SECTION 3
CENTRAL PROCESSING UNIT

The PowerPC-based RISC processor (RCPU) used in the MPC500 family of micro-controllers integrates five independent execution units: an integer unit (IU), a load/store unit (LSU), and a branch processing unit (BPU), floating-point unit (FPU) and integer multiplier divider (IMD). The use of simple instructions with rapid execution times yields high efficiency and throughput for MPC555-based systems.

Most integer instructions execute in one clock cycle. Instructions can complete out of order for increased performance; however, the processor makes execution appear sequential.

This section provides an overview of the RCPU. For a detailed description of this processor, refer to the *RCPU Reference Manual* (RCPURM/AD).

3.1 RCPU Features

Major features of the RCPU include the following:

- High-performance microprocessor
  - Single clock-cycle execution for many instructions
- Five independent execution units and two register files
  - Independent LSU for load and store operations
  - BPU featuring static branch prediction
  - A 32-bit IU
  - Fully IEEE 754-compliant FPU for both single- and double-precision operations
  - Thirty-two general-purpose registers (GPRs) for integer operands
  - Thirty-two floating-point registers (FPRs) for single- or double-precision operands
- Facilities for enhanced system performance
  - Programmable big- and little-endian byte ordering
  - Atomic memory references
- In-system testability and debugging features
- High instruction and data throughput
  - Condition register (CR) look-ahead operations performed by BPU
  - Branch-folding capability during execution (zero-cycle branch execution time)
  - Programmable static branch prediction on unresolved conditional branches
  - A pre-fetch queue that can hold up to four instructions, providing look-ahead capability
  - Interlocked pipelines with feed-forwarding that control data dependencies in hardware

3.2 RCPU Block Diagram

Figure 3-1 provides a block diagram of the RCPU.

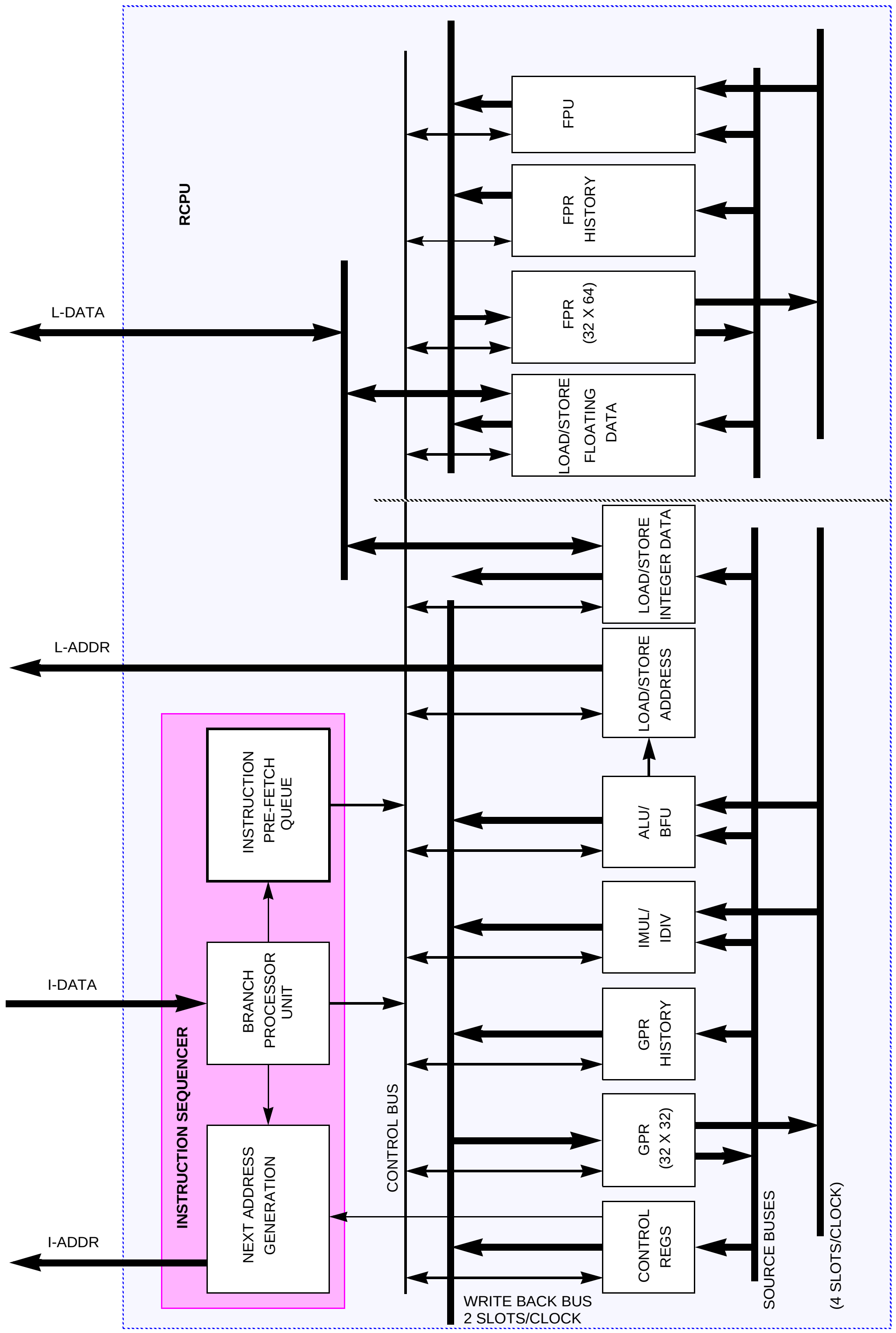


Figure 3-1 RCPU Block Diagram

3.3 Instruction Sequencer

The instruction sequencer provides centralized control over data flow between execution units and register files. The sequencer implements the basic instruction pipeline, fetches instructions from the memory system, issues them to available execution units, and maintains a state history so it can back the machine up in the event of an exception.

The instruction sequencer fetches instructions from the burst buffer controller into the instruction pre-fetch queue. The BPU extracts branch instructions from the pre-fetch queue and uses static branch prediction on unresolved conditional branches to allow the instruction unit to fetch instructions from a predicted target instruction stream while a conditional branch is evaluated. The BPU folds out branch instructions for unconditional branches or conditional branches unaffected by instructions in the execution stage.

Instructions issued beyond a predicted branch do not complete execution until the branch is resolved, preserving the programming model of sequential execution. If branch prediction is incorrect, the instruction unit flushes all predicted path instructions, and instructions are issued from the correct path.

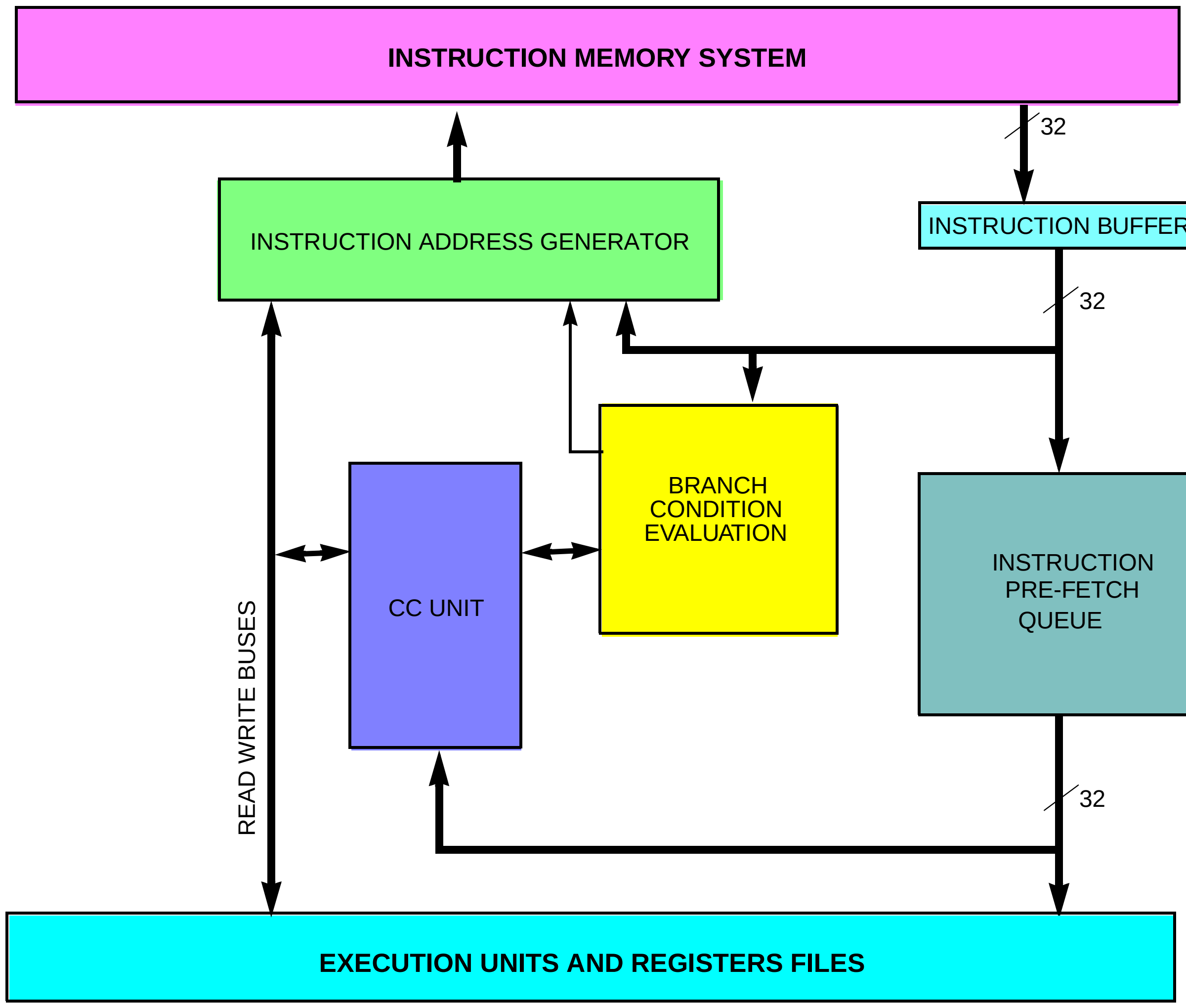

INSTRUCTION MEMORY SYSTEM
32
INSTRUCTION ADDRESS GENERATOR
INSTRUCTION BUFFER
32
BRANCH CONDITION EVALUATION
CC UNIT
INSTRUCTION PRE-FETCH QUEUE
READ WRITE BUSES
32
EXECUTION UNITS AND REGISTERS FILES

Figure 3-2 Sequencer Data Path

3.4 Independent Execution Units

The PowerPC architecture supports independent floating-point, integer, load/store, and branch processing execution units, making it possible to implement advanced features such as look-ahead operations. For example, since branch instructions do not depend on GPRs, branches can often be resolved early, eliminating stalls caused by taken branches.

Table 3-1 summarizes the RCPU execution units.

Table 3-1 RCPU Execution Units

Unit	Description
Branch processing unit (BPU)	Includes the implementation of all branch instructions
Load/store unit (LSU)	Includes implementation of all load and store instructions, whether defined as part of the integer processor or the floating-point processor
Integer unit (IU)	Includes implementation of all integer instructions except load/store instructions. This module includes the GPRs (including GPR history and scoreboard) and the following subunits: The IMUL-IDIV includes the implementation of the integer multiply and divide instructions. The ALU-BFU includes implementation of all integer logic, add and subtract instructions, and bit field instructions.
Floating-point unit (FPU)	Includes the FPRs (including FPR history and scoreboard) and the implementation of all floating-point instructions except load and store floating-point instructions

The following sections describe the execution units in greater detail.

3.4.1 Branch Processing Unit (BPU)

The BPU, located within the instruction sequencer, performs condition register look-ahead operations on conditional branches. The BPU looks through the instruction queue for a conditional branch instruction and attempts to resolve it early, achieving the effect of a zero-cycle branch in many cases.

The BPU uses a bit in the instruction encoding to predict the direction of the conditional branch. Therefore, when an unresolved conditional branch instruction is encountered, the processor pre-fetches instructions from the predicted target stream until the conditional branch is resolved.

The BPU contains an calculation feature to compute branch target addresses and three special-purpose, user-accessible registers: the link register (LR), the count register (CTR), and the condition register (CR). The BPU calculates the return pointer for subroutine calls and saves it into the LR. The LR also contains the branch target address for the branch conditional to link register (**bclr*x***) instruction. The CTR contains the branch target address for the branch conditional to count register (**bcctr*x***) instruction. The contents of the LR and CTR can be copied to or from any GPR. Because the BPU uses dedicated registers rather than general-purpose or floating-point registers, execution of branch instructions is independent from execution of integer instructions.

3.4.2 Integer Unit (IU)

The IU executes all integer processor instructions, except the integer storage access instructions, which are implemented by the load/store unit. The IU contains the following subunits:

- The IMUL–IDIV unit includes the implementation of the integer multiply and divide instructions.
- The ALU–BFU unit includes the implementation of all integer logic, add and subtract, and bit field instructions.

The IU also includes the integer exception register (XER) and the general-purpose register file.

IMUL–IDIV and ALU–BFU are implemented as separate execution units. The ALU–BFU unit can execute one instruction per clock cycle. IMUL–IDIV instructions require multiple clock cycles to execute. IMUL–IDIV is pipelined for multiply instructions, so that consecutive multiply instructions can be issued on consecutive clock cycles. Divide instructions are not pipelined; an integer divide instruction preceded or followed by an integer divide or multiply instruction results in a stall in the processor pipeline. Note that since IMUL–IDIV and ALU–BFU are implemented as separate execution units, an integer divide instruction preceded or followed by an ALU–BFU instruction does not cause a delay in the pipeline.

3.4.3 Load/Store Unit (LSU)

The load/store unit handles all data transfer between the general-purpose register file and the internal load/store bus (L-bus). The load/store unit is implemented as an independent execution unit so that stalls in the memory pipeline do not cause the master instruction pipeline to stall (unless there is a data dependency). The unit is fully pipelined so that memory instructions of any size may be issued on back-to-back cycles.

