fstore zc;  fmul .x,a13,s,.t0;  fadd wc,.t0,wc  /* a13*x */
111   addr ip,1,.word;              fmul .x,a12,s,.t1;  fadd xc,.t1,xc  /* a12*x */
112   addr ip,2,.word;  fload .y;   fdiv rwc,wc,rwc                     /* 1/wc */
113   addr lp,1,.word;  fload .x;   fmul .y,a22,s,.t1;  fadd a24,.t1,yc /* a22*y */
114   mov cp2,cp1;      fstore xc;  fmul .y,a23,s,.t0;  fadd a43,.t0,z2 /* a23*y */
115                                 fmul .x,a32,s,.t1;  fadd yc,.t1,yc  /* a32*z */
116   addr dp,1,.word;  fload .x;   fmul .x,a33,s,.t0;  fadd z2,.t0,z2  /* a33*z */
117   addr dp,2,.word;  fload .x;   fmul .x,vsx,s,.t1;  fadd yc,.t1,yc
118   addr lp,0,.word;  fload zero; fmul .x,wc,mwc,.t0; fadd z2,.t0,zc  /* -1*wc */
```

B.4. Single-Precision 3-D Graphics, continued

```
119
120 /*************************************************************************
121  * Test if the point is within the viewbox:                              *
122  *      -wc <= xc <= +wc  &&  -wc <= yc <= +wc  &&  0 <= zc <= +wc       *
123  * The six signs are accumulated in cp2 to assist in trivial rejection.  *
124  *************************************************************************/
125   movi 0,cp2;  fstore wc;  fcmp xc,wc,.gtz                    /* xc <= wc? */
126                            fcmp xc,mwc,.ltz;   fbr .gtz,outxp1  /* xc >= -wc? */
127                            fcmp yc,wc,.gtz;    fbr .ltz,outxn1  /* yc <= wc? */
128                            fcmp yc,mwc,.ltz;   fbr .gtz,outyp1  /* yc >= -wc? */
129                            fcmp zc,zero,.ltz;  fbr .ltz,outyn1  /* zc >= 0? */
130                            fcmp zc,wc,.gtz;    fbr .ltz,outzn1  /* zc <= wc? */
131
132 /*************************************************************************
133  * Transform clip-space coordinates to screen-space:                     *
134  *      xs = fix[(xc/wc)*vsx + vcx]                                      *
135  *      ys = fix[(yc/wc)*vsy + vcy]                                      *
136  *************************************************************************/
137                                          fbr .gtz,outzp1 /* unload DSR */
138                              fmul yc,vsy,t
139   addr lp,2,.word;           fmul s,rwc,s
140   adda op,opi,op;  fstore yc; fmul t,rwc,t
141   movi 8,opi;                fadd s,vcx,,s;  shsob accept
142   addr dp,0,.word;           fadd t,vcy,t;   ovneut
143 /*************************************************************************
144  *                  A C C E P T A N C E   L O O P   E N D                *
145  *************************************************************************/
146                              fixr s,xs
147   addr+ op,1,.word;          fixr t,ys
148                     store xs                              /* store last */
149   addr+ op,1,.word
150                     store ys
151                                          rts
152
153
154 /*************************************************************************
155  * Dispatch to the appropriate clipping routine                          *
156  *************************************************************************/
157 outxp1:
158 outxn1:
159 outyp1:
160 outyn1:
161 outzn1:
162 outzp1:
163   nop
164
165
166         .data
167 /*************************************************************************
168  * data area                                                             *
169  *************************************************************************/
170         .align 3
171         .float 0,0,0,0
172 lpt:    .float 0,0,0,0,0,0,0,0
173 con:    .float 1.0
174         .float -1.0
175         .float 0.0
```

B.5. Double-Precision Cosine

This example computes the double-precision cosine of its double-precision input argument. It is written for an XL-8164 system (32-bit data bus).

