PowerPC™ 601

RISC Microprocessor User's Manual

IBM PowerPC MOTOROLA

CONTENTS

CONTENTS

Chapter 2
Registers and Data Types

CONTENTS

CONTENTS

CONTENTS

CONTENTS

Chapter 4
Cache and Memory Unit Operation

CONTENTS

Chapter 5
Exceptions

CONTENTS

Chapter 6
Memory Management Unit

CONTENTS

CONTENTS

Chapter 7
Instruction Timing

CONTENTS

CONTENTS

CONTENTS

Chapter 8
Signal Descriptions

CONTENTS

CONTENTS

Chapter 9
System Interface Operation

CONTENTS

Chapter 10
Instruction Set

Appendix A
Instruction Set Listings

CONTENTS

Appendix B
POWER Architecture Cross Reference

Appendix C
PowerPC Instructions Not Implemented

CONTENTS

CONTENTS

Appendix H
Implementation Summary for Programmers

Appendix I
Instruction Timing Examples

CONTENTS

Appendix A
Instruction Set Listings

This appendix lists the instruction set implemented in the PowerPC 601 microprocessor and the additional PowerPC instructions not implemented in the 601 sorted by mnemonic, opcode, and form, and grouped by functional categories.

A.1 Complete Instruction List Sorted by Mnemonic

Table A-1 lists the instructions implemented in the 601 plus those defined in the PowerPC architecture that are not implemented in the 601 in alphabetical order by mnemonic.

Table A-1. Complete Instruction List Sorted by Mnemonic

Key:

Reserved bits

Instruction not implemented in the 601

Name	0	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	31
abs*x* [3]	31		D				A					0	0	0	0	0	OE				360						Rc
add*x*	31		D				A					B					OE				266						Rc
addc*x*	31		D				A					B					OE				10						Rc
adde*x*	31		D				A					B					OE				138						Rc
addi	14		D				A					SIMM															
addic	12		D				A					SIMM															
addic.	13		D				A					SIMM															
addis	15		D				A					SIMM															
addme*x*	31		D				A					0	0	0	0	0	OE				234						Rc
addze*x*	31		D				A					0	0	0	0	0	OE				202						Rc
and*x*	31		S				A					B					28										Rc
andc*x*	31		S				A					B					60										Rc
andi.	28		S				A					UIMM															
andis.	29		S				A					UIMM															

Name	0	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	31
b*x*	18	LI																								AA	LK
bc*x*	16	BO					BI					BD														AA	LK
bcctr*x*	19	BO					BI					0 0 0 0 0				528											LK
bclr*x*	19	BO					BI					0 0 0 0 0				16											LK
clcs [3]	31	D					A					0 0 0 0 0				531											Rc
cmp	31	crfD			0	L	A					B				0											0
cmpi	11	crfD			0	L	A					SIMM															
cmpl	31	crfD			0	L	A					B				32											0
cmpli	10	crfD			0	L	A					UIMM															
cntlzd [5]	31	S					A					0 0 0 0 0				58											Rc
cntlzw*x*	31	S					A					0 0 0 0 0				26											Rc
crand	19	crbD					crbA					crbB				257											0
crandc	19	crbD					crbA					crbB				129											0
creqv	19	crbD					crbA					crbB				289											0
crnand	19	crbD					crbA					crbB				225											0
crnor	19	crbD					crbA					crbB				33											0
cror	19	crbD					crbA					crbB				449											0
crorc	19	crbD					crbA					crbB				417											0
crxor	19	crbD					crbA					crbB				193											0
dcbf	31	0 0 0 0 0					A					B				86											0
dcbi [1]	31	0 0 0 0 0					A					B				470											0
dcbst	31	0 0 0 0 0					A					B				54											0
dcbt	31	0 0 0 0 0					A					B				278											0
dcbtst	31	0 0 0 0 0					A					B				246											0
dcbz	31	0 0 0 0 0					A					B				1014											0
div*x* [3]	31	D					A					B				OE	331										Rc
divd [5]	31	D					A					B				OE	489										Rc
divdu [5]	31	D					A					B				OE	457										Rc
divs*x* [3]	31	D					A					B				OE	363										Rc
divw*x*	31	D					A					B				OE	491										Rc
divwu*x*	31	D					A					B				OE	459										Rc
doz*x* [3]	31	D					A					B				OE	264										Rc
dozi [3]	9	D					A					SIMM															

Name	0	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	31
eciwx	31	D					A					B					310										0
ecowx	31	S					A					B					438										0
eieio	31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	854										0
eqv*x*	31	S					A					B					284										Rc
extsb*x*	31	S					A					0	0	0	0	0	954										Rc
extsh*x*	31	S					A					0	0	0	0	0	922										Rc
extsw [5]	31	S					A					0	0	0	0	0	986										Rc
fabs*x*	63	D					0	0	0	0	0	B					264										Rc
fadd*x*	63	D					A					B					0	0	0	0	0	21					Rc
fadds*x*	59	D					A					B					0	0	0	0	0	21					Rc
fcfid [5]	63	D					0	0	0	0	0	B					846										Rc
fcmpo	63	crfD			0	0	A					B					32										0
fcmpu	63	crfD			0	0	A					B					0										0
fctid [5]	63	D					0	0	0	0	0	B					814										Rc
fctidz [5]	63	D					0	0	0	0	0	B					815										Rc
fctiw*x*	63	D					0	0	0	0	0	B					14										Rc
fctiwz*x*	63	D					0	0	0	0	0	B					15										Rc
fdiv*x*	63	D					A					B					0	0	0	0	0	18					Rc
fdivs*x*	59	D					A					B					0	0	0	0	0	18					Rc
fmadd*x*	63	D					A					B					C					29					Rc
fmadds*x*	59	D					A					B					C					29					Rc
fmr*x*	63	D					0	0	0	0	0	B					72										Rc
fmsub*x*	63	D					A					B					C					28					Rc
fmsubs*x*	59	D					A					B					C					28					Rc
fmul*x*	63	D					A					0	0	0	0	0	C					25					Rc
fmuls*x*	59	D					A					0	0	0	0	0	C					25					Rc
fnabs*x*	63	D					0	0	0	0	0	B					136										Rc
fneg*x*	63	D					0	0	0	0	0	B					40										Rc
fnmadd*x*	63	D					A					B					C					31					Rc
fnmadds*x*	59	D					A					B					C					31					Rc
fnmsub*x*	63	D					A					B					C					30					Rc
fnmsubs*x*	59	D					A					B					C					30					Rc
fres*x*	59	D					0	0	0	0	0	B					0	0	0	0	0	24					Rc

Name	0	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	31
frsp*x*	63	D					0	0	0	0	0	B					12										Rc
frsqrte	63	D					0	0	0	0	0	B					0	0	0	0	0	26					Rc
fsel*x*	63	D					A					B					frC					23					Rc
fsqrt*x*	63	D					0	0	0	0	0	B					0	0	0	0	0	22					Rc
fsqrts	59	D					0	0	0	0	0	B					0	0	0	0	0	22					Rc
fsub*x*	63	D					A					B					0	0	0	0	0	20					Rc
fsubs*x*	59	D					A					B					0	0	0	0	0	20					Rc
icbi	31	0	0	0	0	0	A					B					982										0
isync	19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	150										0
lbz	34	D					A					d															
lbzu	35	D					A					d															
lbzux	31	D					A					B					119										0
lbzx	31	D					A					B					87										0
ld [5]	58	D					A					ds														0	
ldarx [5]	31	D					A					B					84										0
ldu [5]	58	D					A					ds														1	
ldux [5]	31	D					A					B					53										0
ldx [5]	31	D					A					B					21										0
lfd	50	D					A					d															
lfdu	51	D					A					d															
lfdux	31	D					A					B					631										0
lfdx	31	D					A					B					599										0
lfs	48	D					A					d															
lfsu	49	D					A					d															
lfsux	31	D					A					B					567										0
lfsx	31	D					A					B					535										0
lha	42	D					A					d															
lhau	43	D					A					d															
lhaux	31	D					A					B					375										0
lhax	31	D					A					B					343										0
lhbrx	31	D					A					B					790										0
lhz	40	D					A					d															
lhzu	41	D					A					d															

Name	0	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	31
lhzux	31	D					A					B					311										0
lhzx	31	D					A					B					279										0
lmw [4]	46	D					A					d															
lscbx*x* [3, 4]	31	D					A					B					277										Rc
lswi [4]	31	D					A					NB					597										0
lswx [4]	31	D					A					B					533										0
lwa [5]	58	D					A					ds														2	
lwarx	31	D					A					B					20										0
lwaux [5]	31	D					A					B					373										0
lwax [5]	31	D					A					B					341										0
lwbrx	31	D					A					B					534										0
lwz	32	D					A					d															
lwzu	33	D					A					d															
lwzux	31	D					A					B					55										0
lwzx	31	D					A					B					23										0
maskg*x* [3]	31	S					A					B					29										Rc
maskir*x* [3]	31	S					A					B					541										Rc
mcrf	19	crfD			0	0	crfS				0	0	0	0	0	0	0										0
mcrfs	63	crfD			0	0	crfS				0	0	0	0	0	0	64										0
mcrxr	31	crfS			0	0	0	0	0	0	0	0	0	0	0	0	512										0
mfcr	31	D					0	0	0	0	0	0	0	0	0	0	19										0
mffs*x*	63	D					0	0	0	0	0	0	0	0	0	0	583										Rc
mfmsr [1]	31	D					0	0	0	0	0	0	0	0	0	0	83										0
mfspr [2]	31	D					SPR										339										0
mfsr [1]	31	D					0	SR					0	0	0	0	0	595									0
mfsrin [1]	31	D					0	0	0	0	0	B					659										0
mftb	31	D					TBR										371										0
mtcrf	31	S					0	CRM								0	144										0
mtfsb0*x*	63	crbD					0	0	0	0	0	0	0	0	0	0	70										Rc
mtfsb1*x*	63	crbD					0	0	0	0	0	0	0	0	0	0	38										Rc
mtfsf*x*	63	0	FM								0	B					711										Rc
mtfsfi*x*	63	crbD			0	0	0	0	0	0	0	IMM				0	134										Rc
mtmsr [1]	31	S					0	0	0	0	0	0	0	0	0	0	146										0

Name	0	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	31
mtspr [2]	31	D					SPR										467										0
mtsr [1]	31	S					0	SR				0 0 0 0 0					210										0
mtsrin [1]	31	S					0 0 0 0 0					B					242										0
mul*x* [3]	31	D					A					B					OE	107									Rc
mulhd [5]	31	D					A					B					0	73									Rc
mulhdu [5]	31	D					A					B					0	9									Rc
mulhw*x*	31	D					A					B					0	75									Rc
mulhwu*x*	31	D					A					B					0	11									Rc
mulld [5]	31	D					A					B					OE	233									Rc
mulli	7	D					A					SIMM															
mullw*x*	31	D					A					B					OE	235									Rc
nabs*x* [3]	31	D					A					0 0 0 0 0					OE	488									Rc
nand*x*	31	S					A					B					476										Rc
neg*x*	31	D					A					0 0 0 0 0					OE	104									Rc
nor*x*	31	S					A					B					124										Rc
or*x*	31	S					A					B					444										Rc
orc*x*	31	S					A					B					412										Rc
ori	24	S					A					UIMM															
oris	25	S					A					UIMM															
rfi [1]	19	0 0 0 0 0					0 0 0 0 0					0 0 0 0 0					50										0
rldcl [5]	30	S					A					B					MB					8					Rc
rldcr [5]	30	S					A					B					ME					9					Rc
rldic [5]	30	S					A					SH					MB					2			SH		Rc
rldicl [5]	30	S					A					SH					MB					0			SH		Rc
rldicr [5]	30	S					A					SH					ME					1			SH		Rc
rldimi [5]	30	S					A					SH					MB					3			SH		Rc
rlmi*x* [3]	22	S					A					SH					MB					ME					Rc
rlwimi*x*	20	S					A					SH					MB					ME					Rc
rlwinm*x*	21	S					A					SH					MB					ME					Rc
rlwnm*x*	23	S					A					B					MB					ME					Rc
rrib*x* [3]	31	S					A					B					537										Rc
sc	17	0 0 0 0 0					0 0 0 0 0					0 0 0 0 0 0 0 0 0 0 0 0 0 0 0													1	0
slbia [1, 5]	31	0 0 0 0 0					0 0 0 0 0					0 0 0 0 0					498										0

Name	0	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	31
slbie [1, 5]	31	0	0	0	0	0	0	0	0	0	0	B					434										0
sld [5]	31	S					A					B					27										Rc
sle*x* [3]	31	S					A					B					153										Rc
sleq*x* [3]	31	S					A					B					217										Rc
sliq*x* [3]	31	S					A					SH					184										Rc
slliq*x* [3]	31	S					A					SH					248										Rc
sllq*x* [3]	31	S					A					B					216										Rc
slq*x* [3]	31	S					A					B					152										Rc
slw*x*	31	S					A					B					24										Rc
srad [5]	31	S					A					B					794										Rc
sradi [5]	31	S					A					SH					413								SH		Rc
sraiq*x* [3]	31	S					A					SH					952										Rc
sraq*x* [3]	31	S					A					B					920										Rc
sraw*x*	31	S					A					B					792										Rc
srawi*x*	31	S					A					SH					824										Rc
srd [5]	31	S					A					B					539										Rc
sre*x* [3]	31	S					A					B					665										Rc
srea*x* [3]	31	S					A					B					921										Rc
sreq*x* [3]	31	S					A					B					729										Rc
sriq*x* [3]	31	S					A					SH					696										Rc
srliq*x* [3]	31	S					A					SH					760										Rc
srlq*x* [3]	31	S					A					B					728										Rc
srq*x* [3]	31	S					A					B					664										Rc
srw*x*	31	S					A					B					536										Rc
stb	38	S					A					d															
stbu	39	S					A					d															
stbux	31	S					A					B					247										0
stbx	31	S					A					B					215										0
std [5]	62	S					A					ds														0	0
stdcx. [5]	31	S					A					B					214										1
stdu [5]	62	S					A					ds														1	
stdux [5]	31	S					A					B					181										0
stdx [5]	31	S					A					B					149										0