There is a 32-bit wide data path between the load/store unit and the general-purpose register file. Single-word accesses can be achieved with an internal on-chip data RAM, resulting in two clocks latency. Double-word accesses require two clocks, resulting in three clocks latency. Since the L-bus is 32 bits wide, double-word transfers require two bus accesses. The load/store unit performs zero-fill for byte and half-word transfers and sign extension for half-word transfers.

Addresses are formed by adding the source one register operand specified by the instruction (or zero) to either a source two register operand or to a 16-bit, immediate value embedded in the instruction.

3.4.4 Floating-Point Unit (FPU)

The FPU contains a double-precision multiply array, the floating-point status and control register (FPSCR), and the FPRs. The multiply-add array allows the MPC555 to efficiently implement floating-point operations such as multiply, multiply-add, and divide.

The MPC555 depends on a software envelope to fully implement the IEEE floating-point specification. Overflows, underflows, NaNs, and denormalized numbers cause floating-point assist exceptions that invoke a software routine to deliver (with hardware assistance) the correct IEEE result.

To accelerate time-critical operations and make them more deterministic, the MPC555 provides a mode of operation that avoids invoking the software envelope and attempts to deliver results in hardware that are adequate for most applications, if not in strict conformance with IEEE standards. In this mode, denormalized numbers, NaNs, and IEEE invalid operations are treated as legitimate, returning default results rather than causing floating-point assist exceptions.

3.5 Levels of the PowerPC Architecture

The PowerPC architecture consists of three layers. Adherence to the PowerPC architecture can be measured in terms of which of the following levels of the architecture are implemented:

- PowerPC user instruction set architecture (UISA) — Defines the base user-level instruction set, user-level registers, data types, floating-point exception model, memory models for a uniprocessor environment, and programming model for a uniprocessor environment.
- PowerPC virtual environment architecture (VEA) — Describes the memory model for a multiprocessor environment, and describes other aspects of virtual environments. Implementations that conform to the VEA also adhere to the UISA, but may not necessarily adhere to the OEA.
- PowerPC operating environment architecture (OEA) — Defines the memory management model, supervisor-level registers, synchronization requirements, and the exception model. Implementations that conform to the OEA also adhere to the UISA and the VEA.

3.6 RCPU Programming Model

The PowerPC architecture defines register-to-register operations for most computational instructions. Source operands for these instructions are accessed from the registers or are provided as immediate values embedded in the instruction opcode. The three-register instruction format allows specification of a target register distinct from the two source operands. Load and store instructions transfer data between memory and on-chip registers.

PowerPC processors have two levels of privilege: supervisor mode of operation (typically used by the operating environment) and user mode of operation (used by the application software). The programming models incorporate 32 GPRs, special-purpose registers (SPRs), and several miscellaneous registers.

Supervisor-level access is provided through the processor’s exception mechanism. That is, when an exception is taken (either due to an error or problem that needs to be serviced, or deliberately through the use of a trap instruction), the processor begins operating in supervisor mode. The level of access is indicated by the privilege-level (PR) bit in the machine state register (MSR).

Figure 3-3 shows the user-level and supervisor-level RCPU programming models and also illustrates the three levels of the PowerPC architecture. The numbers to the left of the SPRs indicate the decimal number that is used in the syntax of the instruction operands to access the register.

Note that registers such as the general-purpose registers (GPRs) are accessed through operands that are part of the instructions. Access to registers can be explicit (that is, through the use of specific instructions for that purpose such as move to special-purpose register (**mtspr**) and move from special-purpose register (**mfspr**) instructions) or implicitly as the part of the execution of an instruction. Some registers are accessed both explicitly and implicitly.

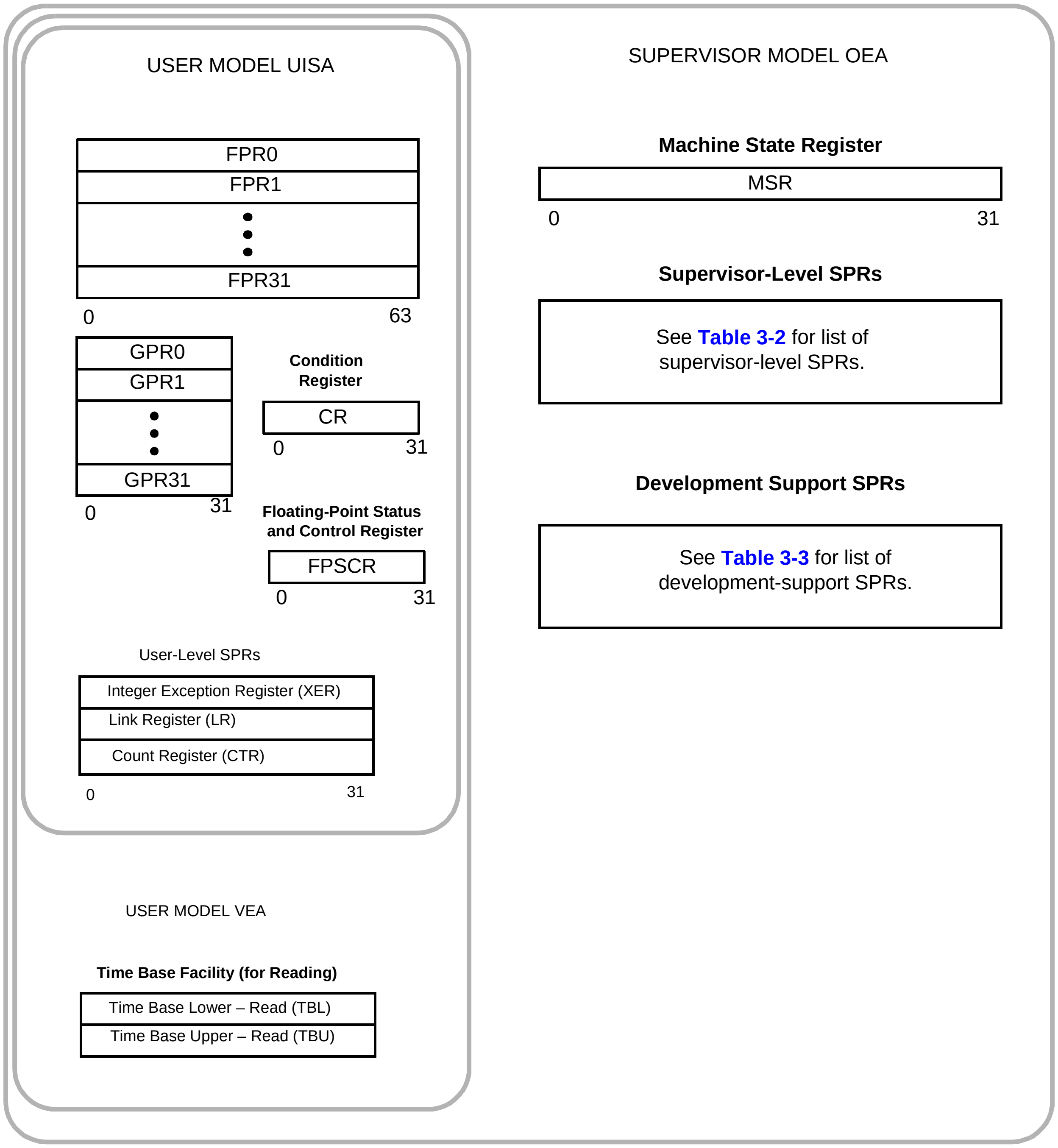

USER MODEL UISA
FPR0
FPR1
FPR31
0
63
GPR0
GPR1
GPR31
0
31
Condition Register
CR
0
31
Floating-Point Status and Control Register
FPSCR
0
31
User-Level SPRs
Integer Exception Register (XER)
Link Register (LR)
Count Register (CTR)
0
31
USER MODEL VEA
Time Base Facility (for Reading)
Time Base Lower – Read (TBL)
Time Base Upper – Read (TBU)
SUPERVISOR MODEL OEA
Machine State Register
MSR
0
31
Supervisor-Level SPRs
See Table 3-2 for list of supervisor-level SPRs.
Development Support SPRs
See Table 3-3 for list of development-support SPRs.

Figure 3-3 RCPU Programming Model

Table 3-2 lists the MPC555 supervisor-level registers.

Table 3-2 Supervisor-Level SPRs

SPR Number (Decimal)	Special-Purpose Register
18	DAE/Source Instruction Service Register (DSISR) See **3.9.2 DAE/Source Instruction Service Register (DSISR)** for bit descriptions.
19	Data Address Register (DAR) See **3.9.3 Data Address Register (DAR)** for bit descriptions.
22	Decrementer Register (DEC) See **3.9.5 Decrementer Register (DEC)** for bit descriptions.
26	Save and Restore Register 0 (SRR0) See **3.9.6 Machine Status Save/Restore Register 0 (SRR0)** for bit descriptions.
27	Save and Restore Register 1 (SRR1) See **3.9.7 Machine Status Save/Restore Register 1 (SRR1)** for bit descriptions.
80	External Interrupt Enable (EIE)[1] See **3.9.10.1 EIE, EID, and NRI Special-Purpose Registers** for bit descriptions.
81	External Interrupt Disable (EID)[1] See **3.9.10.1 EIE, EID, and NRI Special-Purpose Registers** for bit descriptions.
82	Non-Recoverable Interrupt (NRI)[1] See **3.9.10.1 EIE, EID, and NRI Special-Purpose Registers** for bit descriptions.
272	SPR General 0 (SPRG0) See **3.9.8 General SPRs (SPRG0–SPRG3)** for bit descriptions.
273	SPRGeneral 1 (SPRG1) See **3.9.8 General SPRs (SPRG0–SPRG3)** for bit descriptions.
274	SPR General 2 (SPRG2) See **3.9.8 General SPRs (SPRG0–SPRG3)** for bit descriptions.
275	SPR General 3 (SPRG3) See **3.9.8 General SPRs (SPRG0–SPRG3)** for bit descriptions.
284	Time Base Lower – Write (TBL) See **Table 3-14** for bit descriptions.
285	Time Base Upper – Write (TBU) See **Table 3-14** for bit descriptions.
287	Processor Version Register (PVR) See **Table 3-16** for bit descriptions.
528	IMPU Global Region Attribute (MI_GRA)[1] See **Table 4-7** for bit descriptions.
536	L2U Global Region Attribute (L2U_GRA)1 See **Table 11-10** for bit descriptions.
560	BBC Module Configuration Register (BBCMCR)1 See **Table 4-8** for bit descriptions.

Table 3-2 Supervisor-Level SPRs (Continued)

SPR Number (Decimal)	Special-Purpose Register
568	L2U Module Configuration Register (L2U_MCR)[1] See **Table 11-7** for bit descriptions.
784	IMPU Region Base Address 0 (MI_RBA0)1 See **Table 4-5** for bit descriptions.
785	IMPU Region Base Address 1 (MI_RBA1)1 See **Table 4-5** for bit descriptions.
786	IMPU Region Base Address 2 (MI_RBA2)1 See **Table 4-5** for bit descriptions.
787	IMPU Region Base Address 3 (MI_RBA3)[1] See **Table 4-5** for bit descriptions.
792	L2U Region Base Address Register 0 (L2U_RBA0)1 See **Table 11-8** for bit descriptions.
793	L2U Region Base Address Register 1 (L2U_RBA1)1 See **Table 11-8** for bit descriptions.
794	L2U Region Base Address Register 2 (L2U_RBA2)1 See **Table 11-8** for bit descriptions.
795	L2U Region Base Address Register 3 (L2U_RBA3)1 See **Table 11-8** for bit descriptions.
816	IMPU Region Attribute Register 0 (MI_RA0)1 See **Table 4-6** for bit descriptions.
817	IMPU Region Attribute Register 1 (MI_RA1)1 See **Table 4-6** for bit descriptions.
818	IMPU Region Attribute Register 2 (MI_RA2)1 See **Table 4-6** for bit descriptions.
819	IMPU Region Attribute Register 3 (MI_RA3)1 See **Table 4-6** for bit descriptions.
824	L2U Region Attribute Register 0 (L2U_RA0)1 See **Table 11-9** for bit descriptions.
825	L2U Region Attribute Register 1 (L2U_RA1)1 See **Table 11-9** for bit descriptions.
826	L2U Region Attribute Register 2 (L2U_RA2)1 See **Table 11-9** for bit descriptions.
827	L2U Region Attribute Register 3 (L2U_RA3)1 See **Table 11-9** for bit descriptions.
1022	Floating-Point Exception Cause Register (FPECR)1 See **3.9.10.2 Floating-Point Exception Cause Register (FPECR)** for bit descriptions.

NOTES:
1. Implementation-specific SPR.

Table 3-3 lists the MPC555 SPRs used for development support.

Table 3-3 Development Support SPRs[1]

SPR Number (Decimal)	Special-Purpose Register
144	Comparator A Value Register (CMPA) See **Table 21-17** for bit descriptions.
145	Comparator B Value Register (CMPB) See **Table 21-17** for bit descriptions.
146	Comparator C Value Register (CMPC) See **Table 21-17** for bit descriptions.
147	Comparator D Value Register (CMPD) See **Table 21-17** for bit descriptions.
148	Exception Cause Register (ECR) See **Table 21-26** for bit descriptions.
149	Debug Enable Register (DER) See **Table 21-27** for bit descriptions.
150	Breakpoint Counter A Value and Control (COUNTA) See **Table 21-24** for bit descriptions.
151	Breakpoint Counter B Value and Control (COUNTB) See **Table 21-25** for bit descriptions.
152	Comparator E Value Register (CMPE) See **Table 21-18** for bit descriptions.
153	Comparator F Value Register (CMPF) See **Table 21-18** for bit descriptions.
154	Comparator G Value Register (CMPG) See **Table 21-20** for bit descriptions.
155	Comparator H Value Register (CMPH) See **Table 21-20** for bit descriptions.
156	L-bus Support Comparators Control 1 (LCTRL1) See **Table 21-22** for bit descriptions.
157	L-bus Support Comparators Control 2 (LCTRL2) See **Table 21-23** for bit descriptions.
158	I-bus Support Control Register (ICTRL) See **Table 21-21** for bit descriptions.
159	Breakpoint Address Register (BAR) See **Table 21-19** for bit descriptions.
630	Development Port Data Register (DPDR) See **21.7.13** for bit descriptions.