Lines 1–31 of the source listing give some preliminary information on the routine, including its performance (line 17–20). As shown, the routine requires 28 cycles for completion when $|x| \leq \pi/4$, and 34 cycles when $\pi/4 < x \leq 3\pi/4$.

The next section of the source listing, line 33–64, gives the register map for the 3x64 register file. This is a collection of ".reg" assembly directives that assign symbolic names to the registers that are used. The register map for the integer processor follows on line 66–71.

The data area is defined by line 73–99. The main components of the initialized data are the sine expansion coefficients (line 78–83) and the cosine expansion coefficients (line 84–89). Other required constants are also included. In all cases, these initialized values are double-precision floating point values.

Actual execution begins on lin 108. As indicated by the comment on line 105, the first segment of code determines if $|x| \leq \pi/4$ and does additional preliminary calculations. For $|x| \leq \pi/4$ the cosine expansion described on line 117–132 is used. This calculation involves evaluating the indicated polynomial, an operation that is carried out by line 133–156.

When $\pi/4 < |x| \leq 3\pi/4$, the input argument is first reduced as described by line 158–165 of the source. This operation is accomplished by line 166–172. For arguments in this range, the result is determined using a negative sine expansion since

$$\cos(x) = \sin[-(x - \pi/2)].$$

This computation is described on line 174–188 and performed by line 189–207.

For those cases when $3\pi/4 < |x|$ the argument is first reduced by the nearest multiple of $\pi/2$. This range reduction algorithm is described on line 209–226 of the source and implemented by line 227–246. For those arguments which are reduced by an even multiple of $\pi/2$ the evaluation is done using the cosine expansion of line 133–156. When an argument is reduced by an odd multiple of $\pi/2$, a negative sine expansion is used. For efficiency reasons, a second implementation of the sine expansion is used (line 263–289).

B.5. Double-Precision Cosine, continued

```
 1 /*****************************************************************************
 2  * dcos(x) -- compute the double precision cosine of the double precision    *
 3  *   input argument.  First the number is  reduced to a number from {-Pi/4   *
 4  *   to +Pi/4} while noting which quadrant the original number was in.       *
 5  *   Numbers between -Pi/4 and Pi/4 are computed using a cosine expansion    *
 6  *   and are positive.  Numbers between Pi/4 and 3Pi/4 are computed using a  *
 7  *   negative sin expansion.  Numbers between 3Pi/4 and 5Pi/4 are computed   *
 8  *   using a negative cos expansion.  Numbers between 5Pi/4 and 7Pi/4 are    *
 9  *   computed using a positive sin expansion.                                *
10  *                                                                           *
11  * input:                                                                    *
12  *   .f0 = double precision input argument in radians                        *
13  *                                                                           *
14  * output:                                                                   *
15  *   .f0 = double precision cosine of argument                               *
16  *                                                                           *
17  * performance:                                                              *
18  *   abs(x) <= pi/4         --  28 cycles                                    *
19  *   pi/4 < abs(x) <= 3pi/4 --  34 cycles                                    *
20  *   3pi/4 < abs(x)         --  60-65 cycles                                 *
21  *                                                                           *
22  * configuration:                                                            *