Name	0	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	31
stfd	54	frS					A					d															
stfdu	55	frS					A					d															
stfdux	31	frS					A					B					759										0
stfdx	31	frS					A					B					727										0
stfiwx	31	frS					A					B					983										0
stfs	52	frS					A					d															
stfsu	53	frS					A					d															
stfsux	31	frS					A					B					695										0
stfsx	31	frS					A					B					663										0
sth	44	S					A					d															
sthbrx	31	S					A					B					918										0
sthu	45	S					A					d															
sthux	31	S					A					B					439										0
sthx	31	S					A					B					407										0
stmw [4]	47	S					A					d															
stswi [4]	31	S					A					NB					725										0
stswx [4]	31	S					A					B					661										0
stw	36	S					A					d															
stwbrx	31	S					A					B					662										0
stwcx.	31	S					A					B					150										1
stwu	37	S					A					d															
stwux	31	S					A					B					183										0
stwx	31	S					A					B					151										0
subf*x*	31	D					A					B					OE	40									Rc
subfc*x*	31	D					A					B					OE	8									Rc
subfe*x*	31	D					A					B					OE	136									Rc
subfic	08	D					A					SIMM															
subfme*x*	31	D					A					0 0 0 0 0					OE	232									Rc
subfze*x*	31	D					A					0 0 0 0 0					OE	200									Rc
sync	31	0 0 0 0 0					0 0 0 0 0					0 0 0 0 0						598									0
td [5]	31	TO					A					B						68									0
tdi [5]	02	TO					A					SIMM															
tlbia [1]	31	0 0 0 0 0					0 0 0 0 0					0 0 0 0 0						370									0

Name	0	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	31
tlbie [1]	31	0	0	0	0	0	0	0	0	0	0	B					306										0
tlbsync [1]	31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	566										0
tw	31	TO					A					B					4										0
twi	03	TO					A					SIMM															
xor*x*	31	S					A					B					316										Rc
xori	26	S					A					UIMM															
xoris	27	S					A					UIMM															

[1] Supervisor-level instruction.
[2] Supervisor- and user-level instruction.
[3] Non-PowerPC instruction implemented by 601 for POWER architecture compatibliity.
[4] Load or store string or multiple word instruction.
[5] 64-bit instruction.

A.2 Complete Instruction List Sorted by Opcode

Table A-2 lists the instructions implemented in the 601 plus those defined in the PowerPC architecture that are not implemented in the 601 in numerical order by opcode.

Table A-2. Complete Instruction List Sorted by Opcode

Key:

Reserved bits

Instruction not implemented in the 601

Name	0	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	31
tdi [5]	0 0 0 0 1 0	TO						A					SIMM															
twi	0 0 0 0 1 1	TO						A					SIMM															
mulli	0 0 0 1 1 1	D						A					SIMM															
subfic	0 0 1 0 0 0	D						A					SIMM															
dozi [3]	0 0 1 0 0 1	D						A					SIMM															
cmpli	0 0 1 0 1 0	crfD			0		L	A					UIMM															
cmpi	0 0 1 0 1 1	crfD			0		L	A					SIMM															
addic	0 0 1 1 0 0	D						A					SIMM															
addic.	0 0 1 1 0 1	D						A					SIMM															
addi	0 0 1 1 1 0	D						A					SIMM															
addis	0 0 1 1 1 1	D						A					SIMM															
bc*x*	0 1 0 0 0 0	BO						BI					BD														AA	LK
sc	0 1 0 0 0 1	0 0 0 0 0						0 0 0 0 0					0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0														1	0
b*x*	0 1 0 0 1 0	LI																									AA	LK
mcrf	0 1 0 0 1 1	crfD				0 0		crfS				0 0	0 0 0 0 0					0 0 0	0 0 0 0			0 0 0					0
bclr*x*	0 1 0 0 1 1	BO						BI					0 0 0 0 0					0 0 0	0 0 1 0			0 0 0					LK
crnor	0 1 0 0 1 1	crbD						crbA					crbB					0 0 0	0 1 0 0			0 0 1					0
rfi	0 1 0 0 1 1	0 0 0 0 0						0 0 0 0 0					0 0 0 0 0					0 0 0	0 1 1 0			0 1 0					0
crandc	0 1 0 0 1 1	crbD						crbA					crbB					0 0 1	0 0 0 0			0 0 1					0
isync	0 1 0 0 1 1	0 0 0 0 0						0 0 0 0 0					0 0 0 0 0					0 0 1	0 0 1 0			1 1 0					0
crxor	0 1 0 0 1 1	crbD						crbA					crbB					0 0 1	1 0 0 0			0 0 1					0
crnand	0 1 0 0 1 1	crbD						crbA					crbB					0 0 1	1 1 0 0			0 0 1					0
crand	0 1 0 0 1 1	crbD						crbA					crbB					0 1 0	0 0 0 0			0 0 1					0
creqv	0 1 0 0 1 1	crbD						crbA					crbB					0 1 0	0 1 0 0			0 0 1					0
crorc	0 1 0 0 1 1	crbD						crbA					crbB					0 1 1	0 1 0 0			0 0 1					0
cror	0 1 0 0 1 1	crbD						crbA					crbB					0 1 1	1 0 0 0			0 0 1					0
bcctr*x*	0 1 0 0 1 1	BO						BI					0 0 0 0 0					1 0 0	0 0 1 0			0 0 0					LK

Name	0	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	31
rlwimi*x*	0 1 0 1 0 0	S						A					SH					MB					ME				Rc
rlwinm*x*	0 1 0 1 0 1	S						A					SH					MB					ME				Rc
rlmi*x* [3]	0 1 0 1 1 0	S						A					SH					MB					ME				Rc
rlwnm*x*	0 1 0 1 1 1	S						A					B					MB					ME				Rc
ori	0 1 1 0 0 0	S						A					UIMM															
oris	0 1 1 0 0 1	S						A					UIMM															
xori	0 1 1 0 1 0	S						A					UIMM															
xoris	0 1 1 0 1 1	S						A					UIMM															
andi.	0 1 1 1 0 0	S						A					UIMM															
andis.	0 1 1 1 0 1	S						A					UIMM															
rldicl *x* [5]	0 1 1 1 1 0	S						A					SH					MB					0 0 0			SH	Rc
rldicr*x* [5]	0 1 1 1 1 0	S						A					SH					ME					0 0 1			SH	Rc
rldic*x* [5]	0 1 1 1 1 0	S						A					SH					MB					0 1 0			SH	Rc
rldimi *x* [5]	0 1 1 1 1 0	S						A					SH					MB					0 1 1			SH	Rc
rldcl*x* [5]	0 1 1 1 1 0	S						A					B					MB					1 0 0 0				Rc
rldcr *x* [5]	0 1 1 1 1 0	S						A					B					ME					1 0 0 1				Rc
cmp	0 1 1 1 1 1	crfD			0		L	A					B					0 0 0		0 0 0 0			0 0 0				0
tw	0 1 1 1 1 1	TO						A					B					0 0 0		0 0 0 0			1 0 0				0
subfc*x*	0 1 1 1 1 1	D						A					B					OE	0 0		0 0 0 1		0 0 0				Rc
mulhdu*x* [5]	0 1 1 1 1 1	D						A					B					0	0 0		0 0 0 1		0 0 1				Rc
addc*x*	0 1 1 1 1 1	D						A					B					OE	0 0		0 0 0 1		0 1 0				Rc
mulhwu*x*	0 1 1 1 1 1	D						A					B					0	0 0		0 0 0 1		0 1 1				Rc
mfcr	0 1 1 1 1 1	D						0 0 0 0 0					0 0 0 0 0					0 0 0		0 0 1 0			0 1 1				0
lwarx	0 1 1 1 1 1	D						A					B					0 0 0		0 0 1 0			1 0 0				0
ldx [5]	0 1 1 1 1 1	D						A					B					0 0 0		0 0 1 0			1 0 1				0
lwzx	0 1 1 1 1 1	D						A					B					0 0 0		0 0 1 0			1 1 1				0
slw*x*	0 1 1 1 1 1	S						A					B					0 0 0		0 0 1 1			0 0 0				Rc
cntlzw*x*	0 1 1 1 1 1	S						A					0 0 0 0 0					0 0 0		0 0 1 1			0 1 0				Rc
sld*x* [5]	0 1 1 1 1 1	S						A					B					0 0 0		0 0 1 1			0 1 1				Rc
and*x*	0 1 1 1 1 1	S						A					B					0 0 0		0 0 1 1			1 0 0				Rc
maskg*x* [3]	0 1 1 1 1 1	S						A					B					0 0 0		0 0 1 1			1 0 1				Rc
cmpl	0 1 1 1 1 1	crfD			0		L	A					B					0 0 0		0 1 0 0			0 0 0				0
subf*x*	0 1 1 1 1 1	D						A					B					OE	0 0		0 1 0 1		0 0 0				Rc

Name	0	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	31
ldux [5]	0 1 1 1 1 1		D					A					B					0 0 0			0 1 1 0				1 0 1		0
dcbst	0 1 1 1 1 1		0 0 0 0 0					A					B					0 0 0			0 1 1 0				1 1 0		0
lwzux	0 1 1 1 1 1		D					A					B					0 0 0			0 1 1 0				1 1 1		0
cntlzd*x* [5]	0 1 1 1 1 1		S					A					0 0 0 0 0					0 0 0			0 1 1 1				0 1 0		Rc
andc*x*	0 1 1 1 1 1		S					A					B					0 0 0			0 1 1 1				1 0 0		Rc
td [5]	0 1 1 1 1 1		TO					A					B					0 0 0 1 0 0 0 1 0 0										0
mulhd*x* [5]	0 1 1 1 1 1		D					A					B					0	0 0			1 0 0 1				0 0 1		Rc
mulhw*x*	0 1 1 1 1 1		D					A					B					0	0 0			1 0 0 1				0 1 1		Rc
mfmsr	0 1 1 1 1 1		D					0 0 0 0 0					0 0 0 0 0					0 0 0			1 0 1 0				0 1 1		0
ldarx [5]	0 1 1 1 1 1		D					A					B					0 0 0			1 0 1 0				1 0 0		0
dcbf	0 1 1 1 1 1		0 0 0 0 0					A					B					0 0 0			1 0 1 0				1 1 0		0
lbzx	0 1 1 1 1 1		D					A					B					0 0 0			1 0 1 0				1 1 1		0
neg*x*	0 1 1 1 1 1		D					A					0 0 0 0 0					OE	0 0		1 1 0 1				0 0 0		Rc
mul*x* [3]	0 1 1 1 1 1		D					A					B					OE	0 0		1 1 0 1				0 1 1		Rc
lbzux	0 1 1 1 1 1		D					A					B					0 0 0			1 1 1 0				1 1 1		0
nor*x*	0 1 1 1 1 1		S					A					B					0 0 0			1 1 1 1				1 0 0		Rc
subfe*x*	0 1 1 1 1 1		D					A					B					OE	0 1		0 0 0 1				0 0 0		Rc
adde*x*	0 1 1 1 1 1		D					A					B					OE	0 1		0 0 0 1				0 1 0		Rc
mtcrf	0 1 1 1 1 1		S					0	CRM								0	0 0 1			0 0 1 0				0 0 0		0
mtmsr	0 1 1 1 1 1		S					0 0 0 0 0					0 0 0 0 0					0 0 1			0 0 1 0				0 1 0		0
stdx [5]	0 1 1 1 1 1		S					A					B					0 0 1			0 0 1 0				1 0 1		0
stwcx.	0 1 1 1 1 1		S					A					B					0 0 1			0 0 1 0				1 1 0		1
stwx	0 1 1 1 1 1		S					A					B					0 0 1			0 0 1 0				1 1 1		0
slq*x* [3]	0 1 1 1 1 1		S					A					B					0 0 1			0 0 1 1				0 0 0		Rc
sle*x* [3]	0 1 1 1 1 1		S					A					B					0 0 1			0 0 1 1				0 0 1		Rc
stdux [5]	0 1 1 1 1 1		S					A					B					0 0 1			0 1 1 0				1 0 1		0
stwux	0 1 1 1 1 1		S					A					B					0 0 1			0 1 1 0				1 1 1		0
sliq*x* [3]	0 1 1 1 1 1		S					A					SH					0 0 1			0 1 1 1				0 0 0		Rc
subfze*x*	0 1 1 1 1 1		D					A					0 0 0 0 0					OE	0 1		1 0 0 1				0 0 0		Rc
addze*x*	0 1 1 1 1 1		D					A					0 0 0 0 0					OE	0 1		1 0 0 1				0 1 0		Rc
mtsr	0 1 1 1 1 1		S					0	SR				0 0 0 0 0					0 0 1			1 0 1 0				0 1 0		0
stdcx. [5]	0 1 1 1 1 1		S					A					B					0 0 1			1 0 1 0				1 1 0		1
stbx	0 1 1 1 1 1		S					A					B					0 0 1			1 0 1 0				1 1 1		0