NOTES:
1. All development-support SPRs are implementation-specific.

Where not otherwise noted, reserved fields in registers are ignored when written and return zero when read. An exception to this rule is XER[16:23]. These bits are set to the value written to them and return that value when read.

3.7 PowerPC UISA Register Set

The PowerPC UISA registers can be accessed by either user- or supervisor-level instructions. The general-purpose registers are accessed through instruction operands.

3.7.1 General-Purpose Registers (GPRs)

Integer data is manipulated in the integer unit's thirty-two 32-bit GPRs, shown below. These registers are accessed as source and destination registers through operands in the instruction syntax.

GPRs — General-Purpose Registers

MSB 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 LSB 31

GPR0
GPR1
. . .
. . .
GPR31

RESET: UNCHANGED

3.7.2 Floating-Point Registers (FPRs)

The PowerPC architecture provides thirty-two 64-bit FPRs. These registers are accessed as source and destination registers through operands in floating-point instructions. Each FPR supports the double-precision, floating-point format. Every instruction that interprets the contents of an FPR as a floating-point value uses the double-precision floating-point format for this interpretation. That is, all floating-point numbers are stored in double-precision format.

All floating-point arithmetic instructions operate on data located in FPRs and, with the exception of the compare instructions (which update the CR), place the result into an FPR. Information about the status of floating-point operations is placed into the floating-point status and control register (FPSCR) and in some cases, into the CR, after the completion of the operation's writeback stage. For information on how the CR is affected by floating-point operations, see **3.7.4 Condition Register (CR)**.

FPRs— Floating-Point Registers

MSB 0 LSB 63

FPR0
FPR1
. . .
. . .
FPR31

RESET: UNCHANGED

3.7.3 Floating-Point Status and Control Register (FPSCR)

The FPSCR controls the handling of floating-point exceptions and records status resulting from the floating-point operations. FPSCR[0:23] are status bits. FPSCR[24:31] are control bits.

FPSCR[0:12] and FPSCR[21:23] are floating-point exception condition bits. These bits are sticky, except for the floating-point enabled exception summary (FEX) and floating-point invalid operation exception summary (VX). Once set, sticky bits remain set until they are cleared by an **mcrfs**, **mtfsfi**, **mtfsf**, or **mtfsb0** instruction.

Table 3-4 summarizes which bits in the FPSCR are sticky status bits, which are normal status bits, and which are control bits.

Table 3-4 FPSCR Bit Categories

Bits	Type
[0], [3:12], [21:23]	Status, sticky
[1:2], [13:20]	Status, not sticky
[24:31]	Control

FEX and VX are the logical ORs of other FPSCR bits. Therefore these two bits are not listed among the FPSCR bits directly affected by the various instructions.

FPSCR — Floating-Point Status and Control Register

MSB 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
FX	FEX	VX	OX	UX	ZX	XX	VXS-NAN	VXISI	VXIDI	VXZDZ	VXIMZ	VXVC	FR	FI	FPRF0

RESET: UNCHANGED

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	LSB 31
FPRF[1:4]				0	VX-SOFT	VX-SQRT	VXCVI	VE	OE	UE	ZE	XE	NI	RN	

RESET: UNCHANGED

A listing of FPSCR bit settings is shown in **Table 3-5**.

Table 3-5 FPSCR Bit Settings

Bit(s)	Name	Description
0	FX	Floating-point exception summary. Every floating-point instruction implicitly sets FPSCR[FX] if that instruction causes any of the floating-point exception bits in the FPSCR to change from 0 to 1. The **mcrfs** instruction implicitly clears FPSCR[FX] if the FPSCR field containing FPSCR[FX] is copied. The **mtfsf**, **mtfsfi**, **mtfsb0**, and **mtfsb1** instructions can set or clear FPSCR[FX] explicitly. This is a sticky bit.
1	FEX	Floating-point enabled exception summary. This bit signals the occurrence of any of the enabled exception conditions. It is the logical OR of all the floating-point exception bits masked with their respective enable bits. The **mcrfs** instruction implicitly clears FPSCR[FEX] if the result of the logical OR described above becomes zero. The **mtfsf**, **mtfsfi**, **mtfsb0**, and **mtfsb1** instructions cannot set or clear FPSCR[FEX] explicitly. This is not a sticky bit.
2	VX	Floating-point invalid operation exception summary. This bit signals the occurrence of any invalid operation exception. It is the logical OR of all of the invalid operation exceptions. The **mcrfs** instruction implicitly clears FPSCR[VX] if the result of the logical OR described above becomes zero. The **mtfsf**, **mtfsfi**, **mtfsb0**, and **mtfsb1** instructions cannot set or clear FPSCR[VX] explicitly. This is not a sticky bit.
3	OX	Floating-point overflow exception. This is a sticky bit.
4	UX	Floating-point underflow exception. This is a sticky bit.
5	ZX	Floating-point zero divide exception. This is a sticky bit.
6	XX	Floating-point inexact exception. This is a sticky bit.
7	VXSNAN	Floating-point invalid operation exception for SNaN. This is a sticky bit.
8	VXISI	Floating-point invalid operation exception for ×-×. This is a sticky bit.
9	VXIDI	Floating-point invalid operation exception for ×/×. This is a sticky bit.
10	VXZDZ	Floating-point invalid operation exception for 0/0. This is a sticky bit.
11	VXIMZ	Floating-point invalid operation exception for ×*0. This is a sticky bit.
12	VXVC	Floating-point invalid operation exception for invalid compare. This is a sticky bit.
13	FR	Floating-point fraction rounded. The last floating-point instruction that potentially rounded the intermediate result incremented the fraction. This bit is not sticky.
14	FI	Floating-point fraction inexact. The last floating-point instruction that potentially rounded the intermediate result produced an inexact fraction or a disabled exponent overflow. This bit is not sticky.
[15:19]	FPRF	Floating-point result flags. This field is based on the value placed into the target register even if that value is undefined. Refer to **Table 3-6** for specific bit settings. 15 — Floating-point result class descriptor (C). Floating-point instructions other than the compare instructions may set this bit with the FPCC bits, to indicate the class of the result. 16–19 — Floating-point condition code (FPCC). Floating-point compare instructions always set one of the FPCC bits to one and the other three FPCC bits to zero. Other floating-point instructions may set the FPCC bits with the C bit, to indicate the class of the result. Note that in this case the high-order three bits of the FPCC retain their relational significance indicating that the value is less than, greater than, or equal to zero. 16 Floating-point less than or negative (FL or <) 17 Floating-point greater than or positive (FG or >) 18 Floating-point equal or zero (FE or =) 19 Floating-point unordered or NaN (FU or ?)
20	—	Reserved

Table 3-5 FPSCR Bit Settings (Continued)

Bit(s)	Name	Description
21	VXSOFT	Floating-point invalid operation exception for software request. This bit can be altered only by the **mcrfs**, **mtfsfi**, **mtfsf**, **mtfsb0**, or **mtfsb1** instructions. The purpose of VXSOFT is to allow software to cause an invalid operation condition for a condition that is not necessarily associated with the execution of a floating-point instruction. For example, it might be set by a program that computes a square root if the source operand is negative. This is a sticky bit.
22	VXSQRT	Floating-point invalid operation exception for invalid square root. This is a sticky bit. This guarantees that software can simulate **fsqrt** and **frsqrte**, and to provide a consistent interface to handle exceptions caused by square-root operations.
23	VXCVI	Floating-point invalid operation exception for invalid integer convert. This is a sticky bit.
24	VE	Floating-point invalid operation exception enable.
25	OE	Floating-point overflow exception enable.
26	UE	Floating-point underflow exception enable. This bit should not be used to determine whether denormalization should be performed on floating-point stores.
27	ZE	Floating-point zero divide exception enable.
28	XE	Floating-point inexact exception enable.
29	NI	Non-IEEE mode bit.
30–31	RN	Floating-point rounding control. 00 Round to nearest 01 Round toward zero 10 Round toward +infinity 11 Round toward -infinity

Table 3-6 illustrates the floating-point result flags that correspond to FPSCR[15:19].

Table 3-6 Floating-Point Result Flags in FPSCR

Result Flags (Bits 15–19) C<>=?	Result value class
10001	Quiet NaN
01001	– Infinity
01000	– Normalized number
11000	– Denormalized number
10010	– Zero
00010	+ Zero
10100	+ Denormalized number
00100	+ Normalized number
00101	+ Infinity

3.7.4 Condition Register (CR)

The condition register (CR) is a 32-bit register that reflects the result of certain operations and provides a mechanism for testing and branching. The bits in the CR are grouped into eight 4-bit fields, CR0 to CR7.

CR — Condition Register

MSB 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	LSB 31
CR0				CR1				CR2				CR3				CR4				CR5				CR6				CR7			

RESET: UNCHANGED

The CR fields can be set in the following ways:

- Specified fields of the CR can be set by a move instruction (**mtcrf**) to the CR from a GPR.
- Specified fields of the CR can be moved from one CR*x* field to another with the **mcrf** instruction.
- A specified field of the CR can be set by a move instruction (**mcrxr**) to the CR from the XER.
- Condition register logical instructions can be used to perform logical operations on specified bits in the condition register.
- CR0 can be the implicit result of an integer operation.
- A specified CR field can be the explicit result of an integer compare instruction.

Instructions are provided to test individual CR bits.

3.7.4.1 Condition Register CR0 Field Definition

In most integer instructions, when the CR is set to reflect the result of the operation (that is, when Rc = 1), and for **addic.**, **andi.**, and **andis.**, the first three bits of CR0 are set by an algebraic comparison of the result to zero; the fourth bit of CR0 is copied from XER[SO]. For integer instructions, CR[0:3] are set to reflect the result as a signed quantity. The result as an unsigned quantity or a bit string can be deduced from the EQ bit.

The CR0 bits are interpreted as shown in **Table 3-7**. If any portion of the result (the 32-bit value placed into the destination register) is undefined, the value placed in the first three bits of CR0 is undefined.

Table 3-7 Bit Settings for CR0 Field of CR

CR0 Bit	Description
0	Negative (LT) — This bit is set when the result is negative.
1	Positive (GT) — This bit is set when the result is positive (and not zero).
2	Zero (EQ) — This bit is set when the result is zero.
3	Summary overflow (SO) — This is a copy of the final state of XER[SO] at the completion of the instruction.

3.7.4.2 Condition Register CR1 Field Definition

In all floating-point instructions when the CR is set to reflect the result of the operation (that is, when Rc = 1), the CR1 field (bits 4 to 7 of the CR) is copied from FPSCR[0:3] to indicate the floating-point exception status. For more information about the FPSCR, see **3.7.3 Floating-Point Status and Control Register (FPSCR)**. The bit settings for the CR1 field are shown in **Table 3-8**.

Table 3-8 Bit Settings for CR1 Field of CR

CR1 Bit	Description
0	Floating-point exception (FX) — This is a copy of the final state of FPSCR[FX] at the completion of the instruction.
1	Floating-point enabled exception (FEX) — This is a copy of the final state of FPSCR[FEX] at the completion of the instruction.
2	Floating-point invalid exception (VX) — This is a copy of the final state of FPSCR[VX] at the completion of the instruction.
3	Floating-point overflow exception (OX) — This is a copy of the final state of FPSCR[OX] at the completion of the instruction.

3.7.4.3 Condition Register CR*n* Field — Compare Instruction

When a specified CR field is set by a compare instruction, the bits of the specified field are interpreted as shown in **Table 3-9**. A condition register field can also be accessed by the **mfcr**, **mcrf**, and **mtcrf** instructions.

Table 3-9 CR*n* Field Bit Settings for Compare Instructions

CR*n* Bit[1]	Description
0	Less than, floating-point less than (LT, FL). For integer compare instructions, (**r**A) < SIMM, UIMM, or (**r**B) (algebraic comparison) or (**r**A) SIMM, UIMM, or (**r**B) (logical comparison). For floating-point compare instructions, (**fr**A) < (**fr**B).
1	Greater than, floating-point greater than (GT, FG). For integer compare instructions, (**r**A) > SIMM, UIMM, or (**r**B) (algebraic comparison) or (**r**A) SIMM, UIMM, or (**r**B) (logical comparison). For floating-point compare instructions, (**fr**A) > (**fr**B).
2	Equal, floating-point equal (EQ, FE). For integer compare instructions, (**r**A) = SIMM, UIMM, or (**r**B). For floating-point compare instructions, (**fr**A) = (**fr**B).
3	Summary overflow, floating-point unordered (SO, FU). For integer compare instructions, this is a copy of the final state of XER[SO] at the completion of the instruction. For floating-point compare instructions, one or both of (**fr**A) and (**fr**B) is not a number (NaN).

NOTES:
1. Here, the bit indicates the bit number in any one of the four-bit subfields, CR0–CR7

3.7.5 Integer Exception Register (XER)

The integer exception register (XER) is a user-level, 32-bit register.

MSB 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	LSB 31
SO	OV	CA	Reserved																						BYTES						
RESET:																															
U	U	U	U	U	U	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	U	U	U	U	U	U	U

The bit definitions for XER, shown in **Table 3-10**, are based on the operation of an instruction considered as a whole, not on intermediate results. For example, the result of the Subtract from Carrying (**subfc***x*) instruction is specified as the sum of three values. This instruction sets bits in the XER based on the entire operation, not on an intermediate sum.

In most cases, reserved fields in registers are ignored when written to and return zero when read. However, XER[16:23] are set to the value written to them and return that value when read.