23  *   XL-8164 system (32-bit data bus)                                        *
24  *                                                                           *
25  * note:                                                                     *
26  *   This routine will signal an underflow when a tiny number is given as    *
27  *   the input argument.  Detecting this condition and not signaling an      *
28  *   underflow on tiny inputs would cost about 6 cycles on the most common   *
29  *   inputs.  This cost is felt to be too high for a feature (underflow      *
30  *   detection) which is rarely used.                                        *
31  *****************************************************************************/
32
33 /*****************************************************************************
34  * 3x64 register map                                                         *
35  *****************************************************************************/
36   .reg .f0,x             /* input argument */
37   .reg .f0,result        /* by convention, result is returned in .f0 */
38   .reg .f1,pi4           /* Pi / 4 */
39   .reg .f2,ftmp1         /* temp necessary for muladd and chained ops */
40   .reg .f3,ftmp2         /* temp necessary for muladd and chained ops also */
41   .reg .f4,pi2           /* Pi / 2 */
42   .reg .f4,one           /* 1.0 */
43   .reg .f5,x1            /* x' -- the value used in the polynomial expansion */
44   .reg .f6,c6            /* polynomial constant */
45   .reg .f7,c5            /* polynomial constant */
46   .reg .f8,c4            /* polynomial constant */
47   .reg .f6,c3            /* polynomial constant */
48   .reg .f7,c2            /* polynomial constant */
49   .reg .f8,c1            /* polynomial constant */
50   .reg .f6,half          /* 0.5 */
51   .reg .f7,huge          /* maximum of range */
52   .reg .f9,acc1          /* first accumulator */
53   .reg .f10,acc2         /* second accumulator */
54   .reg .f10,n            /* float(fix(abs(x) * 4 / Pi + 0.5)) */
55   .reg .f11,x3           /*  x'*x'*x'  */
56   .reg .f12,xsq          /*  x'*x'  */
57   .reg .f11,x4           /*  x'*x'*x'*x'  */
58   .reg .f13,zero         /* 0.0 */
59   .reg .f14,abs_x        /* absolute value of x */
60   .reg .f15,_2pi         /* 2 / Pi */
61   .reg .f15,fn           /* fix(abs(x) * 4 / Pi + 0.5) */
62   .reg .y,pi2_1          /* first 26 bits of Pi/2 */
63   .reg .x,pi2_2          /* second 26 bits of Pi/2 */
64   .reg .y,pi2_3          /* last 53 bits of Pi/2*/
```

B.5. Double-Precision Cosine, continued

```
 65
 66 /*****************************************************************************
 67  * 8137 register map                                                         *
 68  *****************************************************************************/
 69   .reg .r0,cff          /* address of the coefficient and constants */
 70   .reg .r1,rn           /* number of Pi/2's subtracted to bring arg in range */
 71   .reg .r2,temp         /* scratch */
 72
 73 /*****************************************************************************
 74  * data area                                                                 *
 75  *****************************************************************************/
 76   .data
 77 CFF:
 78 S1:     .word 0x55555542,0xbfc55555    /* These constants have been modified */
 79 S2:     .word 0x1110f304,0x3f811111    /* to protect Weitek confidential     */
 80 S3:     .word 0x19b92303,0xbf2a01a0    /* information.  When these values    */
 81 S4:     .word 0x51e84125,0x3ec71de3    /* used, results should remain within */
 82 S5:     .word 0xaca1c66d,0xbe5ae5e2    /* +|- 10 ulp.                        */
 83 S6:     .word 0xb328da9f,0x3de5d83d
 84 C1:     .word 0xffffff68,0xbfdfffff
 85 C2:     .word 0x5554ddb8,0x3fa55555
 86 C3:     .word 0x16338b21,0xbf56c16c