Name	0	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	31
sllq*x* [3]	0 1 1 1 1	1 1	S	A	B		0 0 1 1 0 1 1 0 0 0	Rc
sleq*x* [3]	0 1 1 1 1	1 1	S	A	B		0 0 1 1 0 1 1 0 0 1	Rc
subfme*x*	0 1 1 1 1	1 1	D	A	0 0 0 0 0	OE	0 1 1 1 0 1 0 0 0	Rc
mulld [5]	0 1 1 1 1	1 1	D	A	B	OE	0 1 1 1 0 1 0 0 1	Rc
addme*x*	0 1 1 1 1	1 1	D	A	0 0 0 0 0	OE	0 1 1 1 0 1 0 1 0	Rc
mullw*x*	0 1 1 1 1	1 1	D	A	B	OE	0 1 1 1 0 1 0 1 1	Rc
mtsrin	0 1 1 1 1	1 1	S	0 0 0 0 0	B		0 0 1 1 1 1 0 1 0	0
dcbtst	0 1 1 1 1	1 1	0 0 0 0 0	A	B		0 0 1 1 1 1 0 1 1 0	0
stbux	0 1 1 1 1	1 1	S	A	B		0 0 1 1 1 1 0 1 1 1	0
slliq*x* [3]	0 1 1 1 1	1 1	S	A	SH		0 0 1 1 1 1 1 0 0 0	Rc
doz*x* [3]	0 1 1 1 1	1 1	D	A	B	OE	1 0 0 0 0 1 0 0 0	Rc
add*x*	0 1 1 1 1	1 1	D	A	B	OE	1 0 0 0 0 1 0 1 0	Rc
lscbx*x* [3, 4]	0 1 1 1 1	1 1	D	A	B		0 1 0 0 0 1 0 1 0 1	Rc
dcbt	0 1 1 1 1	1 1	0 0 0 0 0	A	B		0 1 0 0 0 1 0 1 1 0	0
lhzx	0 1 1 1 1	1 1	D	A	B		0 1 0 0 0 1 0 1 1 1	0
eqv*x*	0 1 1 1 1	1 1	S	A	B		0 1 0 0 0 1 1 1 0 0	Rc
tlbie	0 1 1 1 1	1 1	0 0 0 0 0	0 0 0 0 0	B		0 1 0 0 1 1 0 1 0	0
eciwx	0 1 1 1 1	1 1	D	A	B		0 1 0 0 1 1 0 1 1 0	0
lhzux	0 1 1 1 1	1 1	D	A	B		0 1 0 0 1 1 0 1 1 1	0
xor*x*	0 1 1 1 1	1 1	S	A	B		0 1 0 0 1 1 1 1 0 0	Rc
div*x* [3]	0 1 1 1 1	1 1	D	A	B	OE	1 0 1 0 0 1 0 1 1	Rc
mfspr	0 1 1 1 1	1 1	D	SPR			0 1 0 1 0 1 0 1 1	0
lwax [5]	0 1 1 1 1	1 1	D	A	B		0 1 0 1 0 1 0 1 0 1	0
lhax	0 1 1 1 1	1 1	D	A	B		0 1 0 1 0 1 0 1 1 1	0
abs*x* [3]	0 1 1 1 1	1 1	D	A	0 0 0 0 0	OE	1 0 1 1 0 1 0 0 0	Rc
divs*x* [3]	0 1 1 1 1	1 1	D	A	B	OE	1 0 1 1 0 1 0 1 1	Rc
tlbia	0 1 1 1 1	1 1	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0		0 1 0 1 1 1 0 1 0	0
mftb	0 1 1 1 1	1 1	D	TBR			0 1 0 1 1 1 0 1 1	0
lwaux [5]	0 1 1 1 1	1 1	D	A	B		0 1 0 1 1 1 0 1 0 1	0
lhaux	0 1 1 1 1	1 1	D	A	B		0 1 0 1 1 1 0 1 1 1	0
sthx	0 1 1 1 1	1 1	S	A	B		0 1 1 0 0 1 0 1 1 1	0
orc*x*	0 1 1 1 1	1 1	S	A	B		0 1 1 0 0 1 1 1 0 0	Rc
sradi*x* [5]	0 1 1 1 1	1 1	S	A	SH		1 1 0 0 1 1 1 0 1	SH Rc

Name	0	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	31
slbie [5]	0 1 1 1 1 1		0 0 0 0 0					0 0 0 0 0					B					0 1 1			0 1 1 0			0 1 0			0
ecowx	0 1 1 1 1 1		S					A					B					0 1 1			0 1 1 0			1 1 0			0
sthux	0 1 1 1 1 1		S					A					B					0 1 1			0 1 1 0			1 1 1			0
or*x*	0 1 1 1 1 1		S					A					B					0 1 1			0 1 1 1			1 0 0			Rc
divdu*x* [5]	0 1 1 1 1 1		D					A					B					OE		1 1		1 0 0 1			0 0 1			Rc
divwu*x*	0 1 1 1 1 1		D					A					B					OE		1 1		1 0 0 1			0 1 1			Rc
mtspr	0 1 1 1 1 1		D					SPR										0 1 1			1 0 1 0			0 1 1			0
dcbi	0 1 1 1 1 1		0 0 0 0 0					A					B					0 1 1			1 0 1 0			1 1 0			0
nand*x*	0 1 1 1 1 1		S					A					B					0 1 1			1 0 1 1			1 0 0			Rc
nabs*x* [3]	0 1 1 1 1 1		D					A					0 0 0 0 0					OE		1 1		1 1 0 1			0 0 0			Rc
divd*x* [5]	0 1 1 1 1 1		D					A					B					OE		1 1		1 1 0 1			0 0 1			Rc
divw*x*	0 1 1 1 1 1		D					A					B					OE		1 1		1 1 0 1			0 1 1			Rc
slbia [5]	0 1 1 1 1 1		0 0 0 0 0					0 0 0 0 0					0 0 0 0 0					0 1 1			1 1 1 0			0 1 0			0
mcrxr	0 1 1 1 1 1		crfS			0 0		0 0 0 0 0					0 0 0 0 0					1 0 0			0 0 0 0			0 0 0			0
clcs [3]	0 1 1 1 1 1		D					A					0 0 0 0 0					1 0 0			0 0 1 0			0 1 1			Rc
lswx [4]	0 1 1 1 1 1		D					A					B					1 0 0			0 0 1 0			1 0 1			0
lwbrx	0 1 1 1 1 1		D					A					B					1 0 0			0 0 1 0			1 1 0			0
lfsx	0 1 1 1 1 1		D					A					B					1 0 0			0 0 1 0			1 1 1			0
srw*x*	0 1 1 1 1 1		S					A					B					1 0 0			0 0 1 1			0 0 0			Rc
rrib*x* [3]	0 1 1 1 1 1		S					A					B					1 0 0			0 0 1 1			0 0 1			Rc
srd*x* [5]	0 1 1 1 1 1		S					A					B					1 0 0			0 0 1 1			0 1 1			Rc
maskir*x* [3]	0 1 1 1 1 1		S					A					B					1 0 0			0 0 1 1			1 0 1			Rc
tlbsync	0 1 1 1 1 1		0 0 0 0 0					0 0 0 0 0					0 0 0 0 0					1 0 0			0 1 1 0			1 1 0			0
lfsux	0 1 1 1 1 1		D					A					B					1 0 0			0 1 1 0			1 1 1			0
mfsr	0 1 1 1 1 1		D					0	SR				0 0 0 0 0					1 0 0			1 0 1 0			0 1 1			0
lswi [4]	0 1 1 1 1 1		D					A					NB					1 0 0			1 0 1 0			1 0 1			0
sync	0 1 1 1 1 1		0 0 0 0 0					0 0 0 0 0					0 0 0 0 0					1 0 0			1 0 1 0			1 1 0			0
lfdx	0 1 1 1 1 1		D					A					B					1 0 0			1 0 1 0			1 1 1			0
lfdux	0 1 1 1 1 1		D					A					B					1 0 0			1 1 1 0			1 1 1			0
mfsrin [1]	0 1 1 1 1 1		D					0 0 0 0 0					B					1 0 1			0 0 1 0			0 1 1			0
stswx [4]	0 1 1 1 1 1		S					A					B					1 0 1			0 0 1 0			1 0 1			0
stwbrx	0 1 1 1 1 1		S					A					B					1 0 1			0 0 1 0			1 1 0			0
stfsx	0 1 1 1 1 1		frS					A					B					1 0 1			0 0 1 0			1 1 1			0

Name	0	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	31
srq*x* [3]	0 1 1 1	1 1	S					A					B					1 0 1		0 0 1 1			0 0 0					Rc
sre*x* [3]	0 1 1 1	1 1	S					A					B					1 0 1		0 0 1 1			0 0 1					Rc
stfsux	0 1 1 1	1 1	frS					A					B					1 0 1		0 1 1 0			1 1 1					0
sriq*x* [3]	0 1 1 1	1 1	S					A					SH					1 0 1		0 1 1 1			0 0 0					Rc
stswi [4]	0 1 1 1	1 1	S					A					NB					1 0 1		1 0 1 0			1 0 1					0
stfdx	0 1 1 1	1 1	frS					A					B					1 0 1		1 0 1 0			1 1 1					0
srlq*x* [3]	0 1 1 1	1 1	S					A					B					1 0 1		1 0 1 1			0 0 0					Rc
sreq*x* [3]	0 1 1 1	1 1	S					A					B					1 0 1		1 0 1 1			0 0 1					Rc
stfdux	0 1 1 1	1 1	frS					A					B					1 0 1		1 1 1 0			1 1 1					0
srliq*x* [3]	0 1 1 1	1 1	S					A					SH					1 0 1		1 1 1 1			0 0 0					Rc
lhbrx	0 1 1 1	1 1	D					A					B					1 1 0		0 0 1 0			1 1 0					0
sraw*x*	0 1 1 1	1 1	S					A					B					1 1 0		0 0 1 1			0 0 0					Rc
srad*x* [5]	0 1 1 1	1 1	S					A					B					1 1 0		0 0 1 1			0 1 0					Rc
srawi*x*	0 1 1 1	1 1	S					A					SH					1 1 0		0 1 1 1			0 0 0					Rc
eieio	0 1 1 1	1 1	0 0 0 0 0					0 0 0 0 0					0 0 0 0 0					1 1 0		1 0 1 0			1 1 0					0
sthbrx	0 1 1 1	1 1	S					A					B					1 1 1		0 0 1 0			1 1 0					0
sraq*x* [3]	0 1 1 1	1 1	S					A					B					1 1 1		0 0 1 1			0 0 0					Rc
srea*x* [3]	0 1 1 1	1 1	S					A					B					1 1 1		0 0 1 1			0 0 1					Rc
extsh*x*	0 1 1 1	1 1	S					A					0 0 0 0 0					1 1 1		0 0 1 1			0 1 0					Rc
sraiq*x* [3]	0 1 1 1	1 1	S					A					SH					1 1 1		0 1 1 1			0 0 0					Rc
extsb*x*	0 1 1 1	1 1	S					A					0 0 0 0 0					1 1 1		0 1 1 1			0 1 0					Rc
icbi	0 1 1 1	1 1	0 0 0 0 0					A					B					1 1 1		1 0 1 0			1 1 0					0
stfiwx	0 1 1 1	1 1	frS					A					B					1 1 1		1 0 1 0			1 1 1					0
extsw [5]	0 1 1 1	1 1	S					A					0 0 0 0 0					1 1 1		1 0 1 1			0 1 0					Rc
dcbz	0 1 1 1	1 1	0 0 0 0 0					A					B					1 1 1		1 1 1 0			1 1 0					0
lwz	1 0 0 0	0 0	D					A					d															
lwzu	1 0 0 0	0 1	D					A					d															
lbz	1 0 0 0	1 0	D					A					d															
lbzu	1 0 0 0	1 1	D					A					d															
stw	1 0 0 1	0 0	S					A					d															
stwu	1 0 0 1	0 1	S					A					d															
stb	1 0 0 1	1 0	S					A					d															
stbu	1 0 0 1	1 1	S					A					d															

Name	0	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	31
lhz	1 0 1 0 0 0		D					A					d															
lhzu	1 0 1 0 0 1		D					A					d															
lha	1 0 1 0 1 0		D					A					d															
lhau	1 0 1 0 1 1		D					A					d															
sth	1 0 1 1 0 0		S					A					d															
sthu	1 0 1 1 0 1		S					A					d															
lmw [4]	1 0 1 1 1 0		D					A					d															
stmw [4]	1 0 1 1 1 1		S					A					d															
lfs	1 1 0 0 0 0		D					A					d															
lfsu	1 1 0 0 0 1		D					A					d															
lfd	1 1 0 0 1 0		D					A					d															
lfdu	1 1 0 0 1 1		D					A					d															
stfs	1 1 0 1 0 0		frS					A					d															
stfsu	1 1 0 1 0 1		frS					A					d															
stfd	1 1 0 1 1 0		frS					A					d															
stfdu	1 1 0 1 1 1		frS					A					d															
ld [5]	1 1 1 0 1 0		D					A					ds														0 0
ldu [5]	1 1 1 0 1 0		D					A					ds														0 1
lwa [5]	1 1 1 0 1 0		D					A					ds														1 0
fdivs*x*	1 1 1 0 1 1		D					A					B					0 0 0 0 0					1 0 0 1 0				Rc
fsubs*x*	1 1 1 0 1 1		D					A					B					0 0 0 0 0					1 0 1 0 0				Rc
fadds*x*	1 1 1 0 1 1		D					A					B					0 0 0 0 0					1 0 1 0 1				Rc
fsqrts*x*	1 1 1 0 1 1		D					0 0 0 0 0					B					0 0 0 0 0					1 0 1 1 0				Rc
fres*x*	1 1 1 0 1 1		D					0 0 0 0 0					B					0 0 0 0 0					1 1 0 0 0				Rc
fmuls*x*	1 1 1 0 1 1		D					A					0 0 0 0 0					C					1 1 0 0 1				Rc
fmsubs*x*	1 1 1 0 1 1		D					A					B					C					1 1 1 0 0				Rc
fmadds*x*	1 1 1 0 1 1		D					A					B					C					1 1 1 0 1				Rc
fnmsubs*x*	1 1 1 0 1 1		D					A					B					C					1 1 1 1 0				Rc
fnmadds*x*	1 1 1 0 1 1		D					A					B					C					1 1 1 1 1				Rc
std [5]	1 1 1 1 1 0		S					A					ds														0 0
stdu [5]	1 1 1 1 1 0		S					A					ds														0 1
fcmpu	1 1 1 1 1 1		crfD			0 0		A					B					0 0 0		0 0 0 0			0 0 0				0
frsp*x*	1 1 1 1 1 1		D					0 0 0 0 0					B					0 0 0		0 0 0 1			1 0 0				Rc