Table 3-10 Integer Exception Register Bit Definitions

Bit(s)	Name	Description
0	SO	Summary Overflow (SO) — The summary overflow bit is set whenever an instruction sets the overflow bit (OV) to indicate overflow and remains set until software clears it. It is not altered by compare instructions or other instructions that cannot overflow.
1	OV	Overflow (OV) — The overflow bit is set to indicate that an overflow has occurred during execution of an instruction. Integer and subtract instructions having OE=1 set OV if the carry out of bit 0 is not equal to the carry out of bit 1, and clear it otherwise. The OV bit is not altered by compare instructions or other instructions that cannot overflow.
2	CA	Carry (CA) — In general, the carry bit is set to indicate that a carry out of bit 0 occurred during execution of an instruction. Add carrying, subtract from carrying, add extended, and subtract from extended instructions set CA to one if there is a carry out of bit 0, and clear it otherwise. The CA bit is not altered by compare instructions or other instructions that cannot carry, except that shift right algebraic instructions set the CA bit to indicate whether any ‘1’ bits have been shifted out of a negative quantity.
3:24	—	Reserved
25:31	BYTES	This field specifies the number of bytes to be transferred by a Load String Word Indexed (**lswx**) or Store String Word Indexed (**stswx**) instruction.

3.7.6 Link Register (LR)

The 32-bit link register supplies the branch target address for the Branch Conditional to Link Register (**bclr***x*) instruction, and can be used to hold the logical address of the instruction that follows a branch and link instruction.

Note that although the two least-significant bits can accept any values written to them, they are ignored when the LR is used as an address.

Both conditional and unconditional branch instructions include the option of placing the effective address of the instruction following the branch instruction in the LR. This is done regardless of whether the branch is taken.

LR — Link Register **SPR 8**

MSB 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	LSB 31
Branch Address																															

RESET: UNCHANGED

3.7.7 Count Register (CTR)

The count register (CTR) is a 32-bit register for holding a loop count that can be decremented during execution of branch instructions that contain an appropriately coded BO field. If the value in CTR is 0 before being decremented, it is –1 afterward. The count register provides the branch target address for the Branch Conditional to Count Register (**bcctr*x***) instruction.

CTR — Count Register **SPR 9**

MSB 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	LSB 31
Loop Count																															

RESET: UNCHANGED

3.8 PowerPC VEA Register Set — Time Base

The PowerPC virtual environment architecture (VEA) defines registers in addition to those in the UISA register set. The PowerPC VEA register set can be accessed by all software with either user- or supervisor-level privileges.

The PowerPC VEA includes the time base facility (TB), a 64-bit structure that contains a 64-bit unsigned integer that is incremented periodically. The frequency at which the counter is updated is implementation-dependent. For details on the time base clock in the MPC555, refer to **6.7 MPC555 Time Base (TB)**, **8.6 MPC555 Internal Clock Signals**, and **8.12.1 System Clock Control Register (SCCR)**.

The TB consists of two 32-bit registers: time base upper (TBU) and time base lower (TBL). In the context of the VEA, user-level applications are permitted read-only access to the TB. The OEA defines supervisor-level access to the TB for writing values to the TB. Different SPR encodings are provided for reading and writing the time base.

TB — Time Base (Read Only) **SPR 268, 269**

0 — 31	32 — 63
TBU	TBL

RESET: UNCHANGED

Table 3-11 Time Base Field Definitions (Read Only)

Bits	Name	Description
0-31	TBU	Time Base (Upper) — The high-order 32 bits of the time base
32-63	TBL	Time Base (Lower) — The low-order 32 bits of the time base

In 32-bit PowerPC implementations such as the RCPU, it is not possible to read the entire 64-bit time base in a single instruction. The **mftb** simplified mnemonic copies the lower half of the time base register (TBL) to a GPR, and the **mftbu** simplified mnemonic copies the upper half of the time base (TBU) to a GPR.

3.9 PowerPC OEA Register Set

The PowerPC operating environment architecture (OEA) includes a number of SPRs and other registers that are accessible only by supervisor-level instructions. Some SPRs are RCPU-specific; some RCPU SPRs may not be implemented in other PowerPC processors, or may not be implemented in the same way.

3.9.1 Machine State Register (MSR)

The machine state register is a 32-bit register that defines the state of the processor. When an exception occurs, the current contents of the MSR are loaded into SRR1, and the MSR is updated to reflect the exception-processing machine state. The MSR can also be modified by the **mtmsr**, **sc**, and **rfi** instructions. It can be read by the **mfmsr** instruction.

MSR — Machine State Register

MSB 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
RESERVED													POW	0	ILE
RESET:															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	LSB 31
EE	PR	FP	ME	FE0	SE	BE	FE1	0	IP	IR	DR	RESERVED		RI	LE
RESET:															
0	0	0	U	0	0	0	0	0	ID1*	0	0	0	0	0	0

*Reset value of this bit depends on the value of the internal data bus line during reset.

Table 3-12 shows the bit definitions for the MSR.

Table 3-12 Machine State Register Bit Settings

Bit(s)	Name	Description
0:12	—	Reserved
13	POW	Power management enable 0 = Power management disabled (normal operation mode) 1 = Power management enabled (reduced power mode)
14	—	Reserved
15	ILE	Exception little-endian mode. When an exception occurs, this bit is copied into MSR[LE] to select the endian mode for the context established by the exception. 0 = Processor runs in big-endian mode during exception processing. 1 = Processor runs in little-endian mode during exception processing.
16	EE	External interrupt enable. Interrupts should only be negated while the EE bit is disabled (0). Software should disable interrupts in the CPU core prior to masking or disabling any interrupt which might be currently pending at the CPU core. For external interrupts, it is recommended that the edge triggered interrupt scheme be used. 0 = The processor delays recognition of external interrupts and decrementer exception conditions. 1 = The processor is enabled to take an external interrupt or the decrementer exception.
17	PR	Privilege level 0 = The processor can execute both user- and supervisor-level instructions. 1 = The processor can only execute user-level instructions.
18	FP	Floating-point available 0 = The processor prevents dispatch of floating-point instructions, including floating-point loads, stores and moves. Floating-point enabled program exceptions can still occur and the FPRs can still be accessed. 1 = The processor can execute floating-point instructions, and can take floating-point enabled exception type program exceptions.
19	ME	Machine check enable 0 = Machine check exceptions are disabled. 1 = Machine check exceptions are enabled.
20	FE0	Floating-point exception mode 0 (See **Table 3-13**.)
21	SE	Single-step trace enable 0 = The processor executes instructions normally. 1 = The processor generates a single-step trace exception upon the successful execution of the next instruction. When this bit is set, the processor dispatches instructions in strict program order. Successful execution means the instruction caused no other exception. Single-step tracing may not be present on all implementations.
22	BE	Branch trace enable 0 = No trace exception occurs when a branch instruction is completed 1 = Trace exception occurs when a branch instruction is completed
23	FE1	Floating-point exception mode 1 (See **Table 3-13**.)
24	—	Reserved
25	IP	Exception prefix. The setting of this bit specifies the location of the exception vector table. 0 = Exception vector table starts at the physical address 0x0000 0000. 1 = Exception vector table starts at the physical address 0xFFF0 0000.
26	IR	Instruction relocation. 0 = Instruction address translation is off, the BBC IMPU does not check for address permission attributes. 1 = Instruction address translation is on, the BBC IMPU checks for address permission attributes.

Table 3-12 Machine State Register Bit Settings (Continued)

Bit(s)	Name	Description
27	DR	Data relocation 0 = Data address translation is off, the L2U DMPU does not check for address permission attributes. 1 = Data address translation is on, the L2U DMPU checks for address permission attributes.
28:29	—	Reserved
30	RI	Recoverable exception (for machine check and non-maskable breakpoint exceptions) 0 = Machine state is not recoverable. 1 = Machine state is recoverable.
31	LE	Little-endian mode 0 = Processor operates in big-endian mode during normal processing. 1 = Processor operates in little-endian mode during normal processing.

The floating-point exception mode bits are interpreted as shown in **Table 3-13**.

Table 3-13 Floating-Point Exception Mode Bits

FE[0:1]	Mode
00	Ignore exceptions mode — Floating-point exceptions do not cause the floating-point assist error handler to be invoked.
01, 10, 11	Floating-point precise mode — The system floating-point assist error handler is invoked precisely at the instruction that caused the enabled exception.

3.9.2 DAE/Source Instruction Service Register (DSISR)

The 32-bit DSISR identifies the cause of data access and alignment exceptions.

DSISR — DAE/Source Instruction Service Register **SPR 18**

MSB 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	LSB 31
DSISR																															

RESET: UNCHANGED

3.9.3 Data Address Register (DAR)

After an alignment exception, the DAR is set to the effective address of a load or store element.

DAR — Data Address Register **SPR 19**

MSB 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	LSB 31
Data Address																															

RESET: UNCHANGED

3.9.4 Time Base Facility (TB) — OEA

As described in **3.8 PowerPC VEA Register Set — Time Base**, the time base (TB) provides a 64-bit incrementing counter. The VEA defines user-level, read-only access to the TB. Writing to the TB is reserved for supervisor-level applications such as operating systems and bootstrap routines. The OEA defines supervisor-level, write access to the TB.

TB — Time Base (Write Only) **SPR 284, 285**

0 31	32 63
TBU	TBL

RESET: UNCHANGED

Table 3-14 Time Base Field Definitions (Write Only)

Bits	Name	Description
0:31	TBU	Time Base (Upper) — The high-order 32 bits of the time base
32:63	TBL	Time Base (Lower) — The low-order 32 bits of the time base

The TB can be written to at the supervisor privilege level only. The **mttbl** and **mttbu** simplified mnemonics write the lower and upper halves of the TB, respectively. The **mtspr**, **mttbl**, and **mttbu** instructions treat TBL and TBU as separate 32-bit registers; setting one leaves the other unchanged. It is not possible to write the entire 64-bit time base in a single instruction.

For information about reading the time base, refer to **3.8 PowerPC VEA Register Set — Time Base**.

3.9.5 Decrementer Register (DEC)

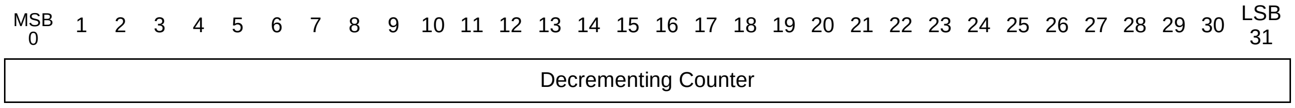
The decrementer (DEC, SPR 22) is a 32-bit decrementing counter defined by the MPC555 to provide a decrementer exception after a programmable delay. The DEC satisfies the following requirements:

- Loading a GPR from the DEC has no effect on the DEC.
- Storing a GPR to the DEC replaces the value in the DEC with the value in the GPR.
- Whenever bit 0 of the DEC changes from zero to one, a decrementer exception request (unless masked) is signaled. Multiple DEC exception requests may be received before the first exception occurs; however, any additional requests are canceled when the exception occurs for the first request.
- If the DEC is altered by software and the content of bit 0 is changed from zero to one, an exception request is signaled.
- PORESET resets and stops the decrementer, HRESET/SRESET do not.

The decrementer frequency is based on a subdivision of the processor clock. A bit in the system clock control register (SCCR) in the SIU determines the clock source of both the decrementer and the time base. For details on the decrementer and time base clock in the MPC555, refer to **6.6 MPC555 Decrementer**, **8.6 MPC555 Internal Clock Signals**, and **8.12.1 System Clock Control Register (SCCR)**.

The DEC does not run after power-up and must be enabled by setting the TBE bit in the TBSCR register, see Table 6-16. Power-on reset stops its counting and clears the register. A decrementer exception may be signaled to software prior to initialization.

DEC — Decrementer Register **SPR 22**



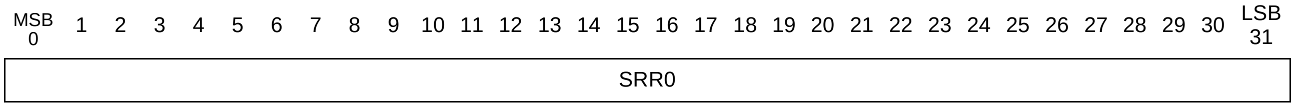
RESET: UNCHANGED

3.9.6 Machine Status Save/Restore Register 0 (SRR0)

The machine status save/restore register 0 (SRR0) is a 32-bit register that identifies where instruction execution should resume when an **rfi** instruction is executed following an exception. It also holds the effective address of the instruction that follows the System Call (**sc**) instruction.

When an exception occurs, SRR0 is set to point to an instruction such that all prior instructions have completed execution and no subsequent instruction has begun execution. The instruction addressed by SRR0 may not have completed execution, depending on the exception type. SRR0 addresses either the instruction causing the exception or the immediately following instruction. The instruction addressed can be determined from the exception type and status bits.

SRR0 — Machine Status Save/Restore Register 0 **SPR 26**



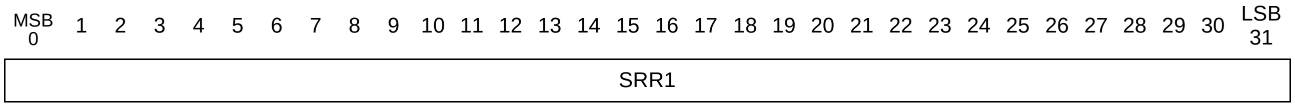
RESET: UNDEFINED

When an exception occurs, SRR0 is set to point to an instruction such that all prior instructions have completed execution and no subsequent instruction has begun execution. The instruction addressed by SRR0 may not have completed execution, depending on the exception type. SRR0 addresses either the instruction causing the exception or the immediately following instruction. The instruction addressed can be determined from the exception type and status bits.

3.9.7 Machine Status Save/Restore Register 1 (SRR1)

SRR1 is a 32-bit register used to save machine status on exceptions and to restore machine status when an **rfi** instruction is executed.