 87 C4:     .word 0x78f60d0a,0x3efa019f
 88 C5:     .word 0x85a1fefe,0xbe927df1
 89 C6:     .word 0xb175ae94,0x3e21b803
 90 _3PI4:  .word 0x7f3321d2,0x4002d97c
 91 PI4:    .word 0x54442d18,0x3fe921fb
 92 _2PI:   .word 0x6dc9c883,0x3fe45f30
 93 HALF:   .word 0x00000000,0x3fe00000
 94 HUGE:   .word 0x00000000,0x41845f30
 95 ONE:    .word 0x00000000,0x3ff00000
 96 PI2_1:  .word 0x54000000,0x3ff921fb
 97 PI2_2:  .word 0x00000000,0x3e110b46
 98 PI2_3:  .word 0x00000000,0x3c91a626
 99 TEMP:   .word 0x0,0x0
100
101
102   .text
103   .globl dcos
104 /*****************************************************************************
105  * First determine if abs(x) <= pi/4, and compute x**2 and x**4.              *
106  *****************************************************************************/
107 dcos:
108   movi CFF,cff;                    dfmul x,x,xsq
109   movih CFF>>16,cff;               fclr zero
110   addrd cff,PI4-CFF;               dfabs x,abs_x
111   addrd cff,PI4-CFF+4;  dloadl pi4; dfmul xsq,xsq,x4
112   addrd cff,C6-CFF;     dloadm pi4
113   addrd cff,C6-CFF+4;   dloadl c6;  dfcmp abs_x,pi4, .gtz
114   movi 0,rn;            dloadm c6
115                                    dfadd pi4,pi4,pi2;    fbr .gtz, Above_Pi4
```

B.5. Double-Precision Cosine, continued

```
116
117 /*****************************************************************************
118  * Do the cos expansion.  The original expansion looked like:                *
119  *                                                                           *
120  *    drcos = 1d0+xsq*(c1+xsq*(c2+xsq*(c3+xsq*(c4+xsq*(c5+xsq*c6)))))        *
121  *                                                                           *
122  * But, we modified it to allow some parallelism:                            *
123  *                                                                           *
124  *   x4 = xsq*xsq                                                            *
125  *   tmp1 = (c2+x4*(c4+x4*c6))                                               *
126  *   tmp2 = (c1+x4*(c3+x4*c5))                                               *
127  *   drcos = 1d0+xsq*(tmp2+xsq*tmp1)                                         *
128  *                                                                           *
129  * This change saves roughly 6 cycles.  It costs a bit of precision,         *
130  * but our initial evaluation indicates that the loss of precision will      *
131  * be less than half an ulp.                                                 *
132  *****************************************************************************/
133 cos_ex:
134   addrd cff,C5-CFF;                dfmul x4,c6,acc1      /* x4*c6 */
135   addrd cff,C5-CFF+4;   dloadl c5
136   addrd cff,C4-CFF;     dloadm c5
137   addrd cff,C4-CFF+4;   dloadl c4;  dfmul x4,c5,acc2      /* x4*c5 */
138   addrd cff,C3-CFF;     dloadm c4
139   addrd cff,C3-CFF+4;   dloadl c3;  dfadd acc1,c4,acc1    /* c4+... */
140   addrd cff,C2-CFF+4;   dloadm c3
141   addrd cff,C2-CFF;     dloadm c2;  dfadd acc2,c3,acc2    /* c3+... */
142   addrd cff,C1-CFF+4;   dloadl c2;  dfmul x4,acc1,acc1    /* x4*(c4+... */
143   addrd cff,C1-CFF;     dloadm c1;  dfmul x4,acc2,acc2    /* x4*(c3+... */
144   addrd cff,ONE-CFF+4;  dloadl c1;  dfadd acc1,c2,acc1    /* c2 + x4*... */
145   addrd cff,ONE-CFF;    dloadm one; dfadd acc2,c1,acc2    /* c1 + x4*... */
146   dloadl one;                       dfmul acc1,xsq,acc1   /* xsq*tmp1 */
147   nop
148                                     dfadd acc1,acc2,acc1; /* tmp2+xsq*tmp1 */
149   nop
150                                     dfmul acc1,xsq,acc1   /* xsq*(tmp2+... */
151   ext rn,1,1,temp;      br .nez cneg_res                  /* odd mul of pi? */
152                         rts;        dfadd acc1,one,result /* 1d0+xsq*... */
153 cneg_res:
154                                     dfadd acc1,one,result /* 1d0+xsq*... */
155   nop