Name	0 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	31
fctiw*x*	1 1 1 1 1	D		0 0 0 0 0	B	0 0 0	0 0 0 1 1 1 0	Rc
fctiwz*x*	1 1 1 1 1	D		0 0 0 0 0	B	0 0 0	0 0 0 1 1 1 1	Rc
fdiv*x*	1 1 1 1 1	D		A	B	0 0 0 0 0	1 0 0 1 0	Rc
fsub*x*	1 1 1 1 1	D		A	B	0 0 0 0 0	1 0 1 0 0	Rc
fadd*x*	1 1 1 1 1	D		A	B	0 0 0 0 0	1 0 1 0 1	Rc
fsqrt*x*	1 1 1 1 1	D		0 0 0 0 0	B	0 0 0 0 0	1 0 1 1 0	Rc
fsel*x*	1 1 1 1 1	D		A	B	frC	1 0 1 1 1	Rc
fmul*x*	1 1 1 1 1	D		A	0 0 0 0 0	C	1 1 0 0 1	Rc
frsqrte*x*	1 1 1 1 1	D		0 0 0 0 0	B	0 0 0 0 0	1 1 0 1 0	Rc
fmsub*x*	1 1 1 1 1	D		A	B	C	1 1 1 0 0	Rc
fmadd*x*	1 1 1 1 1	D		A	B	C	1 1 1 0 1	Rc
fnmsub*x*	1 1 1 1 1	D		A	B	C	1 1 1 1 0	Rc
fnmadd*x*	1 1 1 1 1	D		A	B	C	1 1 1 1 1	Rc
fcmpo	1 1 1 1 1	crfD	0 0	A	B	0 0 0	0 1 0 0 0 0 0	0
mtfsb1*x*	1 1 1 1 1	crbD		0 0 0 0 0	0 0 0 0 0	0 0 0	0 1 0 0 1 1 0	Rc
fneg*x*	1 1 1 1 1	D		0 0 0 0 0	B	0 0 0	0 1 0 1 0 0 0	Rc
mcrfs	1 1 1 1 1	crfD	0 0	crfS 0 0	0 0 0 0 0	0 0 0	1 0 0 0 0 0 0	0
mtfsb0*x*	1 1 1 1 1	crbD		0 0 0 0 0	0 0 0 0 0	0 0 0	1 0 0 0 1 1 0	Rc
fmr*x*	1 1 1 1 1	D		0 0 0 0 0	B	0 0 0	1 0 0 1 0 0 0	Rc
mtfsfi*x*	1 1 1 1 1	crbD	0 0	0 0 0 0 0	IMM 0	0 0 1	0 0 0 0 1 1 0	Rc
fnabs*x*	1 1 1 1 1	D		0 0 0 0 0	B	0 0 1	0 0 0 1 0 0 0	Rc
fabs*x*	1 1 1 1 1	D		0 0 0 0 0	B	0 1 0	0 0 0 1 0 0 0	Rc
mffs*x*	1 1 1 1 1	D		0 0 0 0 0	0 0 0 0 0	1 0 0	1 0 0 0 1 1 1	Rc
mtfsf*x*	1 1 1 1 1	0	FM	0	B	1 0 1	1 0 0 0 1 1 1	Rc
fctid*x* [5]	1 1 1 1 1	D		0 0 0 0 0	B	1 1 0	0 1 0 1 1 1 0	Rc
fctidz*x* [5]	1 1 1 1 1	D		0 0 0 0 0	B	1 1 0	0 1 0 1 1 1 1	Rc
fcfid*x* [5]	1 1 1 1 1	D		0 0 0 0 0	B	1 1 0	1 0 0 1 1 1 0	Rc

[1] Supervisor-level instruction.
[2] Supervisor- and user-level instruction.
[3] Non-PowerPC instruction implemented by 601 for POWER architecture compatibliity.
[4] Load or store string or multiple word instruction.
[5] 64-bit instruction.

A.3 Instructions Grouped by Functional Categories

Table A-3 through Table A-30 list the instructions grouped by functional categories.

Table A-3. Integer Arithmetic Instructions

Key:

Reserved bits

Instruction not implemented in the 601

Name	EU	0	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	31
abs*x* [3]	IU	31		D					A					0	0	0	0	0	OE			360						Rc
addc*x*	IU	31		D					A					B					OE			10						Rc
adde*x*	IU	31		D					A					B					OE			138						Rc
addi	IU	14		D					A					SIMM														
addic	IU	12		D					A					SIMM														
addic.	IU	13		D					A					SIMM														
addis	IU	15		D					A					SIMM														
addme*x*	IU	31		D					A					0	0	0	0	0	OE			234						Rc
add*x*	IU	31		D					A					B					OE			266						Rc
addze*x*	IU	31		D					A					0	0	0	0	0	OE			202						Rc
div*x* [3]	IU	31		D					A					B					OE			331						Rc
divs*x* [3]	IU	31		D					A					B					OE			363						Rc
divw*x*	IU	31		D					A					B					OE			491						Rc
divwu*x*	IU	31		D					A					B					OE			459						Rc
doz*x* [3]	IU	31		D					A					B					OE			264						Rc
dozi [3]	IU	9		D					A					SIMM														
mul*x* [3]	IU	31		D					A					B					OE			107						Rc
mulhw*x*	IU	31		D					A					B					0			75						Rc
mulhwu*x*	IU	31		D					A					B					0			11						Rc
mulli	IU	7		D					A					SIMM														
mullw*x*	IU	31		D					A					B					OE			235						Rc
nabs*x* [3]	IU	31		D					A					0	0	0	0	0	OE			488						Rc
neg*x*	IU	31		D					A					0	0	0	0	0	OE			104						Rc
subf*x*	IU	31		D					A					B					OE			40						Rc
subfc*x*	IU	31		D					A					B					OE			8						Rc
subfe*x*	IU	31		D					A					B					OE			136						Rc
subfme*x*	IU	31		D					A					0	0	0	0	0	OE			232						Rc

subfze*x*	IU	31	D	A	0 0 0 0 0	OE	200	Rc

Table A-4. Integer Compare Instructions

Name	EU	0	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	31
cmp	IU	31		crfD			0	L			A			B					0										0
cmpi	IU	11		crfD			0	L			A			SIMM														
cmpl	IU	31		crfD			0	L			A			B					32										0
cmpli	IU	10		crfD			0	L			A			UIMM														

Table A-5. Integer Logical Instructions

Name	EU	0	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	31
and*x*	IU	31		S							A			B					28										Rc
andc*x*	IU	31		S							A			B					60										Rc
andi.	IU	28		S							A			UIMM														
andis.	IU	29		S							A			UIMM														
cntlzw*x*	IU	31		S							A			0 0 0 0 0					26									Rc
eqv*x*	IU	31		S							A			B					284									Rc
extsb*x*	IU	31		S							A			0 0 0 0 0					954									Rc
extsh*x*	IU	31		S							A			0 0 0 0 0					922									Rc
nand*x*	IU	31		S							A			B					476									Rc
nor*x*	IU	31		S							A			B					124									Rc
or*x*	IU	31		S							A			B					444									Rc
orc*x*	IU	31		S							A			B					412									Rc
ori	IU	24		S							A			UIMM														
oris	IU	25		S							A			UIMM														
xor*x*	IU	31		S							A			B					316									Rc
xori	IU	26		S							A			UIMM														
xoris	IU	27		S							A			UIMM														

Table A-6. Integer Rotate Instructions

Name	EU	0 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	31
maskg*x*[3]	IU	31	S	A	B	29		Rc
maskir*x*[3]	IU	31	S	A	B	541		Rc
rlmi*x*[3]	IU	22	S	A	SH	MB	ME	Rc
rlwimi*x*	IU	20	S	A	SH	MB	ME	Rc
rlwinm*x*	IU	21	S	A	SH	MB	ME	Rc
rlwnm*x*	IU	23	S	A	B	MB	ME	Rc
rrib*x*[3]	IU	31	S	A	B	537		Rc

Table A-7. Integer Shift Instructions

Name	EU	0 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	31
sle*x*[3]	IU	31	S	A	B	153	Rc
sleq*x*[3]	IU	31	S	A	B	217	Rc
sliq*x*[3]	IU	31	S	A	SH	184	Rc
slliq*x*[3]	IU	31	S	A	SH	248	Rc
sllq*x*[3]	IU	31	S	A	B	216	Rc
slq*x*[3]	IU	31	S	A	B	152	Rc
slw*x*	IU	31	S	A	B	24	Rc
sraq*x*[3]	IU	31	S	A	B	920	Rc
sraiq*x*[3]	IU	31	S	A	SH	952	Rc
sraw*x*	IU	31	S	A	B	792	Rc
srawi*x*	IU	31	S	A	SH	824	Rc
sre*x*[3]	IU	31	S	A	B	665	Rc
srea*x*[3]	IU	31	S	A	B	921	Rc
sreq*x*[3]	IU	31	S	A	B	729	Rc
sriq*x*[3]	IU	31	S	A	SH	696	Rc
srliq*x*[3]	IU	31	S	A	SH	760	Rc
srlq*x*[3]	IU	31	S	A	B	728	Rc
srq*x*[3]	IU	31	S	A	B	664	Rc
srw*x*	IU	31	S	A	B	536	Rc

Table A-8. Floating-Point Arithmetic Instructions

Name	EU	0 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	31
fadd*x*	FPU	63	D		A	B	0 0 0 0 0	21	Rc
fadds*x*	FPU	59	D		A	B	0 0 0 0 0	21	Rc
fdiv*x*	FPU	63	D		A	B	0 0 0 0 0	18	Rc
fdivs*x*	FPU	59	D		A	B	0 0 0 0 0	18	Rc
fmul*x*	FPU	63	D		A	0 0 0 0 0	C	25	Rc
fmuls*x*	FPU	59	D		A	0 0 0 0 0	C	25	Rc
fsub*x*	FPU	63	D		A	B	0 0 0 0 0	20	Rc
fsubs*x*	FPU	59	D		A	B	0 0 0 0 0	20	Rc

Table A-9. Floating-Point Rounding and Conversion Instructions

Name	EU	0 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	31
fctiw*x*	FPU	63	D		0 0 0 0 0	B	14	Rc
fctiwz*x*	FPU	63	D		0 0 0 0 0	B	15	Rc
frsp*x*	FPU	63	D		0 0 0 0 0	B	12	Rc

Table A-10. Floating-Point Compare Instructions

Name	EU	0 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	31
fcmpo	FPU	63	crfD	0 0	A	B	32	0
fcmpu	FPU	63	crfD	0 0	A	B	0	0

Table A-11. Floating-Point Status and Control Register Instructions

Name	EU	0 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	31
mcrfs	FPU	63	crfD	0 0	crfS 0 0	0 0 0 0 0	64	0
mffs*x*	FPU	63	D		0 0 0 0 0	0 0 0 0 0	583	Rc
mtfsb0*x*	FPU	63	crbD		0 0 0 0 0	0 0 0 0 0	70	Rc
mtfsb1*x*	FPU	63	crbD		0 0 0 0 0	0 0 0 0 0	38	Rc
mtfsf*x*	FPU	63	0	FM	0	B	711	Rc
mtfsfi*x*	FPU	63	crbD	0 0	0 0 0 0 0	IMM 0	134	Rc

Table A-12. Integer Load Instructions

Name	EU	0 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	31
lbz	IU	34	D	A	d		
lbzu	IU	35	D	A	d		
lbzux	IU	31	D	A	B	119	0
lbzx	IU	31	D	A	B	87	0
lha	IU	42	D	A	d		
lhau	IU	43	D	A	d		
lhaux	IU	31	D	A	B	375	0
lhax	IU	31	D	A	B	343	0
lhz	IU	40	D	A	d		
lhzu	IU	41	D	A	d		
lhzux	IU	31	D	A	B	311	0
lhzx	IU	31	D	A	B	279	0
lwz	IU	32	D	A	d		
lwzu	IU	33	D	A	d		
lwzux	IU	31	D	A	B	55	0
lwzx	IU	31	D	A	B	23	0

Table A-13. Integer Store Instructions

Name	EU	0 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	31
stb	IU	38	S	A	d		
stbu	IU	39	S	A	d		
stbux	IU	31	S	A	B	247	0
stbx	IU	31	S	A	B	215	0
sth	IU	44	S	A	d		
sthu	IU	45	S	A	d		
sthux	IU	31	S	A	B	439	0
sthx	IU	31	S	A	B	407	0
stw	IU	36	S	A	d		
stwu	IU	37	S	A	d		
stwux	IU	31	S	A	B	183	0
stwx	IU	31	S	A	B	151	0

Table A-14. Integer Load and Store with Byte Reversal Instructions

Name	EU	0 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	31
lhbrx	IU	31	D	A	B	790	0
lwbrx	IU	31	D	A	B	534	0
sthbrx	IU	31	S	A	B	918	0
stwbrx	IU	31	S	A	B	662	0