SRR1 — Machine Status Save/Restore Register 1 **SPR 27**



RESET: UNDEFINED

In general, when an exception occurs, SRR1[0:15] are loaded with exception-specific information, and MSR[16:31] are placed into SRR1[16:31].

3.9.8 General SPRs (SPRG0–SPRG3)

SPRG0–SPRG3 are 32-bit registers provided for general operating system use, such as performing a fast-state save and for supporting multiprocessor implementations. SPRG0–SPRG3 are shown below.

SPRG0–SPRG3 — General Special-Purpose Registers 0–3 **SPR 272 – SPR 275**

MSB 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	LSB 31
SPRG0																															
SPRG1																															
SPRG2																															
SPRG3																															

RESET: UNCHANGED

Uses for SPRG0–SPRG3 are shown in **Table 3-15**.

Table 3-15 Uses of SPRG0–SPRG3

Register	Description
SPRG0	Software may load a unique physical address in this register to identify an area of memory reserved for use by the exception handler. This area must be unique for each processor in the system.
SPRG1	This register may be used as a scratch register by the exception handler to save the content of a GPR. That GPR then can be loaded from SPRG0 and used as a base register to save other GPRs to memory.
SPRG2	This register may be used by the operating system as needed.
SPRG3	This register may be used by the operating system as needed.

3.9.9 Processor Version Register (PVR)

The PVR is a 32-bit, read-only register that identifies the version and revision level of the PowerPC processor. The contents of the PVR can be copied to a GPR by the **mfspr** instruction. Read access to the PVR is available in supervisor mode only; write access is not provided.

PVR — Processor Version Register **SPR 287**

MSB 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	LSB 31
VERSION																REVISION															

RESET: UNCHANGED

Table 3-16 Processor Version Register Bit Settings

Bit(s)	Name	Description
0:15	VERSION	A 16-bit number that identifies the version of the processor and of the PowerPC architecture. MPC555 value is 0x0002.
16:31	REVISION	A 16-bit number that distinguishes between various releases of a particular version. The MPC555 value is 0x0020.

3.9.10 Implementation-Specific SPRs

The MPC555 includes several implementation-specific SPRs that are not defined by the PowerPC architecture. These registers can be accessed by supervisor-level instructions only. These registers are listed in **Table 3-2** and **Table 3-3**.

3.9.10.1 EIE, EID, and NRI Special-Purpose Registers

The RCPU includes three implementation-specific SPRs to facilitate the software manipulation of the MSR[RI] and MSR[EE] bits. Issuing the **mtspr** instruction with one of these registers as an operand causes the RI and EE bits to be set or cleared as shown in **Table 3-17**.

A read (**mfspr**) of any of these locations is treated as an unimplemented instruction, resulting in a software emulation exception.

Table 3-17 EIE, EID, AND NRI Registers

SPR Number (Decimal)	Mnemonic	MSR[EE]	MSR[RI]
80	EIE	1	1
81	EID	0	1
82	NRI	0	0

3.9.10.2 Floating-Point Exception Cause Register (FPECR)

The FPECR is a 32-bit supervisor-level internal status and control register used by the floating-point assist firmware envelope. It contains four status bits indicating whether the result of the operation is tiny and whether any of three source operands are denormalized. In addition, it contains one control bit to enable or disable SIE mode. This register must not be accessed by user code.

FPECR — Floating-Point Exception Cause Register

MSB 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
SIE	RESERVED														
RESET:															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	LSB 31
RESERVED												DNC	DNB	DNA	TR
RESET:															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

A listing of FPECR bit settings is shown in **Table 3-18**.

Table 3-18 FPECR Bit Settings

Bit(s)	Name	Description
0	SIE	SIE mode control bit 0 = Disable SIE mode 1 = Enable SIE mode
[1:27]	—	Reserved
28	DNC	Source operand C denormalized status bit 0 = Source operand C is not denormalized 1 = Source operand C is denormalized
29	DNB	Source operand B denormalized status bit 0 = Source operand B is not denormalized 1 = Source operand B is denormalized
30	DNA	Source operand A denormalized status bit 0 = Source operand A is not denormalized 1 = Source operand A is denormalized
31	TR	Floating-point tiny result 0 = Floating-point result is not tiny 1 = Floating-point result is tiny

NOTE

Software must insert a **sync** instruction before reading the FPECR.

3.9.10.3 Additional Implementation-Specific Registers

Refer to the following sections for details on additional implementation-specific registers in the MPC555:

- **4.6 Burst Buffer Programming Model**
- **6.13.1.2 Internal Memory Map Register**
- **11.8 L2U Programming Model**
- **SECTION 21 DEVELOPMENT SUPPORT**

3.10 Instruction Set

All PowerPC instructions are encoded as single words (32 bits). Instruction formats are consistent among all instruction types, permitting efficient decoding to occur in parallel with operand accesses. This fixed instruction length and consistent format greatly simplifies instruction pipelining.

The PowerPC instructions are divided into the following categories:

- Integer instructions include computational and logical instructions.
  - Integer arithmetic instructions
  - Integer compare instructions
  - Integer logical instructions
  - Integer rotate and shift instructions
- Floating-point instructions include floating-point computational instructions, as well as instructions that affect the floating-point status and control register (FPSCR).
  - Floating-point arithmetic instructions
  - Floating-point multiply/add instructions
  - Floating-point rounding and conversion instructions
  - Floating-point compare instructions
  - Floating-point status and control instructions
- Load/store instructions include integer and floating-point load and store instructions.
  - Integer load and store instructions
  - Integer load and store multiple instructions
  - Floating-point load and store
  - Primitives used to construct atomic memory operations (**lwarx** and **stwcx.** instructions)
- Flow control instructions include branching instructions, condition register logical instructions, trap instructions, and other instructions that affect the instruction flow.
  - Branch and trap instructions
  - Condition register logical instructions
- Processor control instructions are used for synchronizing memory accesses.
  - Move to/from SPR instructions
  - Move to/from MSR
  - Synchronize
  - Instruction synchronize

Note that this grouping of the instructions does not indicate which execution unit executes a particular instruction or group of instructions.

Integer instructions operate on byte, half-word, and word operands. Floating-point instructions operate on single-precision (one word) and double-precision (one double word) floating-point operands. The PowerPC architecture uses instructions that are four bytes long and word-aligned. It provides for byte, half-word, and word operand loads and stores between memory and a set of 32 GPRs.

Computational instructions do not modify memory. To use a memory operand in a computation and then modify the same or another memory location, the memory contents must be loaded into a register, modified, and then written back to the target location with distinct instructions.

PowerPC processors follow the program flow when they are in the normal execution state. However, the flow of instructions can be interrupted directly by the execution of an instruction or by an asynchronous event. Either kind of exception may cause one of several components of the system software to be invoked.

3.10.1 Instruction Set Summary

Table 3-19 provides a summary of RCPU instructions. Refer to the *RCPU Reference Manual* (RCPURM/AD) for a detailed description of the instruction set.

Table 3-19 Instruction Set Summary

Mnemonic	Operand Syntax	Name
add (add. addo addo.)	rD,rA,rB	Add
addc (addc. addco addco.)	rD,rA,rB	Add Carrying
adde (adde. addeo addeo.)	rD,rA,rB	Add Extended
addi	rD,rA,SIMM	Add Immediate
addic	rD,rA,SIMM	Add Immediate Carrying
addic.	rD,rA,SIMM	Add Immediate Carrying and Record
addis	rD,rA,SIMM	Add Immediate Shifted
addme (addme. addmeo addmeo.)	rD,rA	Add to Minus One Extended
addze (addze. addzeo addzeo.)	rD,rA	Add to Zero Extended
and (and.)	rA,rS,rB	AND
andc (andc.)	rA,rS,rB	AND with Complement
andi.	rA,rS,UIMM	AND Immediate
andis.	rA,rS,UIMM	AND Immediate Shifted
b (ba bl bla)	target_addr	Branch
bc (bca bcl bcla)	BO,BI,target_addr	Branch Conditional
bcctr (bcctrl)	BO,BI	Branch Conditional to Count Register
bclr (bclrl)	BO,BI	Branch Conditional to Link Register
cmp	crfD,L,rA,rB	Compare
cmpi	crfD,L,rA,SIMM	Compare Immediate
cmpl	crfD,L,rA,rB	Compare Logical
cmpli	crfD,L,rA,UIMM	Compare Logical Immediate
cntlzw (cntlzw.)	rA,rS	Count Leading Zeros Word
crand	crbD,crbA,crbB	Condition Register AND
crandc	crbD,crbA, crbB	Condition Register AND with Complement
creqv	crbD,crbA, crbB	Condition Register Equivalent
crnand	crbD,crbA,crbB	Condition Register NAND
crnor	crbD,crbA,crbB	Condition Register NOR
cror	crbD,crbA,crbB	Condition Register OR

Table 3-19 Instruction Set Summary (Continued)

Mnemonic	Operand Syntax	Name
crorc	**crb**D,**crb**A, **crb**B	Condition Register OR with Complement
crxor	**crb**D,**crb**A,**crb**B	Condition Register XOR
divw (divw. divwo divwo.)	**r**D,**r**A,**r**B	Divide Word
divwu divwu. divwuo divwuo.	**r**D,**r**A,**r**B	Divide Word Unsigned
eieio	—	Enforce In-Order Execution of I/O
eqv (eqv.)	**r**A,**r**S,**r**B	Equivalent
extsb (extsb.)	**r**A,**r**S	Extend Sign Byte
extsh (extsh.)	**r**A,**r**S	Extend Sign Half Word
fabs (fabs.)	**fr**D,**fr**B	Floating Absolute Value
fadd (fadd.)	**fr**D,**fr**A,**fr**B	Floating Add (Double-Precision)
fadds (fadds.)	**fr**D,**fr**A,**fr**B	Floating Add Single
fcmpo	**crf**D,**fr**A,**fr**B	Floating Compare Ordered
fcmpu	**crf**D,**fr**A,**fr**B	Floating Compare Unordered
fctiw (fctiw.)	**fr**D,**fr**B	Floating Convert to Integer Word
fctiwz (fctiwz.)	**fr**D,**fr**B	Floating Convert to Integer Word with Round toward Zero
fdiv (fdiv.)	**fr**D,**fr**A,**fr**B	Floating Divide (Double-Precision)
fdivs (fdivs.)	**fr**D,**fr**A,**fr**B	Floating Divide Single
fmadd (fmadd.)	**fr**D,**fr**A,**fr**C,**fr**B	Floating Multiply-Add (Double-Precision)
fmadds (fmadds.)	**fr**D,**fr**A,**fr**C,**fr**B	Floating Multiply-Add Single
fmr (fmr.)	**fr**D,**fr**B	Floating Move Register
fmsub (fmsub.)	**fr**D,**fr**A,**fr**C,**fr**B	Floating Multiply-Subtract (Double-Precision)
fmsubs (fmsubs.)	**fr**D,**fr**A,**fr**C,**fr**B	Floating Multiply-Subtract Single
fmul (fmul.)	**fr**D,**fr**A,**fr**C	Floating Multiply (Double-Precision)
fmuls (fmuls.)	**fr**D,**fr**A,**fr**C	Floating Multiply Single
fnabs (fnabs.)	**fr**D,**fr**B	Floating Negative Absolute Value
fneg (fneg.)	**fr**D,**fr**B	Floating Negate
fnmadd (fnmadd.)	**fr**D,**fr**A,**fr**C,**fr**B	Floating Negative Multiply-Add (Double-Precision)
fnmadds (fnmadds.)	**fr**D,**fr**A,**fr**C,**fr**B	Floating Negative Multiply-Add Single
fnmsub (fnmsub.)	**fr**D,**fr**A,**fr**C,**fr**B	Floating Negative Multiply-Subtract (Double-Precision)
fnmsubs (fnmsubs.)	**fr**D,**fr**A,**fr**C,**fr**B	Floating Negative Multiply-Subtract Single
frsp (frsp.)	**fr**D,**fr**B	Floating Round to Single
fsub (fsub.)	**fr**D,**fr**A,**fr**B	Floating Subtract (Double-Precision)
fsubs (fsubs.)	**fr**D,**fr**A,**fr**B	Floating Subtract Single
isync	—	Instruction Synchronize
lbz	**r**D,d(**r**A)	Load Byte and Zero
lbzu	**r**D,d(**r**A)	Load Byte and Zero with Update
lbzux	**r**D,**r**A,**r**B	Load Byte and Zero with Update Indexed
lbzx	**r**D,**r**A,**r**B	Load Byte and Zero Indexed
lfd	**fr**D,d(**r**A)	Load Floating-Point Double
lfdu	**fr**D,d(**r**A)	Load Floating-Point Double with Update

Table 3-19 Instruction Set Summary (Continued)