156                         rts;        dfsub zero,result,result /* fix sign */
157
158 /*****************************************************************************
159  * If abs(x) is in the range pi/4 to 3pi/4, then reduction of x is           *
160  * accomplished by subtracting Pi/2.  Note that Pi/2 differs from its 64-bit *
161  * ieee representation by only .26 lsbs, so even though the lsb of Pi/4 is   *
162  * less than the lsb of Pi/2, subtracting Pi/2 from x results in an error no *
163  * greater than .52 lsbs.  The result is then computed using a negative sin  *
164  * expansion.                                                                *
165  *****************************************************************************/
166 Above_Pi4:
167   addrd cff,S6-CFF+4;               dfsub abs_x,pi2,x1    /* |x|-pi/2 */
168   addrd cff,S6-CFF;     dloadm c6
169   addrd cff,S5-CFF;     dloadl c6;  dfmul x1,x1,xsq
170   addrd cff,S5-CFF+4;   dloadl c5;  dfcmp x1,pi4,.gtz     /* |x| <= 3pi/4? */
171                         dloadm c5;
172                         fbr .gtz,above_3Pi4; dfmul xsq,xsq,x4
```

B.5. Double-Precision Cosine, continued

```
173
174 /*************************************************************************
175  * Do the negative sin expansion.  The original expansion looked like:    *
176  *                                                                        *
177  *   drsin = x + x*xsq*(c1+xsq*(c2+xsq*(c3+xsq*(c4+xsq*(c5+xsq*c6)))))    *
178  *                                                                        *
179  * But, we modified it to allow some parallelism:                         *
180  *                                                                        *
181  *   x4 = xsq*xsq                                                         *
182  *   tmp1 = (c2+x4*(c4+x4*c6))                                            *
183  *   tmp2 = (c1+x4*(c3+x4*c5))                                            *
184  *   drsin = x+(x**3)*(tmp2+xsq*tmp1)                                     *
185  *                                                                        *
186  * Since cos(x) = sin(pi/2-x) = sin[-(x-pi/2)], the expansion is evaluated *
187  * with -(x-pi/2).                                                        *
188  *************************************************************************/
189   addrd cff,S5-CFF+4;   dloadl c5;  dfcmp x1,pi4,.gtz       /* |x| <= 3pi/4? */
190 sin_ex:
191   addrd cff,S4-CFF+4;               dfsub zero,x1,x1        /* -(x-pi/2) */
192   addrd cff,S4-CFF;     dloadm c4;  dfmul x4,c6,acc1        /* x4*c6 */
193   addrd cff,S3-CFF;     dloadl c4;  dfmul x4,c5,acc2        /* x4*c5 */
194   addrd cff,S3-CFF+4;   dloadl c3;  dfadd acc1,c4,acc1      /* c4+... */
195   addrd cff,S2-CFF+4;   dloadm c3
196   addrd cff,S2-CFF;     dloadm c2;  dfadd acc2,c3,acc2      /* c3+... */
197   addrd cff,S1-CFF+4;   dloadl c2;  dfmul x4,acc1,acc1      /* x4*(c4+... */
198   addrd cff,S1-CFF;     dloadm c1;  dfmul x4,acc2,acc2      /* x4*(c3+... */
199                         dloadl c1;  dfadd acc1,c2,acc1      /* c2+x4*... */
200                                     dfadd acc2,c1,acc2      /* c1+x4*... */
201                                     dfmul acc1,xsq,acc1     /* xsq*tmp1 */
202                                     dfmul xsq,x1,x3         /* x**3 */
203                                     dfadd acc1,acc2,acc1    /* tmp2+xsq*tmp1 */
204   nop
205                                     dfmul acc1,x3,acc1      /* x3*(tmp2+... */
206   nop
207                         rts;        dfadd acc1,x1,result    /* x + x3*(... */
208
209 /*************************************************************************
210  * At this point we know that x, the input parameter, has an absolute value *
211  * greater than 3Pi/4.  The following code performs the range reduction and *
212  * decides whether to do a sin or cos expansion.                          *