Table A-15. Integer Load and Store Multiple Instructions

Name	EU	0 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 31
lmw [4]	IU	46	D	A	d	
stmw [4]	IU	47	S	A	d	

Table A-16. Integer Move String Instructions

Name	EU	0 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	31
lscbx*x* [3, 4]	IU	31	D	A	B	277	Rc
lswi [4]	IU	31	D	A	NB	597	0
lswx [4]	IU	31	D	A	B	533	0
stswi [4]	IU	31	S	A	NB	725	0
stswx [4]	IU	31	S	A	B	661	0

Table A-17. Memory Synchronization Instructions

Name	EU	0 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	31
eieio	IU	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	854	0
isync	IU	19	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	150	0
lwarx	IU	31	D	A	B	20	0
stwcx.	IU	31	S	A	B	150	1
sync	IU	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	598	0

Table A-18. Floating-Point Load Instructions

Name	EU	0	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	31
lfd	FPU	50		D					A					d														
lfdu	FPU	51		D					A					d														
lfdux	FPU	31		D					A					B					631									0
lfdx	FPU	31		D					A					B					599									0
lfs	FPU	48		D					A					d														
lfsu	FPU	49		D					A					d														
lfsux	FPU	31		D					A					B					567									0
lfsx	IU, FPU	31		D					A					B					535									0

Table A-19. Floating-Point Store Instructions

Name	EU	0	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	31
stfd	FPU	54		frS					A					d														
stfdu	FPU	55		frS					A					d														
stfdux	FPU	31		frS					A					B					759									0
stfdx	FPU	31		frS					A					B					727									0
stfs	IU, FPU	52		frS					A					d														
stfsu	IU, FPU	53		frS					A					d														
stfsux	IU, FPU	31		frS					A					B					695									0
stfsx	IU, FPU	31		frS					A					B					663									0

Table A-20. Floating-Point Move Instructions

Name	EU	0	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	31
fabs*x*	FPU	63		D					0	0	0	0	0	B					264									Rc
fmr*x*	FPU	63		D					0	0	0	0	0	B					72									Rc
fnabs*x*	FPU	63		D					0	0	0	0	0	B					136									Rc
fneg*x*	FPU	63		D					0	0	0	0	0	B					40									Rc

Table A-21. Branch Instructions

Name	EU	0	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	31
b*x*	BPU	18	LI																									AA	LK
bc*x*	BPU	16	BO					BI						BD														AA	LK
bcctr*x*	BPU	19	BO					BI						0 0 0 0 0					528										LK
bclr*x*	BPU	19	BO					BI						0 0 0 0 0					16										LK

Table A-22. Condition Register Logical Instructions

Name	EU	0	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	31
crand	IU	19	crbD					crbA						crbB					257										0
crandc	IU	19	crbD					crbA						crbB					129										0
creqv	IU	19	crbD					crbA						crbB					289										0
crnand	IU	19	crbD					crbA						crbB					225										0
crnor	IU	19	crbD					crbA						crbB					33										0
cror	IU	19	crbD					crbA						crbB					449										0
crorc	IU	19	crbD					crbA						crbB					417										0
crxor	IU	19	crbD					crbA						crbB					193										0
mcrf	IU	19	crfD				0 0	crfS					0 0	0 0 0 0 0					0										0

Table A-23. System Linkage Instructions

Name	EU	0	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	31
rfi [1]	IU	19	0 0 0 0 0					0 0 0 0 0						0 0 0 0 0					50										0
sc	IU	17	0 0 0 0 0					0 0 0 0 0						0 0 0 0 0 0 0 0 0 0 0 0 0 0 0														1	0

Table A-24. Trap Instructions

Name	EU	0	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	31
tw	IU	31	TO					A						B					4										0
twi	IU	03	TO					A						SIMM															

Table A-25. Move to/from Special Purpose Register Instructions

Name	EU	0 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	31
mcrxr	IU	31	crfS 0 0	0 0 0 0 0	0 0 0 0 0	512	0
mfcr	IU	31	D	0 0 0 0 0	0 0 0 0 0	19	0
mfmsr [1]	IU	31	D	0 0 0 0 0	0 0 0 0 0	83	0
mfspr [2]	IU	31	D	SPR		339	0
mtcrf	IU	31	S	0 CRM	0	144	0
mtmsr [1]	IU	31	S	0 0 0 0 0	0 0 0 0 0	146	0
mtspr [2]	IU	31	D	SPR		467	0

Table A-26. Cache Management Supervisor-Level Instruction

Name	EU	0 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	31
dcbi [1]	IU	31	0 0 0 0 0	A	B	470	0

Table A-27. User-Level Cache Instructions

Name	EU	0 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	31
clcs [3]	IU	31	D	A	0 0 0 0 0	531	Rc
dcbf	IU	31	0 0 0 0 0	A	B	86	0
dcbst	IU	31	0 0 0 0 0	A	B	54	0
dcbt	IU	31	0 0 0 0 0	A	B	278	0
dcbtst	IU	31	0 0 0 0 0	A	B	246	0
dcbz	IU	31	0 0 0 0 0	A	B	1014	0

Table A-28. Segment Register Manipulation Instructions

Name	EU	0 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	31
mfsr [1]	IU	31	D	0 SR	0 0 0 0 0	595	0
mfsrin [1]	IU	31	D	0 0 0 0 0	B	659	0
mtsr [1]	IU	31	S	0 SR	0 0 0 0 0	210	0
mtsrin [1]	IU	31	S	0 0 0 0 0	B	242	0

Table A-29. Translation Lookaside Buffer Management Instruction

Name	EU	0 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	31
tlbie [1]	IU	31	0 0 0 0 0	0 0 0 0 0	B	306	0

Table A-30. External Control Instructions

Name	EU	0	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	31
eciwx	IU	31	D					A					B					310											0
ecowx	IU	31	S					A					B					438											0

[1] Supervisor-level instruction.
[2] Supervisor- and user-level instruction.
[3] Non-PowerPC instruction implemented by the 601 for POWER architecture compatibility.
[4] Load or store string or multiple word instruction.
[5] 64-bit instruction.

A.4 Complete Instruction List Sorted by Form

Table A-31 through Table A-45 list the instructions grouped by form.

Key:

Reserved bits

Instruction not implemented in the 601

Table A-31. I-Form

OPCD	LI	AA	LK

Specific Instruction

Name	0	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	31
b*x*	18	LI																									AA	LK

Table A-32. B-Form

OPCD	BO	BI	BD	AA	LK

Specific Instruction

Name	0	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	31
bc*x*	16	BO						BI								BD											AA	LK

Table A-33. SC-Form

OPCD	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1	0

Specific Instruction

Name	0	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	31
sc	17	0 0 0 0 0						0 0 0 0 0								0 0 0 0 0 0 0 0 0 0 0 0 0 0 0												1	0

Table A-34. D-Form

OPCD	D	A	d
OPCD	D	A	SIMM
OPCD	S	A	d
OPCD	S	A	UIMM
OPCD	crfD 0 L	A	SIMM
OPCD	crfD 0 L	A	UIMM
OPCD	TO	A	SIMM

OPCD	D	A	d
OPCD	frS	A	d

Specific Instructions

Name	0 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 31
addi	14	D			A	SIMM
addic	12	D			A	SIMM
addic.	13	D			A	SIMM
addis	15	D			A	SIMM
andi.	28	S			A	UIMM
andis.	29	S			A	UIMM
cmpi	11	crfD	0	L	A	SIMM
cmpli	10	crfD	0	L	A	UIMM
dozi [3]	9	D			A	SIMM
lbz	34	D			A	d
lbzu	35	D			A	d
lfd	50	D			A	d
lfdu	51	D			A	d
lfs	48	D			A	d
lfsu	49	D			A	d
lha	42	D			A	d
lhau	43	D			A	d
lhz	40	D			A	d
lhzu	41	D			A	d
lmw [4]	46	D			A	d
lwz	32	D			A	d
lwzu	33	D			A	d
mulli	7	D			A	SIMM
ori	24	S			A	UIMM
oris	25	S			A	UIMM
stb	38	S			A	d
stbu	39	S			A	d
stfd	54	frS			A	d
stfdu	55	frS			A	d
stfs	52	frS			A	d

stfsu	53	frS	A	d
sth	44	S	A	d
sthu	45	S	A	d
stmw [4]	47	S	A	d
stw	36	S	A	d
stwu	37	S	A	d
subfic	08	D	A	SIMM
tdi [5]	02	TO	A	SIMM
twi	03	TO	A	SIMM
xori	26	S	A	UIMM
xoris	27	S	A	UIMM

Table A-35. DS-Form

OPCD	D	A	ds	2

Specific Instructions

Name	0 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 31
ld [5]	58	D	A	ds	0
ldu [5]	58	D	A	ds	1
lwa [5]	58	D	A	ds	2
std [5]	62	S	A	ds	0 0
stdu [5]	62	S	A	ds	1

Table A-36. X-Form

OPCD	D	A	B	XO	0
OPCD	D	A	B	XO	Rc
OPCD	D	0 0 0 0 0	B	XO	0
OPCD	D	0 0 0 0 0	B	XO	Rc
OPCD	D	0 0 0 0 0	0 0 0 0 0	XO	0
OPCD	D	0 0 0 0 0	0 0 0 0 0	XO	Rc
OPCD	D	0 SR	0 0 0 0 0	XO	0
OPCD	S	A	B	XO	Rc
OPCD	S	A	B	XO	1
OPCD	S	A	B	XO	0
OPCD	S	A	0 0 0 0 0	XO	Rc

OPCD	S	0 0 0 0 0	B	XO	0
OPCD	S	0 0 0 0 0	0 0 0 0 0	XO	0
OPCD	S	0 SR	0 0 0 0 0	XO	0
OPCD	S	A	SH	XO	Rc
OPCD	crfD 0 L	A	B	XO	0
OPCD	crfD 0 0	A	B	XO	0
OPCD	crfD 0 0	crfS 0 0	0 0 0 0 0	XO	0
OPCD	crfS 0 0	0 0 0 0 0	0 0 0 0 0	XO	0
OPCD	crbD 0 0	0 0 0 0 0	IMM 0	XO	Rc
OPCD	TO	A	B	XO	0
OPCD	0 0 0 0 0	A	B	XO	0
OPCD	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	XO	0

Specific Instructions

Name	0	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	31
and*x*	31	S						A					B					28										Rc
andc*x*	31	S						A					B					60										Rc
cmp	31	crfD			0	L		A					B					0										0
cmpl	31	crfD			0	L		A					B					32										0
cntlzd [5]	31	S						A					0 0 0 0 0					58										Rc
cntlzw*x*	31	S						A					0 0 0 0 0					26										Rc
dcbf	31	0 0 0 0 0						A					B					86										0
dcbi [1]	31	0 0 0 0 0						A					B					470										0
dcbst	31	0 0 0 0 0						A					B					54										0
dcbt	31	0 0 0 0 0						A					B					278										0
dcbtst	31	0 0 0 0 0						A					B					246										0
dcbz	31	0 0 0 0 0						A					B					1014										0
eciwx	31	D						A					B					310										0
ecowx	31	S						A					B					438										0
eieio	31	0 0 0 0 0						0 0 0 0 0					0 0 0 0 0					854										0
eqv*x*	31	S						A					B					284										Rc
extsb*x*	31	S						A					0 0 0 0 0					954										Rc
extsh*x*	31	S						A					0 0 0 0 0					922										Rc
extsw [5]	31	S						A					0 0 0 0 0					986										Rc
fabs*x*	63	D						0 0 0 0 0					B					264										Rc

fctid [5]	63	D	0 0 0 0 0	B	814	Rc
fctidz [5]	63	D	0 0 0 0 0	B	815	Rc
fcfid [5]	63	D	0 0 0 0 0	B	846	Rc
fctiw*x*	63	D	0 0 0 0 0	B	14	Rc
fctiwz*x*	63	D	0 0 0 0 0	B	15	Rc
fcmpo	63	crfD 0 0	A	B	32	0
fcmpu	63	crfD 0 0	A	B	0 0 0 0 0 0 0 0 0 0	0
fmr*x*	63	D	0 0 0 0 0	B	72	Rc
fnabs*x*	63	D	0 0 0 0 0	B	136	Rc
fneg*x*	63	D	0 0 0 0 0	B	40	Rc
frsp*x*	63	D	0 0 0 0 0	B	12	Rc
icbi	31	0 0 0 0 0	A	B	982	0
lbzux	31	D	A	B	119	0
lbzx	31	D	A	B	87	0
ldux [5]	31	D	A	B	53	0
ldx [5]	31	D	A	B	21	0
ldarx [5]	31	D	A	B	84	0
lfdux	31	D	A	B	631	0
lfdx	31	D	A	B	599	0
lfsux	31	D	A	B	567	0
lfsx	31	D	A	B	535	0
lhaux	31	D	A	B	375	0
lhax	31	D	A	B	343	0
lhbrx	31	D	A	B	790	0
lhzux	31	D	A	B	311	0
lhzx	31	D	A	B	279	0
lscbx*x* [3, 4]	31	D	A	B	277	Rc
lswi [4]	31	D	A	B	597	0
lswx [4]	31	D	A	B	533	0
lwarx	31	D	A	B	20	0
lwbrx	31	D	A	B	534	0
lwzux	31	D	A	B	55	0
lwzx	31	D	A	B	23	0
lwaux [5]	31	D	A	B	373	0
lwax [5]	31	D	A	B	341	0