Mnemonic	Operand Syntax	Name
lfdux	**fr**D,**r**A,**r**B	Load Floating-Point Double with Update Indexed
lfdx	**fr**D,**r**A,**r**B	Load Floating-Point Double Indexed
lfs	**fr**D,d(**r**A)	Load Floating-Point Single
lfsu	**fr**D,d(**r**A)	Load Floating-Point Single with Update
lfsux	**fr**D,**r**A,**r**B	Load Floating-Point Single with Update Indexed
lfsx	**fr**D,**r**A,**r**B	Load Floating-Point Single Indexed
lha	**r**D,d(**r**A)	Load Half-Word Algebraic
lhau	**r**D,d(**r**A)	Load Half-Word Algebraic with Update
lhaux	**r**D,**r**A,**r**B	Load Half-Word Algebraic with Update Indexed
lhax	**r**D,**r**A,**r**B	Load Half-Word Algebraic Indexed
lhbrx	**r**D,**r**A,**r**B	Load Half-Word Byte-Reverse Indexed
lhz	**r**D,d(**r**A)	Load Half-Word and Zero
lhzu	**r**D,d(**r**A)	Load Half-Word and Zero with Update
lhzux	**r**D,**r**A,**r**B	Load Hal-Word and Zero with Update Indexed
lhzx	**r**D,**r**A,**r**B	Load Half-Word and Zero Indexed
lmw	**r**D,d(**r**A)	Load Multiple Word
lswi	**r**D,**r**A,NB	Load String Word Immediate
lswx	**r**D,**r**A,**r**B	Load String Word Indexed
lwarx	**r**D,**r**A,**r**B	Load Word and Reserve Indexed
lwbrx	**r**D,**r**A,**r**B	Load Word Byte-Reverse Indexed
lwz	**r**D,d(**r**A)	Load Word and Zero
lwzu	**r**D,d(**r**A)	Load Word and Zero with Update
lwzux	**r**D,**r**A,**r**B	Load Word and Zero with Update Indexed
lwzx	**r**D,**r**A,**r**B	Load Word and Zero Indexed
mcrf	**crf**D,**crf**S	Move Condition Register Field
mcrfs	**crf**D,**crf**S	Move to Condition Register from FPSCR
mcrxr	**crf**D	Move to Condition Register from XER
mfcr	**r**D	Move from Condition Register
mffs (mffs.)	**fr**D	Move from FPSCR
mfmsr	**r**D	Move from Machine State Register
mfspr	**r**D,SPR	Move from Special Purpose Register
mftb	**r**D, TBR	Move from Time Base
mtcrf	CRM,**r**S	Move to Condition Register Fields
mtfsb0 (mtfsb0.)	**crb**D	Move to FPSCR Bit 0
mtfsb1 (mtfsb1.)	**crb**D	Move to FPSCR Bit 1
mtfsf (mtfsf.)	FM,**fr**B	Move to FPSCR Fields
mtfsfi (mtfsfi.)	**crf**D,IMM	Move to FPSCR Field Immediate
mtmsr	**r**S	Move to Machine State Register
mtspr	SPR,**r**S	Move to Special Purpose Register
mulhw (mulhw.)	**r**D,**r**A,**r**B	Multiply High Word
mulhwu (mulhwu.)	**r**D,**r**A,**r**B	Multiply High Word Unsigned

Table 3-19 Instruction Set Summary (Continued)

Mnemonic	Operand Syntax	Name
mulli	**r**D,**r**A,SIMM	Multiply Low Immediate
mullw (mullw. mullwo mullwo.)	**r**D,**r**A,**r**B	Multiply Low
nand (nand.)	**r**A,**r**S,**r**B	NAND
neg (neg. nego nego.)	**r**D,**r**A	Negate
nor (nor.)	**r**A,**r**S,**r**B	NOR
or (or.)	**r**A,**r**S,**r**B	OR
orc (orc.)	**r**A,**r**S,**r**B	OR with Complement
ori	**r**A,**r**S,UIMM	OR Immediate
oris	**r**A,**r**S,UIMM	OR Immediate Shifted
rfi	—	Return from Interrupt
rlwimi (rlwimi.)	**r**A,**r**S,SH,MB,ME	Rotate Left Word Immediate then Mask Insert
rlwinm (rlwinm.)	**r**A,**r**S,SH,MB,ME	Rotate Left Word Immediate then AND with Mask
rlwnm (rlwnm.)	**r**A,**r**S,**r**B,MB,ME	Rotate Left Word then AND with Mask
sc	—	System Call
slw (slw.)	**r**A,**r**S,**r**B	Shift Left Word
sraw (sraw.)	**r**A,**r**S,**r**B	Shift Right Algebraic Word
srawi (srawi.)	**r**A,**r**S,SH	Shift Right Algebraic Word Immediate
srw (srw.)	**r**A,**r**S,**r**B	Shift Right Word
stb	**r**S,d(**r**A)	Store Byte
stbu	**r**S,d(**r**A)	Store Byte with Update
stbux	**r**S,**r**A,**r**B	Store Byte with Update Indexed
stbx	**r**S,**r**A,**r**B	Store Byte Indexed
stfd	**fr**S,d(**r**A)	Store Floating-Point Double
stfdu	**fr**S,d(**r**A)	Store Floating-Point Double with Update
stfdux	**fr**S,**r**B	Store Floating-Point Double with Update Indexed
stfdx	**fr**S,**r**B	Store Floating-Point Double Indexed
stfiwx	**fr**S,**r**B	Store Floating-Point as Integer Word Indexed
stfs	**fr**S,d(**r**A)	Store Floating-Point Single
stfsu	**fr**S,d(**r**A)	Store Floating-Point Single with Update
stfsux	**fr**S,**r**B	Store Floating-Point Single with Update Indexed
stfsx	**fr**S,**r** B	Store Floating-Point Single Indexed
sth	**r**S,d(**r**A)	Store Half Word
sthbrx	**r**S,**r**A,**r**B	Store Half Word Byte-Reverse Indexed
sthu	**r**S,d(**r**A)	Store Half Word with Update
sthux	**r**S,**r**A,**r**B	Store Half Word with Update Indexed
sthx	**r**S,**r**A,**r**B	Store Half Word Indexed
stmw	**r**S,d(**r**A)	Store Multiple Word
stswi	**r**S,**r**A,NB	Store String Word Immediate
stswx	**r**S,**r**A,**r**B	Store String Word Indexed
stw	**r**S,d(**r**A)	Store Word

Table 3-19 Instruction Set Summary (Continued)

Mnemonic	Operand Syntax	Name
stwbrx	**r**S,**r**A,**r**B	Store Word Byte-Reverse Indexed
stwcx.	**r**S,**r**A,**r**B	Store Word Conditional Indexed
stwu	**r**S,d(**r**A)	Store Word with Update
stwux	**r**S,**r**A,**r**B	Store Word with Update Indexed
stwx	**r**S,**r**A,**r**B	Store Word Indexed
subf (subf. subfo subfo.)	**r**D,**r**A,**r**B	Subtract From
subfc (subfc. subfco subfco.)	**r**D,**r**A,**r**B	Subtract from Carrying
subfe (subfe. subfeo subfeo.)	**r**D,**r**A,**r**B	Subtract from Extended
subfic	**r**D,**r**A,SIMM	Subtract from Immediate Carrying
subfme (subfme. subfmeo subfmeo.)	**r**D,**r**A	Subtract from Minus One Extended
subfze (subfze. subfzeo subfzeo.)	**r**D,**r**A	Subtract from Zero Extended
sync	—	Synchronize
tw	TO,**r**A,**r**B	Trap Word
twi	TO,**r**A,SIMM	Trap Word Immediate
xor (xor.)	**r**A,**r**S,**r**B	XOR
xori	**r**A,**r**S,UIMM	XOR Immediate
xoris	**r**A,**r**S,UIMM	XOR Immediate Shifted

3.10.2 Recommended Simplified Mnemonics

To simplify assembly language coding, a set of alternative mnemonics is provided for some frequently used operations (such as no-op, load immediate, load address, move register, and complement register).

For a complete list of simplified mnemonics, see the *RCPU Reference Manual* (RCPURM/AD). Programs written to be portable across the various assemblers for the PowerPC architecture should not assume the existence of mnemonics not described in that manual.

3.10.3 Calculating Effective Addresses

The effective address (EA) is the 32-bit address computed by the processor when executing a memory access or branch instruction or when fetching the next sequential instruction.

The PowerPC architecture supports two simple memory addressing modes:

- EA = (**r**A|0) + 16-bit offset (including offset = 0) (register indirect with immediate index)
- EA = (**r**A|0) + **r**B (register indirect with index)

These simple addressing modes allow efficient address generation for memory accesses. Calculation of the effective address for aligned transfers occurs in a single clock cycle.

For a memory access instruction, if the sum of the effective address and the operand length exceeds the maximum effective address, the storage operand is considered to wrap around from the maximum effective address to effective address 0.

Effective address computations for both data and instruction accesses use 32-bit unsigned binary arithmetic. A carry from bit 0 is ignored in 32-bit implementations.

3.11 Exception Model

The PowerPC exception mechanism allows the processor to change to supervisor state as a result of external signals, errors, or unusual conditions arising in the execution of instructions. When exceptions occur, information about the state of the processor is saved to certain registers, and the processor begins execution at an address (exception vector) predetermined for each exception. Processing of exceptions occurs in supervisor mode.

Although multiple exception conditions can map to a single exception vector, a more specific condition may be determined by examining a register associated with the exception — for example, the DAE/source instruction service register (DSISR). Additionally, some exception conditions can be explicitly enabled or disabled by software.

3.11.1 Exception Classes

The MPC555 exception classes are shown in **Table 3-20**.

Table 3-20 MPC555 Exception Classes

Class	Exception Type
Asynchronous, unordered	Machine check System reset
Asynchronous, ordered	External interrupt Decrementer
Synchronous (ordered, precise)	Instruction-caused exceptions

3.11.2 Ordered Exceptions

In the MPC555, all exceptions except for reset, debug port non-maskable interrupts, and machine check exceptions are ordered. Ordered exceptions satisfy the following criteria:

- Only one exception is reported at a time. If, for example, a single instruction encounters multiple exception conditions, those conditions are encountered sequentially. After the exception handler handles an exception, instruction execution continues until the next exception condition is encountered.
- When the exception is taken, no program state is lost.

3.11.3 Unordered Exceptions

Unordered exceptions may be reported at any time and are not guaranteed to preserve program state information. The processor can never recover from a reset exception. It can recover from other unordered exceptions in most cases. However, if a debug port non-maskable interrupt or machine check exception occurs during the servicing of a previous exception, the machine state information in SRR0 and SRR1 (and, in some cases, the DAR and DSISR) may not be recoverable; the processor may be in the process of saving or restoring these registers.

To determine whether the machine state is recoverable, the user can read the RI (recoverable exception) bit in SRR1. During exception processing, the RI bit in the MSR is copied to SRR1 and then cleared. The operating system should set the RI bit in the MSR at the end of each exception handler's prologue (after saving the program state) and clear the bit at the start of each exception handler's epilogue (before restoring the program state). Then, if an unordered exception occurs during the servicing of an exception handler, the RI bit in SRR1 will contain the correct value.

3.11.4 Precise Exceptions

In the MPC555, all synchronous (instruction-caused) exceptions are precise. When a precise exception occurs, the processor backs the machine up to the instruction causing the exception. This ensures that the machine is in its correct architecturally-defined state. The following conditions exist at the point a precise exception occurs:

1. Architecturally, no instruction following the faulting instruction in the code stream has begun execution.
2. All instructions preceding the faulting instruction appear to have completed with respect to the executing processor.
3. SRR0 addresses either the instruction causing the exception or the immediately following instruction. Which instruction is addressed can be determined from the exception type and the status bits.
4. Depending on the type of exception, the instruction causing the exception may not have begun execution, may have partially completed, or may have completed execution.

3.11.5 Exception Vector Table

The setting of the exception prefix (IP) bit in the MSR determines how exceptions are vectored. If the bit is cleared, the exception vector table begins at the physical address 0x0000 0000; if IP is set, the exception vector table begins at the physical address 0xFFF0 0000. **Table 3-21** shows the exception vector offset of the first instruction of the exception handler routine for each exception type.

Table 3-21 Exception Vector Offset Table

Vector Offset (Hexadecimal)	Exception Type
00000	Reserved
00100	System reset, NMI interrupt
00200	Machine check
00300	Reserved
00400	Reserved
00500	External interrupt
00600	Alignment
00700	Program
00800	Floating-point unavailable
00900	Decrementer
00A00	Reserved
00B00	Reserved
00C00	System call
00D00	Trace
00E00	Floating-point assist
01000	Implementation-dependent software emulation
01100	Reserved
01200	Reserved
01300	Implementation-dependent instruction protection error
01400	Implementation-dependent data protection error
01500–01BFF	Reserved
01C00	Implementation-dependent data breakpoint
01D00	implementation-dependent instruction breakpoint
01E00	Implementation-dependent maskable external breakpoint
01F00	Implementation-dependent non-maskable external breakpoint

3.12 Instruction Timing

The MPC555 processor is pipelined. Because the processing of an instruction is broken into a series of stages, an instruction does not require the entire resources of the processor.

The instruction pipeline in the MPC555 has four stages:

1. The dispatch stage is implemented using a distributed mechanism. The central dispatch unit broadcasts the instruction to all units. In addition, scoreboard information (regarding data dependencies) is broadcast to each execution unit. Each execution unit decodes the instruction. If the instruction is not implemented, a program exception is taken. If the instruction is legal and no data dependency is found, the instruction is accepted by the appropriate execution unit, and the data found in the destination register is copied to the history buffer. If a data dependency exists, the machine is stalled until the dependency is resolved.
2. In the execute stage, each execution unit that has an executable instruction executes the instruction. (For some instructions, this occurs over multiple cycles.)
3. In the writeback stage, the execution unit writes the result to the destination register and reports to the history buffer that the instruction is completed.
4. In the retirement stage, the history buffer retires instructions in architectural order. An instruction retires from the machine if it completes execution with no exceptions and if all instructions preceding it in the instruction stream have finished execution with no exceptions. As many as six instructions can be retired in one clock.

The history buffer maintains the correct architectural machine state. An exception is taken only when the instruction is ready to be retired from the machine (i.e., after all previously-issued instructions have already been retired from the machine). When an exception is taken, all instructions following the excepting instruction are canceled, i.e., the values of the affected destination registers are restored using the values saved in the history buffer during the dispatch stage.

Figure 3-4 shows basic instruction pipeline timing.