213  *                                                                        *
214  * Reduction of x is done by splitting Pi/2 into three parts.  The top two *
215  * parts each have 26 or fewer significant bits.  These can therefore be  *
216  * multiplied by integers with up to 27 significant bits without any loss of *
217  * precision.  So, the algorithm looks like:                              *
218  *                                                                        *
219  *   n = Float(Fix_truncate(Abs(x) * 2/Pi + 0.5))                         *
220  *       (i.e. x*2/Pi rounded to the nearest integer)                     *
221  *   x -= (n * Pi/2 part 1)     (most significant 26 bits of Pi)          *
222  *   x -= (n * Pi/2 part 2)     (next most significant 26 bits of Pi)     *
223  *   x -= (n * Pi/2 part 3)     (54 more bits of Pi)                      *
224  *                                                                        *
225  * If x is greater than "huge", then it is out of range.                  *
226  *************************************************************************/
227 above_3Pi4:
228   addrd cff,_2PI-CFF
229   addrd cff,_2PI-CFF+4;  dloadl _2pi
230   addrd cff,HALF-CFF+4;  dloadm _2pi
231   addrd cff,HALF-CFF;    dloadm half;  dfmul abs_x,_2pi,n;  /* x*2/pi */
232   addrd cff,PI2_1-CFF+4; dloadl half
233   addrd cff,PI2_1-CFF;   dloadm pi2_1; dfadd n,half,n
234   addrd cff,PI2_2-CFF+4; dloadl pi2_1
235   addrd cff,PI2_2-CFF;   dloadm pi2_2; dfix n,fn
```

B.5. Double-Precision Cosine, continued

```
236                          dloadl pi2_2
237                                        dfloat fn,n
238    addrd cff,HUGE-CFF+4
239    addrd cff,HUGE-CFF; dloadm huge;dfmul pi2_1,n,ftmp1,.t1; dfadd ftmp2,.t1,ftmp2
240    addrd cff,PI2_3-CFF+4; dloadl huge;
241    addrd cff,PI2_3-CFF; dloadm pi2_3; dfmul pi2_2,n,ftmp1,.t1; dfsub abs_x,.t1,x1
242    addrd cff,TEMP-CFF;    dloadl pi2_3
243                          store fn;     dfmul pi2_3,n,ftmp1,.t1; dfsub x1,.t1,x1
244    addrd cff,TEMP-CFF;                 dfcmp abs_x,huge,.gtz
245                          load rn;      dfmul .x,ftmp1,ftmp1,.t1; dfsub x1,.t1,x1
246                          fbr .gtz,out_of_range;
247    ext rn,0,1,temp;      br .eqz $+2                          /* even multiple? */
248                          br Sin_ex                            /* no, use sin */
249    addrd cff,C6-CFF;                   dfmul x1,x1,xsq
250    addrd cff,C6-CFF+4;   dloadl c6
251                          dloadm c6;
252                          br cos_ex;    dfmul xsq,xsq,x4       /* use cos */
253
254 /*****************************************************************************
255  * x is out of range -- return 0.                                            *
256  *****************************************************************************/
257 out_of_range:
258                          rts;          clr .f0
259
260 /*****************************************************************************
261  * The sin expansion is repeated here in order to save several cycles.       *
262  *****************************************************************************/
263 Sin_ex:
264    addrd cff,S6-CFF+4;              dfmul x1,x1,xsq
265    addrd cff,S6-CFF;     dloadm c6; dfsub zero,x1,x1          /* -(x-n*pi/2) */
266    addrd cff,S5-CFF+4;   dloadl c6; dfmul xsq,xsq,x4
267    addrd cff,S5-CFF;     dloadm c5
268    addrd cff,S4-CFF;     dloadl c5; dfmul x4,c6,acc1          /* x4*c6 */
269    addrd cff,S4-CFF+4;   dloadl c4; dfmul x4,c5,acc2          /* x4*c5 */
270    addrd cff,S3-CFF;     dloadm c4
271    addrd cff,S3-CFF+4;   dloadl c3; dfadd acc1,c4,acc1        /* c4+... */
272    addrd cff,S2-CFF+4;   dloadm c3
273    addrd cff,S2-CFF;     dloadm c2; dfadd acc2,c3,acc2        /* c3+... */
274    addrd cff,S1-CFF+4;   dloadl c2; dfmul x4,acc1,acc1        /* x4*(c4+... */