maskg*x* [3]	31	S	A	B	29	Rc
maskir*x* [3]	31	S	A	B	541	Rc
mcrfs	63	crfD 0 0	crfS 0 0	0 0 0 0 0	64	0
mcrxr	31	crfS 0 0	0 0 0 0 0	0 0 0 0 0	512	0
mfcr	31	D	0 0 0 0 0	0 0 0 0 0	19	0
mffs*x*	63	D	0 0 0 0 0	0 0 0 0 0	583	Rc
mfmsr [1]	31	D	0 0 0 0 0	0 0 0 0 0	83	0
mfsr [1]	31	D	0 SR	0 0 0 0 0	595	0
mfsrin [1]	31	D	0 0 0 0 0	B	659	0
mtfsb0*x*	63	crbD	0 0 0 0 0	0 0 0 0 0	70	Rc
mtfsb1*x*	63	crbD	0 0 0 0 0	0 0 0 0 0	38	Rc
mtfsfi*x*	63	crbD 0 0	0 0 0 0 0	IMM 0	134	Rc
mtmsr [1]	31	S	0 0 0 0 0	0 0 0 0 0	146	0
mtsr [1]	31	S	0 SR	0 0 0 0 0	210	0
mtsrin [1]	31	S	0 0 0 0 0	B	242	0
nand*x*	31	S	A	B	476	Rc
nor*x*	31	S	A	B	124	Rc
or*x*	31	S	A	B	444	Rc
orc*x*	31	S	A	B	412	Rc
rrib*x* [3]	31	S	A	B	537	Rc
slbia [1, 5]	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	498	0
slbie [1, 5]	31	0 0 0 0 0	0 0 0 0 0	B	434	0
sld [5]	31	S	A	B	27	Rc
sle*x* [3]	31	S	A	B	153	Rc
sleq*x* [3]	31	S	A	B	217	Rc
sliq*x* [3]	31	S	A	SH	184	Rc
slliq*x* [3]	31	S	A	SH	248	Rc
sllq*x* [3]	31	S	A	B	216	Rc
slq*x* [3]	31	S	A	B	152	Rc
slw*x*	31	S	A	B	24	Rc
srad [5]	31	S	A	B	794	Rc
sraq*x* [3]	31	S	A	B	920	Rc
sraiq*x* [3]	31	S	A	SH	952	Rc
sraw*x*	31	S	A	B	792	Rc
srawi*x*	31	S	A	SH	824	Rc

srd [5]	31	S	A	B	539	Rc
sre*x* [3]	31	S	A	B	665	Rc
srea*x* [3]	31	S	A	B	921	Rc
sreq*x* [3]	31	S	A	B	729	Rc
sriq*x* [3]	31	S	A	SH	696	Rc
srliq*x* [3]	31	S	A	SH	760	Rc
srlq*x* [3]	31	S	A	B	728	Rc
srq*x* [3]	31	S	A	B	664	Rc
srw*x*	31	S	A	B	536	Rc
stbux	31	S	A	B	247	0
stbx	31	S	A	B	215	0
stdcx. [5]	31	S	A	B	214	1
stdux [5]	31	S	A	B	181	0
stdx [5]	31	S	A	B	149	0
stfdux	31	frS	A	B	759	0
stfdx	31	frS	A	B	727	0
stfsux	31	frS	A	B	695	0
stfiwx	31	frS	A	B	983	0
stfsx	31	frS	A	B	663	0
sthbrx	31	S	A	B	918	0
sthux	31	S	A	B	439	0
sthx	31	S	A	B	407	0
stswi [4]	31	S	A	NB	725	0
stswx [4]	31	S	A	B	661	0
stwbrx	31	S	A	B	662	0
stwcx.	31	S	A	B	150	1
stwx	31	S	A	B	151	0
stwux	31	S	A	B	183	0
sync	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	598	0
td [5]	31	TO	A	B	68	0
tlbia [1]	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	370	0
tlbie [1]	31	0 0 0 0 0	0 0 0 0 0	B	306	0
tlbsync [1]	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	566	0
tw	31	TO	A	B	4	0
xor*x*	31	S	A	B	316	Rc

Table A-37. XL-Form

OPCD	BO	BI	0 0 0 0 0	XO	LK
OPCD	crbD	crbA	crbB	XO	0
OPCD	crfD 0 0	crfS 0 0	0 0 0 0 0	0 0 0 0 0 0 0 0 0 0	0
OPCD	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	XO	0

Specific Instructions

Name	0 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	31
bcctr*x*	19	BO	BI	0 0 0 0 0	528	LK
bclr*x*	19	BO	BI	0 0 0 0 0	16	LK
crand	19	crbD	crbA	crbB	257	0
creqv	19	crbD	crbA	crbB	289	0
crnand	19	crbD	crbA	crbB	225	0
crnor	19	crbD	crbA	crbB	33	0
cror	19	crbD	crbA	crbB	449	0
crorc	19	crbD	crbA	crbB	417	0
crxor	19	crbD	crbA	crbB	193	0
isync	19	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	150	0
mcrf	19	crfD 0 0	crfS 0 0	0 0 0 0 0	0 0 0 0 0 0 0 0 0 0	0
rfi [1]	19	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	50	0

Table A-38. XFS-Form

OPCD	D	SPR	XO	0
OPCD	S	0 CRM 0	XO	0
OPCD	D	TBR	XO	0

Specific Instructions

Name	0 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	31
mfspr [2]	31	D	SPR		339	0
mtspr [2]	31	D	SPR		467	0
mtcrf	31	S	0 CRM 0		144	0
mftb	31	D	TBR		371	0

Table A-39. XFL-Form

OPCD	0	FM	0	frB	XO	Rc

Specific Instruction

Name	0	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	31
mtfsf*x*	31		0				FM					0			frB			711										Rc

Table A-40. XS-Form

OPCD	S	A	SH	XO	SH	Rc

Specific Instruction

Name	0	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	31
sradi [5]	31			S				A					SH					413									SH	Rc

Table A-41. XO-Form

OPCD	D	A	B	OE	XO	Rc
OPCD	D	A	B	0	XO	Rc
OPCD	D	A	0 0 0 0 0	OE	XO	Rc

Specific Instructions

Name	0	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	31
abs*x* [3]	31			D				A					0 0 0 0 0					OE										Rc
add*x*	31			D				A					B					OE										Rc
addc*x*	31			D				A					B					OE										Rc
adde*x*	31			D				A					B					OE										Rc
addme*x*	31			D				A					0 0 0 0 0					OE										Rc
addze*x*	31			D				A					0 0 0 0 0					OE										Rc
div*x* [3]	31			D				A					B					OE										Rc
divd [5]	31			D				A					B					OE										Rc
divdu [5]	31			D				A					B					OE										Rc
divs*x* [3]	31			D				A					B					OE										Rc
divw*x*	31			D				A					B					OE										Rc
divwu*x*	31			D				A					B					OE										Rc
doz*x* [3]	31			D				A					B					OE										Rc
mul*x* [3]	31			D				A					B					OE										Rc
mulhd [5]	31			D				A					B					0										Rc

mulhdu [5]	31	D	A	B	0	9	Rc
mulhw*x*	31	D	A	B	0	75	Rc
mulhwu*x*	31	D	A	B	0	11	Rc
mulld [5]	31	D	A	B	OE	233	Rc
mullw*x*	31	D	A	B	OE	235	Rc
nabs*x* [3]	31	D	A	0 0 0 0 0	OE	488	Rc
neg*x*	31	D	A	0 0 0 0 0	OE	104	Rc
subf*x*	31	D	A	B	OE	40	Rc
subfc*x*	31	D	A	B	OE	8	Rc
subfe*x*	31	D	A	B	OE	136	Rc
subfme*x*	31	D	A	0 0 0 0 0	OE	232	Rc
subfze*x*	31	D	A	0 0 0 0 0	OE	200	Rc

Table A-42. A-Form

OPCD	D	A	B	0 0 0 0 0	XO	Rc
OPCD	D	A	B	C	XO	Rc
OPCD	D	A	0 0 0 0 0	C	XO	Rc
OPCD	frD	0 0 0 0 0	frB	0 0 0 0 0	XO	Rc

Specific Instructions

Name	0 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	31
fadd*x*	63	D	A	B	0 0 0 0 0	21	Rc
fadds*x*	59	D	A	B	0 0 0 0 0	21	Rc
fdiv*x*	63	D	A	B	0 0 0 0 0	18	Rc
fdivs*x*	59	D	A	B	0 0 0 0 0	18	Rc
fmadd*x*	63	D	A	B	C	29	Rc
fmadds*x*	59	D	A	B	C	29	Rc
fmsub*x*	63	D	A	B	C	28	Rc
fmsubs*x*	59	D	A	B	C	28	Rc
fmul*x*	63	D	A	0 0 0 0 0	C	25	Rc
fmuls*x*	59	D	A	0 0 0 0 0	C	25	Rc
fnmadd*x*	63	D	A	B	C	31	Rc
fnmadds*x*	59	D	A	B	C	31	Rc
fnmsub*x*	63	D	A	B	C	30	Rc
fnmsubs*x*	59	D	A	B	C	30	Rc
fres*x*	59	frD	0 0 0 0 0	frB	0 0 0 0 0	24	Rc

frsqrte	63	frD	0 0 0 0 0	frB	0 0 0 0 0	26	Rc
fsel*x*	63	frD	frA	frB	frC	23	Rc
fsqrt*x*	63	frD	0 0 0 0 0	frB	0 0 0 0 0	22	Rc
fsqrts	59	frD	0 0 0 0 0	frB	0 0 0 0 0	22	Rc
fsub*x*	63	D	A	B	0 0 0 0 0	20	Rc
fsubs*x*	59	D	A	B	0 0 0 0 0	20	Rc

Table A-43. M-Form

OPCD	S	A	SH	MB	ME	Rc
OPCD	S	A	B	MB	ME	Rc

Specific Instructions

Name	0	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	31
rlmi*x* [3]	22		S				A					SH					MB					ME					Rc
rlwimi*x*	20		S				A					SH					MB					ME					Rc
rlwinm*x*	21		S				A					SH					MB					ME					Rc
rlwnm*x*	23		S				A					B					MB					ME					Rc

Table A-44. MD-Form

OPCD	S	A	SH	MB	XO	SH	Rc
OPCD	S	A	SH	ME	XO	SH	Rc

Specific Instructions

Name	0	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	31
rldic [5]	30		S				A					SH					MB					2				SH	Rc
rldicl [5]	30		S				A					SH					MB					0				SH	Rc
rldicr [5]	30		S				A					SH					ME					1				SH	Rc
rldimi [5]	30		S				A					SH					MB					3				SH	Rc

Table A-45. MDS-Form

OPCD	S	A	B	MB	XO	SH	Rc
OPCD	S	A	SH	ME	XO	SH	Rc

Specific Instructions

Name	0 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	31
rldcl [5]	30	S	A	B	MB	8	Rc
rldcr [5]	30	S	A	B	ME	9	Rc

[1] Supervisor-level instruction.
[2] Supervisor- and user-level instruction.
[3] Non-PowerPC instruction implemented by the 601 for POWER architecture compatibility.
[4] Load or store string or multiple word instruction.
[5] 64-bit instruction.

Appendix B
POWER Architecture Cross Reference

This appendix identifies the incompatibilities that must be managed in migration from the POWER architecture to PowerPC architecture. Some of the incompatibilities can, at least in principle, be detected by the processor, which traps and lets software simulate the POWER operation. Others cannot be detected by the processor.

In general, the incompatibilities identified here are those that affect a POWER application program. Incompatibilities for instructions that can be used only by POWER system programs are not discussed. Note that this appendix describes incompatibilities with respect to the PowerPC architecture in general. The PowerPC 601 microprocessor is more closely compatible with the POWER architecture. POWER instructions implemented in the 601 are listed in Table B-4.

B.1 New Instructions, Formerly Supervisor-Level Instructions

Instructions new to the PowerPC architecture typically use opcode values (including extended opcode) that are illegal in the POWER architecture. A few instructions that are supervisor-level in the POWER architecture (for example, **dclz**, called **dcbz** in the PowerPC architecture) have been made non-supervisor-level in the PowerPC architecture. Any POWER program that executes one of these now-valid or now-non-supervisor-level instructions, expecting to cause the system illegal instruction error handler (program exception) or the system supervisor-level instruction error handler to be invoked, will not execute correctly on PowerPC processors.

B.2 Newly Supervisor-Level Instructions

The following instructions are user-level in the POWER architecture but are supervisor-level in PowerPC processors.

- **mfmsr**
- **mfsr**

B.3 Reserved Bits in Instructions

These are shown with '/'s in the instruction opcode definitions. In the POWER architecture such bits are ignored by the processor. In PowerPC architecture they must be 0 or the instruction form is invalid. In several cases the PowerPC architecture assumes that such bits in POWER instructions are indeed 0. The cases include the following:

- **cmpi**, **cmp**, **cmpli**, and **cmpl** assume that bit 10 in the POWER instructions is 0.
- **mtspr** and **mfspr** assume that bits 16–20 in the POWER instructions are 0.

B.4 Reserved Bits in Registers

The POWER architecture defines these bits to be 0 when read, and either 0 or 1 when written to. In the PowerPC architecture it is implementation-dependent, for each register, whether these bits are 0 when read and ignored when written to or are copied from source to target when read or written to.

B.5 Alignment Check

The AL bit in the POWER machine-state register, MSR[24], is not supported in the PowerPC architecture. The bit is reserved in the PowerPC architecture. The low-order bits of the EA are always used. Notice that the value 0—the normal value for a reserved SPR bit—means “ignore the low-order EA bits” in the POWER architecture, and the value 1 means “use the low-order EA bits.” However, MSR[24] is not assigned new meaning in the PowerPC architecture.