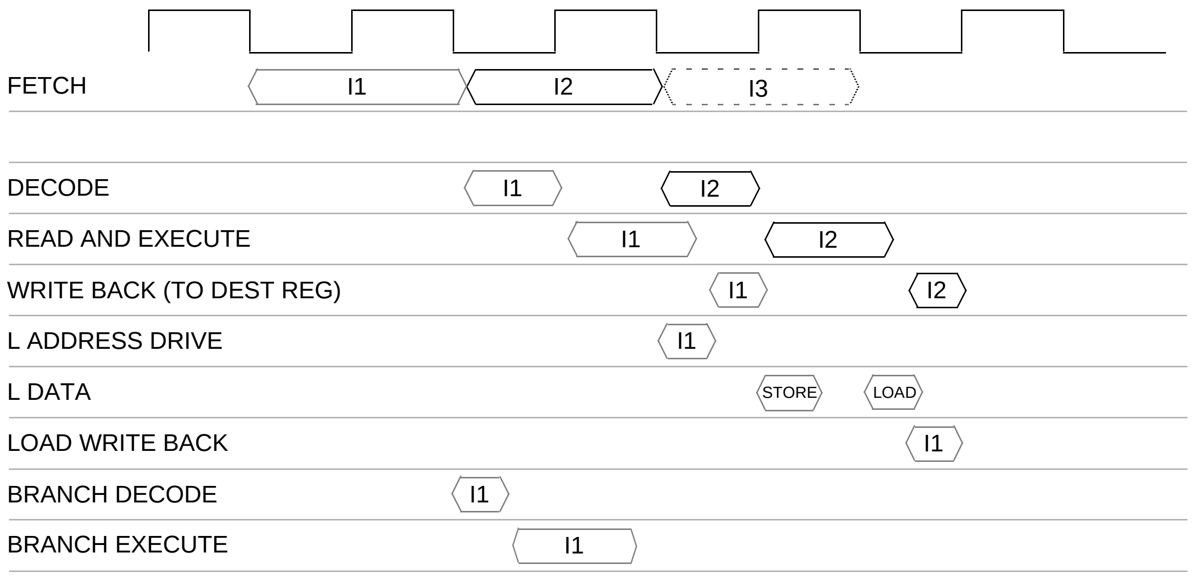


Figure 3-4 Basic Instruction Pipeline

Table 3-22 indicates the latency and blockage for each type of instruction. Latency refers to the interval from the time an instruction begins execution until it produces a result that is available for use by a subsequent instruction. Blockage refers to the interval from the time an instruction begins execution until its execution unit is available for a subsequent instruction. Note that when the blockage equals the latency, it is not possible to issue another instruction to the same unit in the same cycle in which the first instruction is being written back.

Table 3-22 Instruction Latency and Blockage

Instruction Type	Precision	Latency	Blockage
Floating-point multiply-add	Double Single	7 6	7 6
Floating-point add or subtract	Double Single	4 4	4 4
Floating-point multiply	Double Single	5 4	5 4
Floating-point divide	Double Single	17 10	17 10
Integer multiply	—	2	1 or 2[1]
Integer divide	—	2 to 11[1]	2 to 11[1]
Integer load/store	—	See note[1]	See note[1]

NOTES:
1. Refer to **Section 7 Instruction Timing,** in the *RCPU Reference Manual* (RCPURM/AD) for details.

3.13 PowerPC User Instruction Set Architecture (UISA)

3.13.1 Computation Modes

The core of the MPC555 is a 32-bit implementation of the PowerPC architecture. Any reference in the PowerPC Architecture Books (UISA, VEA, OEA) regarding 64-bit implementations are not supported by the core. All registers except the floating-point registers are 32 bits wide.

3.13.2 Reserved Fields

Reserved fields in instructions are described under the specific instruction definition sections. Unless otherwise stated in the specific instruction description, fields marked *"I"*, "*II"* and "*III"* in the instruction are discarded by the core decoding. Thus, this type of invalid form instructions yield results of the defined instructions with the appropriate field zero.

In most cases, the reserved fields in registers are ignored on write and return zeros for them on read on any control register implemented by the core. Exception to this rule are bits 16:23 of the fixed-point exception cause register (XER) and the reserved bits of the machine state register (MSR), which are set by the source value on write and return the value last set for it on read.

3.13.3 Classes of Instructions

Non-optional instructions are implemented by the hardware. Optional instructions are executed by implementation-dependent code and any attempt to execute one of these commands causes the core to take the implementation-dependent software emulation interrupt (offset 0x01000 of the vector table).

Illegal and reserved instruction class instructions are supported by implementation-dependent code and, thus, the core hardware generates the implementation-dependent software emulation interrupt. Invalid and preferred instruction forms treatment by the core is described under the specific processor compliance sections.

3.13.4 Exceptions

Invocation of the system software for any instruction-caused exception in the core is precise, regardless of the type and setting.

3.13.5 The Branch Processor

3.13.6 Instruction Fetching

The core fetches a number of instructions into its internal buffer (the instruction prefetch queue) prior to execution. If a program modifies the instructions it intends to execute, it should call a system library program to ensure that the modifications have been made visible to the instruction fetching mechanism prior to execution of the modified instructions.

3.13.7 Branch Instructions

The core implements all the instructions defined for the branch processor by the **UISA** in the hardware. For performance of various instructions, refer to **Table 3-22** of this manual.

3.13.7.1 Invalid Branch Instruction Forms

Bits marked with *z* in the BO encoding definition are discarded by the core decoding. Thus, these types of invalid form instructions yield result of the defined instructions with the *z* bit zero. If the decrement and test CTR option is specified for the **bcctr** or **bcctrl** instructions, the target address of the branch is the new value of the CTR. Condition is evaluated correctly, including the value of the counter after decrement.

3.13.7.2 Branch Prediction

The core uses the *y* bit to predict path for pre-fetch. Prediction is only done for not-ready branch conditions. No prediction is done for branches to link or count register if the target address is not ready. Refer to *RCPU Reference Manual (*Conditional Branch Control) for more information.

3.13.8 The Fixed-Point Processor

3.13.8.1 Fixed-Point Instructions

The core implements the following instructions:

- Fixed-point arithmetic instructions
- Fixed-point compare instructions
- Fixed-point trap instructions
- Fixed-point logical instructions
- Fixed-point rotate and shift instructions
- Move to/from system register instructions

All instructions are defined for the fixed-point processor in the **UISA** in the hardware. For performance of the various instructions, refer to **Table 3-22**.

— **Move To/From System Register Instructions.** Move to/from invalid special registers in which spr0 = 1 yields invocation of the privilege instruction error interrupt handler if the processor is in problem state. For a list of all implemented special registers, refer to **Table 3-2 Supervisor-Level SPRs**, and **Table 3-3 Development Support SPRs.**

— **Fixed-Point Arithmetic Instructions.** If an attempt is made to perform any of the divisions in the divw[o][.] instruction:
0x80000000 ÷ -1
<anything> ÷ 0
Then, the contents of RT are 0x80000000 and if Rc =1, the contents of bits in CR field 0 are LT = 1, GT = 0, EQ = 0, and SO is set to the correct value. If an attempt is made to perform any of the divisions in the divw[o][.] instruction, <anything> ÷ 0. Then, the contents of RT are 0x80000000 and if Rc = 1, the contents of bits in CR field 0 are LT = 1, GT = 0, EQ = 0, and SO is set to the

correct value. In cmpi, cmp, cmpli, and cmpl instructions, the L-bit is applicable for 64-bit implementations. In 32-bit implementations, if L = 1 the instruction form is invalid. The core ignores this bit and therefore, the behavior when L = 1 is identical to the valid form instruction with L = 0

3.13.9 Floating-Point Processor

3.13.9.1 General

The core implements all floating-point features as defined in the **UISA**, including the non-IEEE working mode. Some features require software assistance. For more information refer to *RCPU Reference Manual* (Floating-point Load Instructions) for more information.

3.13.9.2 Optional instructions

The only optional instruction implemented by MPC555 hardware is Store Floating-Point as Integer Word Indexed (**stfiwx**). An attempt to execute any other optional instruction causes the implementation dependent software emulation interrupt to be taken.

3.13.10 Load/Store Processor

The load/store processor supports all of the 32-bit implementation fixed-point PowerPC load/store instructions in the hardware.

3.13.10.1 Fixed-Point Load With Update and Store With Update Instructions

For load with update and store with update instructions, where RA = 0, the EA is written into R0. For load with update instructions, where RA = RT, RA is boundedly undefined.

3.13.10.2 Fixed-Point Load and Store Multiple Instructions

For these types of instructions, EA must be a multiple of four. If it is not, the system alignment error handler is invoked. For a **lmw** instruction (if RA is in the range of registers to be loaded), the instruction completes normally. RA is then loaded from the memory location as follows:

RA $\leftarrow$ MEM(EA+(RA-RT)*4, 4)

3.13.10.3 Fixed-Point Load String Instructions

Load string instructions behave the same as load multiple instructions, with respect to invalid format in which RA is in the range of registers to be loaded. In case RA is in the range, it is updated from memory.

3.13.10.4 Storage Synchronization Instructions

For these type of instructions, EA must be a multiple of four. If it is not, the system alignment error handler is invoked.

3.13.10.5 Floating-Point Load and Store With Update Instructions

For Load and Store with update instructions, if RT = 0 then the EA is written into R0.

3.13.10.6 Floating-Point Load Single Instructions

In case the operand falls in the range of a single denormalized number the Floating-Point Assist Interrupt handler is invoked.

Refer to *RCPU Reference Manual* (Floating-Point Assist for Denormalized Operands) for complete description of handling denormalized floating-point numbers.

3.13.10.7 Floating-Point Store Single Instructions

In case the operand falls in the range of a single denormalized number, the Floating-Point Assist Interrupt handler is invoked.

In case the operand is ZERO it is converted to the correct signed ZERO in single-precision format.

In case the operand is between the range of single denormalized and double denormalized it is considered a programming error. The hardware will handle this case as if the operand was single denormalized.

In case the operand falls in the range of double denormalized numbers it is considered a programming error. The hardware will handle this case as if the operand was ZERO.

The following check is done on the stored operand in order to determine whether it is a denormalized single-precision operand and invoke the **Floating-Point Assist Interrupt** handler:

$$(FRS_{1:11} \neq 0) \text{ AND } (FRS_{1:11} \leq 896)$$

Refer to *RCPU Reference Manual* (Floating-Point Assist for Denormalized Operands) for complete description of handling denormalized floating-point numbers.

3.13.10.8 Optional Instructions

No optional instructions are supported.

3.13.10.9 Little-Endian Byte Ordering

The load/store unit supports little-endian byte ordering as specified in the **UISA**. In little-endian mode, if an attempt is made to execute an individual scalar unaligned transfer, as well as a multiple or string instruction, an alignment interrupt is taken.

3.14 PowerPC Virtual Environment Architecture (VEA)

3.14.1 Atomic Update Primitives

Both the **lwarx** and **stwcx** instructions are implemented according to the PowerPC architecture requirements. The MPC555 does not provide support for snooping an external bus activity outside the chip. The provision is made to cancel the reservation inside the MPC555 by using the CR_B and KR_B input pins.

3.14.2 Effect of Operand Placement on Performance

The load/store unit hardware supports all of the PowerPC load/store instructions. An optimal performance is obtained for naturally aligned operands. These accesses result in optimal performance (one bus cycle) for up to 4 bytes size and good performance (two bus cycles) for double precision floating-point operands. Unaligned operands are supported in hardware and are broken into a series of aligned transfers. The effect of operand placement on performance is as stated in the **VEA**, except for the case of 8-byte operands. In that case, since the MPC555 uses a 32-bit wide data bus, the performance is good rather than optimal.

3.14.3 Storage Control Instructions

The MPC555 does not implement cache control instructions (**icbi**, **isync**, **dcbt**, **dcbi**, **dcbf**, **dcbz**, **dcbst**, and **dcbtst**) .

3.14.4 Instruction Synchronize (isync) Instruction

The **isync** instruction causes a reflect which waits for all prior instructions to complete and then executes the next sequential instruction. Any instruction after an **isync** will see all effects of prior instructions.

3.14.4.1 Enforce In-Order Execution of I/O (eieio) Instruction

When executing an **eieio** instruction, the load/store unit will wait until all previous accesses have terminated before issuing cycles associated with load/store instructions following the **eieio** instruction.

3.14.5 Timebase

A description of the timebase register may be found in **SECTION 6 SYSTEM CONFIGURATION AND PROTECTION** and in **SECTION 8 CLOCKS AND POWER CONTROL**.

3.15 POWERPC Operating Environment Architecture (OEA)

The MPC555 has an internal memory space that includes memory-mapped control registers and internal memory used by various modules on the chip. This memory is part of the main memory as seen by the core but cannot be accessed by any external system master.

3.15.1 Branch Processor Registers

3.15.1.1 Machine State Register (MSR)

The Floating-Point exception mode encoding in the MPC555 core is as follows:

Table 3-23 Floating-Point Exception Mode Encoding

Mode	FE0	FE1
Ignore exceptions	0	0
Precise	0	1
Precise	1	0
Precise	1	1

The SF bit is reserved set to zero

The IP bit initial state after reset is set as programmed by the reset configuration as specified by the USIU specification.

3.15.1.2 Branch Processors Instructions

The core implements all the instructions defined for the branch processor in the **UISA** in the hardware.

3.15.2 Fixed-Point Processor

3.15.2.1 Special Purpose Registers

- **Unsupported Registers —** The following registers are not supported by the MPC555: SDR, EAR, IBAT0U, IBAT0L, IBAT1U, IBAT1L, IBAT2U, IBAT2L, IBAT3U, IBAT3L, DBAT0U, DBAT0L, DBAT1U, DBAT1L, DBAT2L, DBAT3U, DBAT3L
- **Added Registers —** For a list of added special purpose registers, refer to **Table 3-2**, and **Table 3-3**.

3.15.3 Storage Control Instructions

Storage Control Instructions **mtsr, mtsrin, mfsr, mfsrin, dcbi, tlbie, tlbia,** and **tlb-sync** are not implemented by the MPC555.

3.15.4 Interrupts

The core implements all storage-associated interrupts as precise interrupts. This means that a load/store instruction is not complete until all possible error indications have been sampled from the load/store bus. This also implies that a store, or a non-speculative load instruction is not issued to the load/store bus until all previous instructions have completed. In case of a late error, a store cycle (or a nonspeculative load cycle) can be issued and then aborted.

In each interrupt handler, when registers SRR0 and SRR1 are saved, MSR_{RI} can be set to 1.

The following paragraphs define the types of OEA interrupts The exception table vector defines the offset value by interrupt type. Refer to **Table 3-21**.