275    addrd cff,S1-CFF;     dloadm c1; dfmul x4,acc2,acc2        /* x4*(c3+... */
276                          dloadl c1; dfadd acc1,c2,acc1        /* c2+x4*... */
277                                     dfadd acc2,c1,acc2        /* c1+x4*... */
278                                     dfmul acc1,xsq,acc1       /* xsq*tmp1 */
279                                     dfmul xsq,x1,x3           /* x**3 */
280                                     dfadd acc1,acc2,acc1      /* tmp2+xsq*tmp1 */
281    nop
282                                     dfmul acc1,x3,acc1        /* x3*(tmp2+... */
283    ext rn,1,1,temp;      br .nez sneg_res                     /* odd mul of pi? */
284                          rts;       dfadd acc1,x1,result      /* x + x3*... */
285
286 sneg_res:
287                                     dfadd acc1,x1,result      /* x + x3*... */
288    nop
289                          rts;       dfsub zero,result,result
290
```

Appendix C. Known Bugs in the 3x64

If the result of any floating-point operation (single or double; add, subtract; mul, div, sqrt; convert; etc.) is bypassed into the second operand of a division operation of corresonding precision, the division operation may produce an incorrect result. The code below describes the situation in which the bug may occur:

```
(1)   fpop   .fa   .fb   .fc
      nop
      fdiv   .fc   .fc   .fe
```

For example (operands given in hex):

```
      fadd   40d00000   00000000   .fc
      nop
      fdiv   3f800000   .fc        .fd
```

In this example .fd should be .3e1d89d9, but the 3x64 would give 3e00000.

This bug is pattern-dependent; it occurs only if both the preceding fpop and the division operation have specific operands. We do not know what these patterns are. If the operands are random, the bug will occur in at most 1% of divide operations.

The bug affects the current production silicon.

WORKAROUND

Do not bypass into the second operand of division operations. For example, the following two code sequences will be executed correctly.

Current code should be examined for all occurrences of "2" and replaced with "3" (see figure 212).

This fix increases division latency by one cycle; in the case of single precision, from 10 to 11; double precision, from 17 to 18 cycles. Considering the relative frequency of division operations, the overall performance impact is negligible.

XL-8064 CUSTOMERS

Custsomers using WEITEK compilers in conjunction with the XL-8164 or XL-8364 chip set must replace the parallelizer and math libraries with the updated version that contains the fix described in figure 212. A tape that contains the updates is available from WEITEK free of charge.

```
(2)   fpop   .fa   .fb   .fc     Result in .fc is bypassed into the
      nop                       first, not second operand of the
      fdiv   .fc   .fc   .fe     division operation

(3)   fpop   .fa   .fb   .fc     Result in .fc is not bypassed at all
      nop
      nop
      fdiv   .fc   .fc   .fe
```

Figure 212. Workaround

Ordering Information

Ordering P/N	Cycle time	Package	Case temperature range	Comments
3164-GCD-100	100 ns	144-pin PGA	0–85° C	In production
3164-GCD-075	75 ns	144-pin PGA	0–85° C	In production
3164-GCD-060	60 ns	144-pin PGA	0–85° C	In production
3164-GCD-050	50 ns	144-pin PGA	0–85° C	Samples available*
3364-GCD-100	100ns	168-pin PGA	0–85° C	In production
3364-GCD-075	75 ns	168-pin PGA	0–85° C	In production
3364-GCD-060	60 ns	168-pin PGA	0–85° C	In production
3364-GCD-050	50 ns	168-pin PGA	0–85° C	Samples available*

* Status as of press time. Check with your sales representative for the latest availability.

Revision Summary

The following changes have been made since the April, 1989 data book:

1. Specifications for the 3x64-50 were added; those for the 3x64-50T were deleted
2. Appendix C, *Known Bugs in the 3x64*, was added
3. The *Ordering Information* section was updated