B.6 Condition Register

The following instructions specify a field in the CR explicitly (via the BF field) and also have the record bit option. In the PowerPC architecture, if Rc = 1 for these instructions the instruction form is invalid. In the POWER architecture, if Rc = 1 the instructions execute normally except as shown in Table B-1.

Table B-1. Condition Register Settings

Instruction	Setting
cmp	CR0 is undefined if Rc = 1 and BF ≠ 0
cmpl	CR0 is undefined if Rc = 1 and BF ≠ 0
mcrxr	CR0 is undefined if Rc = 1 and BF ≠ 0
fcmpu	CR1 is undefined if Rc = 1
fcmpo	CR1 is undefined if Rc = 1
mcrfs	CR1 is undefined if Rc = 1 and BF ≠ 1

B.7 Inappropriate Use of LK and Rc bits

For the instructions listed below, if LK = 1 or Rc = 1, POWER processors execute the instruction normally with the exception of setting the link register (if LK = 1) or the condition register field 0 or 1 (if Rc = 1) to an undefined value. In the PowerPC architecture, such instruction forms are invalid.

PowerPC instruction form is invalid if LK = 1:

- **sc** (**svc** in the POWER architecture)
- Condition register logical instructions
- **mcrf**
- **isync** (**ics** in the POWER architecture)

PowerPC instruction form is invalid if Rc=1:

- Integer X-form load and store instructions
- Integer X-form compare instructions
- X-form trap instruction
- **mtspr**, **mfspr**, **mtcrf**, **mcrxr**, **mfcr**, **mtmsr**, **mfmsr**, **mtsr**, **mtsrin**, **tlbi**, **eciwx**, **ecowx**, **clcs**, **mfsr**, **mfsrin**, **sync**, **eieio**, **icli**
- Floating-point X-form load and store instructions and floating-point compare instructions
- **mcrfs**
- **dcbz** (**dclz** in the POWER architecture)

B.8 BO Field

The POWER architecture shows certain bits in the BO field—used by branch conditional instructions—as x without indicating how these bits are to be interpreted. These bits are ignored by POWER processors. The PowerPC architecture treats these bits differently, as shown in Table B-2.

Table B-2. Differences in the BO Field

BO Field	Description
BO[0–3]	The PowerPC architecture shows the bits as z. If it is not cleared, the instruction form is invalid.
BO[4]	This bit, which is shown as x in the POWER architecture independent of the other four bits—is shown in the PowerPC architecture as y. It gives a hint about whether the branch is likely to be taken. If a POWER program has the wrong value for this bit, the program runs correctly but performance may suffer.

B.9 Branch Conditional to Count Register

For the case in which the count register is decremented and tested (that is, the case in which BO[2] = 0), the POWER architecture specifies only that the branch target address is undefined, implying that the count register, and the link register (if LK = 1), are updated in the normal way. The PowerPC architecture considers this instruction form invalid.

B.10 System Call/Supervisor Call

The System Call (**sc**) instruction in the PowerPC architecture is called Supervisor Call (**svc***x*) in the POWER architecture. Differences in implementations are as follows:

- The POWER architecture provides a version of the Supervisor Call instruction (bit 30 = 0) that allows instruction fetching to continue at any one of 128 locations. It is used for “fast SVCs.” The PowerPC architecture provides no such version
- The POWER architecture provides a version of the Supervisor Call instruction (bits 30–31 = b'11') that resumes instruction fetching at one location and sets the link register to the address of the next instruction. The PowerPC architecture provides no such version; if bit 31 of the instruction is 1, the instruction form is invalid.
- For the POWER architecture, information from the MSR is saved in the count register. For the PowerPC architecture, this information is saved in SRR1.
- The POWER architecture permits bits 16–29 of the Supervisor Call (**sc**) instruction to be nonzero, while in the PowerPC architecture, such an instruction form is invalid.

  Because bits 16–29 of the **sc** instruction are regarded as reserved for the POWER architecture, they are ignored by the 601.
- The POWER architecture saves the low-order 16 bits of the Supervisor Call instruction in the count register; the PowerPC architecture does not save them.
- The settings of the MSR bits by the system call exception differ between the POWER architecture and the PowerPC architecture.

B.11 Integer Exception Register (XER)

Bits 18–23 of the XER are reserved in the PowerPC architecture, whereas in the POWER architecture they are defined to contain the comparison byte for the **lscbx** instruction, which is not included in the PowerPC architecture.

B.12 Update Forms of Memory Access

The PowerPC architecture requires that **r**A not be equal to either **r**D (integer load only) or 0. If the restriction is violated, the instruction form is invalid. See Appendix D, “Classes of Instructions,” for information about invalid instructions. The POWER architecture permits these cases and simply avoids saving the EA.

B.13 Multiple Register Loads

The PowerPC architecture requires that **r**A and **r**B if present in the instruction format, not be in the range of registers to be loaded, while the POWER architecture permits this and does not alter **r**A or **r**B in this case. (The PowerPC architecture restriction applies even if **r**A = 0, although there is no obvious benefit to the restriction in this case since **r**A is not used to compute the effective address if **r**A = 0.) If the PowerPC architecture restriction is violated, the instruction form is invalid. The instructions affected are listed as follows:

- **lmw** (**lm** in the POWER architecture)
- **lswi** (**lsi** in the POWER architecture)
- **lswx** (**lsx** in the POWER architecture)

Thus, for example, an **lmw** instruction that loads all 32 registers is valid in the POWER architecture but is an invalid form in the PowerPC architecture.

B.14 Alignment for Load/Store Multiple

The PowerPC architecture requires the EA to be word-aligned and yields an alignment exception or boundedly undefined results if it is not. The POWER architecture specifies that an alignment exception occurs (if AL = 1).

B.15 Load String Instructions

In the PowerPC architecture, an **lswx** instruction with zero length leaves the content of **r**D undefined, while in the POWER architecture the corresponding instruction (**lsx**) does not alter **r**D.

B.16 Synchronization

The **sync** instruction (called **dcs** in the POWER architecture) causes a much more pervasive synchronization in the PowerPC architecture than in the POWER architecture. For more information, refer to Chapter 10, “Instruction Set.”

B.17 Move to/from SPR

Differences in how the Move to/from Special Purpose Register (**mtspr** and **mfspr**) instructions are as follows:

- The SPR field is 10 bits long in the PowerPC architecture, but only 5 in POWER architecture.
- The **mfspr** instruction can be used to read the decrementer (DEC) register in user-level mode in the POWER architecture, but only in supervisor state in the PowerPC architecture.
- If the SPR value specified in the instruction is not one of the defined values, the PowerPC architecture considers the instruction form invalid. (In user mode, the allowed SPR values exclude those accessible only in supervisor mode.) The POWER architecture does not alter any architected registers in this case and generates a program exception if the instruction is executed in user mode and SPR[0]=1.

For PowerPC processors except the 601 processor, a program exception is generated for an attempt to execute an **mtspr** or **mfspr** instruction with SPR[0–4]=0 (which denotes the MQ register). Similarly, a program exception is generated for attempts to execute an **mfspr** instruction with SPR[0–4] = 6 (which denotes reading the decrementer register in the POWER architecture).

B.18 Effects of Exceptions on FPSCR Bits FR and FI

For the following cases, the POWER architecture does not specify how the FR and FI bits are set, while the PowerPC architecture preserves them for illegal operation exceptions caused by compare instructions and clears them otherwise:

- Invalid operation exception (enabled or disabled)
- Zero divide exception (enabled or disabled)
- Disabled overflow exception

B.19 Floating-Point Store Instructions

The POWER architecture uses FPSCR[UE] to help determine whether denormalization should be done, while the PowerPC architecture does not. Using FPSCR[UE] is in fact incorrect: in the PowerPC architecture if FPSCR[UE] = 1 and a denormalized single-precision number is copied from one memory location to another by means of an **lfs** instruction followed by an **stfs** instruction, the two “copies” may not be the same.

B.20 Move from FPSCR

The POWER architecture defines the high-order 32 bits of the result of **mffs** to be x'FFFF FFFF'. In the PowerPC architecture they are undefined.

B.21 Clearing Bytes in the Data Cache

The **dclz** instruction of the POWER architecture and the **dcbz** instruction of the PowerPC architecture have the same opcode. However, the functions differ in the following respects:

- The **dclz** instruction clears a line; **dcbz** clears a block (a sector in the 601).
- The **dclz** instruction saves the EA in **r**A (if **r**A ≠ 0); **dcbz** does not.
- The **dclz** instruction is supervisor-level; **dcbz** is not.

B.22 Segment Register Instructions

The definitions of the four segment register instructions (**mtsr**, **mtsrin**, **mfsr**, and **mfsrin**) differ in two respects between the POWER architecture and the PowerPC architecture. Instructions similar to **mtsrin** and **mfsrin** are called **mtsri** and **mfsri** in the POWER architecture.

Privilege—**mfsr** and **mfsri** are user-level instructions in the POWER architecture, while **mfsr** and **mfsrin** are supervisor-level in the PowerPC architecture.

Function—the indirect instructions (**mtsri** and **mfsri**) in the POWER architecture use an **r**A register in computing the segment register number, and the computed EA is stored into **r**A (if **r**A ≠ 0 and **r**A ≠ **r**D); in the PowerPC architecture **mtsrin** and **mfsrin** have no **r**A field and EA is not stored.

The **mtsr**, **mtsrin** (**mtsri**), and **mfsr** instructions have the same opcodes in the PowerPC architecture as in the POWER architecture. The **mfsri** instruction in the POWER architecture and the **mfsrin** instruction in PowerPC architecture have different opcodes.

B.23 TLB Entry Invalidation

The **tlbi** instruction of the POWER architecture and the **tlbie** instruction of the PowerPC architecture have the same opcode. However, the functions differ in the following respects.

- The **tlbi** instruction computes the EA as (**r**A|0) + (**r**B), while **tlbie** lacks an **r**A field and computes the EA as (**r**B).
- The **tlbi** instruction saves the EA in **r**A (if **r**A ≠ 0); **tlbie** lacks an **r**A field and does not save the EA.

B.24 Timing Facilities

This section describes differences between the POWER architecture and the PowerPC architecture timer facilities.

B.24.1 Real-Time Clock

The 601 implements a POWER-based RTC. Note that the POWER RTC is not supported in the PowerPC architecture. Instead, the PowerPC architecture provides a time base (TB).

Both the RTC and the time base are 64-bit special purpose registers, but they differ in the following respects.

- The RTC counts seconds and nanoseconds, while the TB counts “ticks.” The frequency of the RTC is implementation-dependent.
- The RTC increments discontinuously—1 is added to RTCU when the value in RTCL passes 999_999_999. The TB increments continuously—1 is added to TBU when the value in TBL passes x'FFFF FFFF'.
- The RTC is written and read by the **mtspr** and **mfspr** instructions, using SPR numbers that denote the RTCU and RTCD. The TB is written to and read by the instructions **mtspr** and **mftb**.
- The SPR numbers that denote RTCL and RTCU are invalid in the PowerPC architecture except the 601.
- The RTC is guaranteed to increment at least once in the time required to execute 10 Add Immediate (**addi**) instructions. No analogous guarantee is made for the TB.
- Not all bits of RTCL need be implemented, while all bits of the TB must be implemented.

B.24.2 Decrementer

The PowerPC architecture DEC register decrements at the same rate that the TB increments, while the POWER decrementers decrement every nanosecond (which is the same rate that the RTC increments).

Not all bits of the POWER DEC need be implemented, while all bits of the PowerPC architecture DEC must be implemented.

B.25 Deleted Instructions

Table B-3 lists the instructions that are part of the POWER architecture but have been dropped from the PowerPC architecture. The POWER instructions that are implemented in the 601 processor are identified in the table

Table B-3. POWER Instructions Deleted from PowerPC Architecture

Mnemonic	Instruction	Primary Opcode	Secondary Opcode	In PowerPC 601 Microprocessor
abs	Absolute	31	360	Yes
clcs	Cache Line Compute Size	31	531	Yes
clf	Cache Line Flush	31	118	No
cli	Cache Line Invalidate	31	502	No
dclst	Data Cache Line Store	31	630	No
div	Divide	31	331	Yes
divs	Divide Short	31	363	Yes

Table B-3. POWER Instructions Deleted from PowerPC Architecture (Continued)

Mnemonic	Instruction	Primary Opcode	Secondary Opcode	In PowerPC 601 Microprocessor
doz	Difference or Zero	31	264	Yes
dozi	Difference or Zero Immediate	09	—	Yes
lscbx	Load String and Compare Byte Indexed	31	277	Yes
maskg	Mask Generate	31	29	Yes
maskir	Mask Insert from Register	31	541	Yes
mfsrin	Move from Segment Register Indirect	31	627	Yes
mul	Multiply	31	107	Yes
nabs	Negative Absolute	31	488	Yes
rac	Real Address Compute	31	818	No
rlmi	Rotate Left then Mask Insert	22	—	Yes
rrib	Rotate Right and Insert Bit	31	537	Yes
sle	Shift Left Extended	31	153	Yes
sleq	Shift Left Extended with MQ	31	217	Yes
sliq	Shift Left Immediate with MQ	31	184	Yes
slliq	Shift Left Long Immediate with MQ	31	248	Yes
sllq	Shift Left Long with MQ	31	216	Yes
slq	Shift Left with MQ	31	152	Yes
sraiq	Shift Right Algebraic Immediate with MQ	31	952	Yes
sraq	Shift Right Algebraic with MQ	31	920	Yes
sre	Shift Right Extended	31	665	Yes
srea	Shift Right Extended Algebraic	31	921	Yes
sreq	Shift Right Extended with MQ	31	729	Yes
sriq	Shift Right Immediate with MQ	31	696	Yes
srliq	Shift Right Long Immediate with MQ	31	760	Yes
srlq	Shift Right Long with MQ	31	728	Yes
srq	Shift Right with MQ	31	664	Yes
svc*x*	Supervisor Call, with SA = 0	17	0	No

Note: Many of these instructions use the MQ register. The MQ is not defined in the PowerPC architecture, but is implemented in the 601 processor.