3.15.4.1 System Reset Interrupt

A system reset interrupt occurs when the IRQ0 pin is asserted and the following registers are set.

Register Name	Bits	Description
Save/Restore Register 0 (SRR0)		Set to the effective address of the instruction that the processor attempts to execute next if no interrupt conditions are present
Save/Restore Register 1 (SRR1)	1:4	Set to 0
	10:15	Set to 0
	Other	Loaded from bits 16:31 of MSR. In the current implementation, Bit 30 of the SRR1 is never cleared, except by loading a zero value from MSR_{RI}
Machine State Register (MSR)	IP	No change
	ME	No change
	LE	Bit is copied from ILE
	Other	Set to 0

3.15.4.2 Machine Check Interrupt

A machine check interrupt indication is received from the U-bus as a possible response either to the address or data phase. It is usually caused by one of the following conditions:

- The accessed address does not exist
- A data error is detected

As defined in the **OEA**, machine check interrupts are enabled when MSR_{ME} = 1. If MSR_{ME} = 0 and a machine check interrupt indication is received, the processor enters the checkstop state. The behavior of the core in checkstop state is dependent on the working mode as defined in **21.4.1.1 Debug Mode Enable vs. Debug Mode Disable**. When the processor is in debug mode enable, it enters the debug mode instead of the checkstop state. When in debug mode disable, instruction processing is suspended and cannot be restarted without resetting the core.

An indication is sent to the SIU which may generate an automatic reset in this condition. Refer to **SECTION 7 RESET** for more details. If the machine check interrupt is enabled, MSR_{ME} = 1, it is taken. If SRR1 Bit 30 = 1, the interrupt is recoverable and the following registers are set.

Register Name	Bits	Description
Save/Restore Register 0 (SRR0)		Set to the effective address of the instruction that caused the interrupt
Save/Restore Register 1 (SRR1)	1	Set to 1 for instruction fetch-related errors and 0 for load/store-related errors
	2:4	Set to 0
	10:15	Set to 0
	Other	Loaded from bits 16:31 of MSR. In the current implementation, Bit 30 of the SRR1 is never cleared, except by loading a zero value from MSR_{RI}
Machine State Register (MSR)	IP	No change
	ME	No change
	LE	Bit is copied from ILE
	Other	Set to 0

For load/store bus cases, these registers are also set:

Register Name	Bits	Description
Data/Storage Interrupt Status Register (DSISR)	0:14	Set to 0
	15:16	Set to bits 29:30 of the instruction if X-form and to 0b00 if D-form
	17	Set to Bit 25 of the instruction if X-form and to Bit 5 if D-form
	18:21	Set to bits 21:24 of the instruction if X-form and to bits 1:4 if D-form
	22:31	Set to bits 6:15 of the instruction
Data Address Register (DAR)		Set to the effective address of the data access that caused the interrupt

Execution resumes at offset 0x00200 from the base address indicated by MSR_{IP}.

3.15.4.3 Data Storage Interrupt

A data storage interrupt is never generated by the hardware. The software may branch to this location as a result of implementation-specific data storage protection error interrupt.

3.15.4.4 Instruction Storage Interrupt

An instruction storage interrupt is never generated by the hardware. The software may branch to this location as a result of an implementation-specific instruction storage protection error interrupt.

3.15.4.5 Alignment Interrupt

An alignment exception occurs as a result of one of the following conditions:

- The operand of a floating-point load or store is not word aligned.
- The operand of load/store multiple is not word aligned.
- The operand of **lwarx** or **stwcx** is not word aligned.
- The operand of load/store individual scalar instruction is not naturally aligned when MSR_{LE} = 1.
- An attempt to execute multiple/string instruction is made when MSR_{LE} = 1.

3.15.4.6 Floating-Point Enabled Exception Type Program Interrupt

A floating-point enabled exception type program interrupt is generated if ((MSR_{FE0} | MSR_{FE1}) &$FPSCR_{FEX}$) is set as a result of move to FPSCR instruction, move to MSR instruction or the execution of the **rfi** instruction. A floating-point enabled exception type program interrupt is not generated by floating-point arithmetic instructions. Instead if ((MSR_{FE0} | MSR_{FE1}) &$FPSCR_{FEX}$) is set, the floating-point assist interrupt is generated.

3.15.4.7 Illegal Instruction Type Program Interrupt

An illegal instruction type program interrupt is not generated by the core. An implementation dependent software emulation interrupt is generated instead.

3.15.4.8 Privileged Instruction Type Program interrupt

A privileged instruction type program interrupt is generated for an on-core valid SPR field or any SPR encoded as an external to the core special register if SPR_0 = 1 and MSR_{PR} = 1, as well as an attempt to execute privileged instruction when MSR_{PR} = 1.

3.15.4.9 Floating-Point Unavailable Interrupt

The floating-point unavailable interrupt is generated by the MPC555 core as defined in the **OEA**.

3.15.4.10 Trace Interrupt

A trace interrupt occurs if MSR_{SE} = 1 and any instruction except **rfi** is successfully completed or MSR_{BE} = 1 and a branch is completed. Notice that the trace interrupt does not occur after an instruction that caused an interrupt (for instance, **sc**). A monitor/debugger software must change the vectors of other possible interrupt addresses to single-step such instructions. If this is unacceptable, other debug features can be used. Refer to **SECTION 21 DEVELOPMENT SUPPORT** for more information. The following registers are set:

Register Name	Bits	Description
Save/Restore Register 0 (SRR0)		Set to the effective address of the instruction following the executed instruction
Save/Restore Register 1 (SRR1)	1:4	Set to 0
	10:15	Set to 0
	Other	Loaded from bits 16:31 of MSR. In the current implementation, Bit 30 of the SRR1 is never cleared, except by loading a zero value from MSR_{RI}
Machine State Register (MSR)	IP	No change
	ME	No change
	LE	Bit is copied from ILE
	Other	Set to 0

Execution resumes at offset 0x00D00 from the base address indicated by MSR_{IP}.

3.15.4.11 Floating-Point Assist Interrupt

A floating-point assist interrupt occurs in the following cases:

- When a floating-point exception condition is detected, the corresponding floating-point enable bit in the FPSCR (floating-point status and control register) is set (exception enabled) and $((MSR_{FE0} \mid MSR_{FE1}) = 1)$. Note that when $((MSR_{FE0} \mid MSR_{FE1})$ and $FPSCR_{FEX})$ is set as a result of move to FPSCR, move to MSR or **rfi**, the floating-point assist interrupt handler is not invoked.
- When an intermediate result is detected and the floating-point underflow exception is disabled ($FPSCR_{UE} = 0$)
- In some cases when at least one of the source operands is denormalized.

The following registers are set:

Register Name	Bits	Description
Save/Restore Register 0 (SRR0)		Set to the effective address of the instruction that caused the interrupt
Save/Restore Register 1 (SRR1)	1:4	Set to 0
	10:15	Set to 0
	Other	Loaded from bits 16:31 of MSR[1]
Machine State Register (MSR)	IP	No change
	ME	No change
	LE	Bit is copied from ILE
	Other	Set to 0

NOTES:

1. In the current implementation bit 30 of the SRR1 is never cleared other then by loading zero value from MSR RI.

Execution resumes at offset 0x00E00 from the base address indicated by $MSR_{IP.}$

3.15.4.12 Implementation-Dependent Software Emulation Interrupt

An implementation-dependent software emulation interrupt occurs in the following instances:

- When executing any non-implemented instruction. This includes all illegal and unimplemented optional instructions and all floating-point instructions.
- When executing a **mtspr** or **mfspr** that specifies on-core non-implemented register, regardless of SPR_0.
- When executing a **mtspr** or **mfspr** that specifies off-core non-implemented register and $SPR_0 = 0$ or $MSR_{PR} = 0$ (no program interrupt condition).
- Program interrupt is generated if ((MSR_{FE0} | MSR_{FE1}) and $FPSCR_{FEX}$) is set as a result of move to FPSCR instruction, move to MSR instruction, or the execution of the **rfi** instruction.
- Floating-point enabled exception type program interrupt is not generated by floating-point arithmetic instructions, instead if ((MSR_{FE0} | MSR_{FE1}) &$FPSCR_{FEX}$) is set, the floating-point assist interrupt is generated.

In addition, the following registers are set:

Register Name	Bits	Description
Save/Restore Register 0 (SRR0)		Set to the effective address of the instruction that caused the interrupt
Save/Restore Register 1 (SRR1)	1:4	Set to 0
	10:15	Set to 0
	Other	Loaded from bits 16:31 of MSR. In the current implementation, Bit 30 of the SRR1 is never cleared, except by loading a zero value from MSR_{RI}
Machine State Register (MSR)	IP	No change
	ME	No change
	LE	Bit is copied from ILE
	Other	Set to 0

Execution resumes at offset 0x01000 from the base address indicated by MSR_{IP}.

3.15.4.13 Implementation-Specific Instruction Storage Protection Error Interrupt

The implementation-specific instruction storage protection error interrupt occurs in the following cases:

- The fetch access violates storage protection.
- The fetch access is to guarded storage and MSR_{IR} = 1.

The following registers are set:

Register Name	Bits	Description
Save/Restore Register 0 (SRR0)		Set to the effective address of the instruction that caused the interrupt
Save/Restore Register 1 (SRR1)	1	Set to 0
	2	Set to 0
	3	Set to 1 if the fetch access was to a guarded storage when MSR_{IR} = 1, otherwise set to 0
	4	Set to 1 if the storage access is not permitted by the protection mechanism; otherwise set to 0
	10	Set to 0
	11:15	Set to 0
	Other	Loaded from bits 16:31 of MSR. In the current implementation, Bit 30 of the SRR1 is never cleared, except by loading a zero value from MSR_{RI}
Machine State Register (MSR)	IP	No change
	ME	No change
	LE	Bit is copied from ILE
	Other	Set to 0

Execution resumes at offset 0x01300 from the base address indicated by MSR_{IP}.

3.15.4.14 Implementation-Specific Data Storage Protection Error Interrupt

The implementation-specific data storage protection error interrupt occurs in the following case:

- The access violates the storage protection.

The following registers are set:

Register Name	Bits	Description
Save/Restore Register 0 (SRR0)		Set to the effective address of the instruction that caused the interrupt
Save/Restore Register 1 (SRR1)	1:4	Set to 0
	10:15	Set to 0
	Other	Loaded from bits 16:31 of MSR. In the current implementation, Bit 30 of the SRR1 is never cleared, except by loading a zero value from MSR_{RI}
Machine State Register (MSR)	IP	No change
	ME	No change
	LE	Bit is copied from ILE
	Other	Set to 0
Data/Storage Interrupt Status Register (DSISR)	0	Set to 0
	1	Set to 0
	2:3	Set to 0
	4	Set to 1 if the storage access is not permitted by the protection mechanism. Otherwise set to 0
	5	Set to 0
	6	Set to 1 for a store operation and to 0 for a load operation
	7:31	Set to 0
Data Address Register (DAR)		Set to the effective address of the data access that caused the interrupt

Execution resumes at offset 0x01400 from the base address indicated by MSR_{IP}.

3.15.4.15 Implementation-Specific Debug Interrupts

Implementation-specific debug interrupts occur in the following cases:

- When there is an internal breakpoint match (for more details, refer to **SECTION 21 DEVELOPMENT SUPPORT**.
- When a peripheral breakpoint request is asserted to the MPC555 core.
- When the development port request is asserted to the MPC555 core. Refer to **SECTION 21 DEVELOPMENT SUPPORT** for details on how to generate the development port-interrupt request.

The following registers are set:

Register Name	Bits	Description
Save/Restore Register 0 (SRR0)		For I-breakpoints, set to the effective address of the instruction that caused the interrupt. For L-breakpoint, set to the effective address of the instruction following the instruction that caused the interrupt. For development port maskable request or a peripheral breakpoint, set to the effective address of the instruction that the processor would have executed next if no interrupt conditions were present. If the development port request is asserted at reset, the value of SRR0 is undefined.
Save/Restore Register 1 (SRR1)	1:4	Set to 0
	10:15	Set to 0
	Other	Loaded from bits 16:31 of MSR. In the current implementation, Bit 30 of the SRR1 is never cleared, except by loading a zero value from MSR_{RI}. If the development port request is asserted at reset, the value of SRR1 is undefined.
Machine State Register (MSR)	IP	No change
	ME	No change
	LE	Bit is copied from ILE
	Other	Set to 0

For L-bus breakpoint instances, these registers are set to:

Register Name	Bits	Description
BAR		Set to the effective address of the data access as computed by the instruction that caused the interrupt
DAR and DSISR		Do not change

Execution resumes at offset from the base address indicated by MSR_{IP} as follows:

- 0x01D00 – For instruction breakpoint match
- 0x01C00 – For data breakpoint match
- 0x01E00 – For development port maskable request or a peripheral breakpoint
- 0x01F00 – For development port non-maskable request

3.15.4.16 Partially Executed Instructions

In general, the architecture permits instructions to be partially executed when an alignment or data storage interrupt occurs. In the core, instructions are not executed at all if an alignment interrupt condition is detected and data storage interrupt is never generated by the hardware. In the MPC555, the instruction can be partially executed only in the case of the load/store instructions that cause multiple access to the memory subsystem. These instructions are:

- Multiple/string instructions
- Unaligned load/store instructions

In the last case, the store instruction can be partially completed if one of the accesses (except the first one) causes the data storage protection error. The implementation-specific data storage protection interrupt is taken in this case. For the update forms, the update register (RA) is not altered.

3.15.5 Timer Facilities

Descriptions of the timebase and decrementer registers can be found in **SECTION 6 SYSTEM CONFIGURATION AND PROTECTION** and in **SECTION 8 CLOCKS AND POWER CONTROL**.

3.15.6 Optional Facilities and Instructions

Any other OEA optional facilities and instructions (except those that are discussed here) are not implemented by the MPC555 hardware. Attempting to execute any of these instructions causes an implementation dependent software emulation interrupt to be taken.