B.26 POWER Instructions Supported by the PowerPC Architecture

Table B-4 lists the POWER instructions implemented in the PowerPC architecture.

Table B-4. POWER Instructions Implemented in PowerPC Architecture

POWER		PowerPC	
Mnemonic	**Instruction**	**Mnemonic**	**Instruction**
a*x*	Add	**addc***x*	Add Carrying
ae*x*	Add Extended	**adde***x*	Add Extended
ai	Add Immediate	**addic**	Add Immediate Carrying
ai.	Add Immediate and Record	**addic.**	Add Immediate Carrying and Record
ame*x*	Add to Minus One Extended	**addme***x*	Add to Minus One Extended
andil.	AND Immediate Lower	**andi.**	AND Immediate
andiu.	AND Immediate Upper	**andis.**	AND Immediate Shifted
aze*x*	Add to Zero Extended	**addze***x*	Add to Zero Extended
bcc*x*	Branch Conditional to Count Register	**bcctr***x*	Branch Conditional to Count Register
bcr*x*	Branch Conditional to Link Register	**bclr***x*	Branch Conditional to Link Register
cal	Compute Address Lower	**addi**	Add Immediate
cau	Compute Address Upper	**addis**	Add Immediate Shifted
cax*x*	Compute Address	**add***x*	Add
cntlz*x*	Count Leading Zeros	**cntlzw***x*	Count Leading Zeros Word
dclz	Data Cache Line Set to Zero	**dcbz**	Data Cache Block Set to Zero
dcs	Data Cache Synchronize	**sync**	Synchronize
exts*x*	Extend Sign	**extsh***x*	Extend Sign Half Word
fa*x*	Floating Add	**fadd***x*	Floating-Point Add
fd*x*	Floating Divide	**fdiv***x*	Floating-Point Divide
fm*x*	Floating Multiply	**fmul***x*	Floating-Point Multiply
fma*x*	Floating Multiply-Add	**fmadd***x*	Floating-Point Multiply-Add
fms*x*	Floating Multiply-Subtract	**fmsub***x*	Floating-Point Multiply-Subtract
fnma*x*	Floating Negative Multiply-Add	**fnmadd***x*	Floating-Point Negative Multiply-Add
fnms*x*	Floating Negative Multiply-Subtract	**fnmsub***x*	Floating-Point Negative Multiply-Subtract
fs*x*	Floating Subtract	**fsub***x*	Floating-Point Subtract
ics	Instruction Cache Synchronize	**isync**	Instruction Synchronize
l	Load	**lwz**	Load Word and Zero
lbrx	Load Byte-Reverse Indexed	**lwbrx**	Load Word Byte-Reverse Indexed

Table B-4. POWER Instructions Implemented in PowerPC Architecture (Continued)

POWER		PowerPC	
Mnemonic	**Instruction**	**Mnemonic**	**Instruction**
lm	Load Multiple	**lmw**	Load Multiple Word
lsi	Load String Immediate	**lswi**	Load String Word Immediate
lsx	Load String Indexed	**lswx**	Load String Word Indexed
lu	Load with Update	**lwzu**	Load Word and Zero with Update
lux	Load with Update Indexed	**lwzux**	Load Word and Zero with Update Indexed
lx	Load Indexed	**lwzx**	Load Word and Zero Indexed
mtsri	Move to Segment Register Indirect	**mtsrin**	Move to Segment Register Indirect *
muli	Multiply Immediate	**mulli**	Multiply Low Immediate
muls*x*	Multiply Short	**mull***wx*	Multiply Low
oril	OR Immediate Lower	**ori**	OR Immediate
oriu	OR Immediate Upper	**oris**	OR Immediate Shifted
rlimi*x*	Rotate Left Immediate then Mask Insert	**rlwimi***x*	Rotate Left Word Immediate then Mask Insert
rlinm*x*	Rotate Left Immediate then AND With Mask	**rlwinm***x*	Rotate Left Word Immediate then AND with Mask
rlnm*x*	Rotate Left then AND with Mask	**rlwnm***x*	Rotate Left Word then AND with Mask
sf*x*	Subtract from	**subfc***x*	Subtract from Carrying
sfe*x*	Subtract from Extended	**subfe***x*	Subtract from Extended
sfi	Subtract from Immediate	**subfic**	Subtract from Immediate Carrying
sfme*x*	Subtract from Minus One Extended	**subfme***x*	Subtract from Minus One Extended
sfze*x*	Subtract from Zero Extended	**subfze***x*	Subtract from Zero Extended
sl*x*	Shift Left	**slw***x*	Shift Left Word
sr*x*	Shift Right	**srw***x*	Shift Right Word
sra*x*	Shift Right Algebraic	**sraw***x*	Shift Right Algebraic Word
srai*x*	Shift Right Algebraic Immediate	**srawi***x*	Shift Right Algebraic Word Immediate
st	Store	**stw**	Store Word
stbrx	Store Byte-Reverse Indexed	**stwbrx**	Store Word Byte-Reverse Indexed
stm	Store Multiple	**stmw**	Store Multiple Word
stsi	Store String Immediate	**stswi**	Store String Word Immediate
stsx	Store String Indexed	**stswx**	Store String Word Indexed
stu	Store with Update	**stwu**	Store Word with Update

Table B-4. POWER Instructions Implemented in PowerPC Architecture (Continued)

POWER		PowerPC	
Mnemonic	**Instruction**	**Mnemonic**	**Instruction**
stux	Store with Update Indexed	**stwux**	Store Word with Update Indexed
stx	Store Indexed	**stwx**	Store Word Indexed
svca	Supervisor Call	**sc**	System Call
t	Trap	**tw**	Trap Word
ti	Trap Immediate	**twi**	Trap Word Immediate *
tlbi	TLB Invalidate Entry	**tlbie**	Translation Lookaside Buffer Invalidate Entry
xoril	XOR Immediate Lower	**xori**	XOR Immediate
xoriu	XOR Immediate Upper	**xoris**	XOR Immediate Shifted

* Supervisor-level instruction

Appendix C
PowerPC Instructions Not Implemented

This appendix describes the 32-bit and 64-bit PowerPC instructions that are not implemented in the PowerPC 601 microprocessor. It also provides the 32-bit and 64-bit SPR encodings that are not implemented by the 601.

Table C-1 provides a list of 32-bit PowerPC instructions that are optional to the PowerPC architecture but not implemented by the 601. Attempting to execute these instructions causes an illegal instruction exception.

Table C-1. 32-Bit Instructions Not Implemented by the PowerPC 601 Microprocessor

Mnemonic	Instruction
fres	Floating-Point Reciprocal Estimate Single-Precision
frsqrte	Floating-Point Reciprocal Square Root Estimate
fsel	Floating-Point Select
fsqrt	Floating-Point Square Root
fsqrts	Floating-Point Square Root Single-Precision
mfspr [1]	Move from Special Purpose Register
mftb	Move from Time Base
mtspr [1]	Move to Special Purpose Register
stfiwx	Store Floating-Point as Integer Word Indexed
tlbia	Translation Lookaside Buffer Invalidate All
tlbsync	Translation Lookaside Buffer Synchronize

1 Some SPR encodings are not supported by the 601. These SPR encodings are listed in Table C-2 and Table C-4.

Table C-2 provides a list of 32-bit SPR encodings that are not implemented by the 601.

Table C-2. 32-Bit SPR Encodings Not Implemented by the PowerPC 601 Microprocessor

SPR			Register Name	Access
Decimal	SPR[5–9]	SPR[0–4]		
284	01000	11100	TB	Supervisor
285	01000	11101	TBU	Supervisor
536	10000	11000	DBAT0U	Supervisor
537	10000	11001	DBAT0L	Supervisor
538	10000	11010	DBAT1U	Supervisor
539	10000	11011	DBAT1L	Supervisor
540	10000	11100	DBAT2U	Supervisor
541	10000	11101	DBAT2L	Supervisor
542	10000	11110	DBAT3U	Supervisor
543	10000	11111	DBAT3L	Supervisor

Table C-3 provides a list of 64-bit instructions that are not implemented by the 601, and that generate an illegal instruction exception.

Table C-3. 64-Bit Instructions Not Implemented by the PowerPC 601 Microprocessor

Mnemonic	Instruction
cntlzd	Count Leading Zeros Double Word
divd	Divide Double Word
divdu	Divide Double Word Unsigned
extsw	Extend Sign Word
fcfid	Floating-Point Convert From Integer Double Word
fctid	Floating-Point Convert to Integer Double Word
fctidz	Floating-Point Convert to Integer Double Word with Round toward Zero
ld	Load Double Word
ldarx	Load Double Word and Reserve Indexed
ldu	Load Double Word with Update
ldux	Load Double Word with Update Indexed
ldx	Load Double Word Indexed
lwa	Load Word Algebraic
lwaux	Load Word Algebraic with Update Indexed

Table C-3. 64-Bit Instructions Not Implemented by the PowerPC 601 Microprocessor (Continued)

Mnemonic	Instruction
lwax	Load Word Algebraic Indexed
mulld	Multiply Low Double Word
mulhd	Multiply High Double Word
mulhdu	Multiply High Double Word Unsigned
rldcl	Rotate Left Double Word then Clear Left
rldcr	Rotate Left Double Word then Clear Right
rldic	Rotate Left Double Word Immediate then Clear
rldicl	Rotate Left Double Word Immediate then Clear Left
rldicr	Rotate Left Double Word Immediate then Clear Right
rldimi	Rotate Left Double Word Immediate then Mask Insert
slbia	SLB Invalidate All
slbie	SLB Invalidate Entry
sld	Shift Left Double Word
srad	Shift Right Algebraic Double Word
sradi	Shift Right Algebraic Double Word Immediate
srd	Shift Right Double Word
std	Store Double Word
stdcx.	Store Double Word Conditional Indexed
stdu	Store Double Word with Update
stdux	Store Double Word Indexed with Update
stdx	Store Double Word Indexed
td	Trap Double Word
tdi	Trap Double Word Immediate

Table C-4 provides the 64-bit SPR encoding that is not implemented by the 601.

Table C-4. 64-Bit SPR Encoding Not Implemented by the PowerPC 601 Microprocessor

SPR			Register Name	Access
Decimal	**SPR[5–9]**	**SPR[0–4]**		
280	01000	11000	ASR	Supervisor

cntlzd — Not Implemented in 601 — cntlzd

Count Leading Zeros Double Word — Integer Unit

cntlzd **r**A,**r**S (Rc=0)
cntlzd. **r**A,**r**S (Rc=1)

n←0
do while *n*<64
  if **r**S[*n*]=1 then leave
  n←*n*+1
rA←*n*

A count of the number of consecutive zero bits starting at bit 0 of register **r**S is placed into **r**A. This number ranges from 0 to 64, inclusive.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Other registers altered:

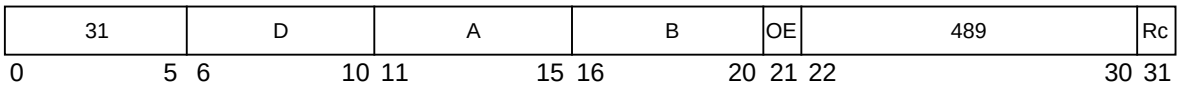
- Condition Register (CR0 Field):

  Affected: LT, GT, EQ, SO (Rc=1)

divd — Not Implemented in 601 — divd

Divide Double Word — Integer Unit

divd **r**D,**r**A,**r**B (OE=0 Rc=0)
divd. **r**D,**r**A,**r**B (OE=0 Rc=1)
divdo **r**D,**r**A,**r**B (OE=1 Rc=0)
divdo. **r**D,**r**A,**r**B (OE=1 Rc=1)



dividend[0–63]←**r**A
divisor[0–63]←**r**B
rD←dividend+divisor

The 64-bit dividend is **r**A. The 64-bit divisor is **r**B. The 64-bit quotient of the dividend and divisor is placed into **r**D. The remainder is not supplied as a result.

Both the dividend and the divisor are interpreted as signed integers. The quotient is the unique signed integer that satisfies

dividend=(quotient∗divisor)+r

where 0 ≤ r < |divisor| if the dividend is non-negative, and –|divisor| < r≤ 0 if the dividend is negative.

If an attempt is made to perform any of the divisions

0x8000_0000_0000_0000 ÷ -1
<anything> ÷ 0

then the contents of **r**D are undefined.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

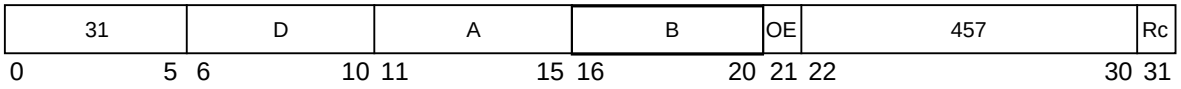
Other registers altered:

- Condition Register (CR0 Field):
  Affected: LT, GT, EQ, SO (if Rc=1)
- Exception Register:
  Affected: SO, OV (if OE=1)

divdu — Not Implemented in 601 — divdu

Divide Double Word Unsigned — Integer Unit

divdu	**r**D,**r**A,**r**B	(OE=0 Rc=0)
divdu.	**r**D,**r**A,**r**B	(OE=0 Rc=1)
divduo	**r**D,**r**A,**r**B	(OE=1 Rc=0)
divduo.	**r**D,**r**A,**r**B	(OE=1 Rc=1)



dividend[0–63]←**r**A
divisor[0–63]←**r**B
rD←dividend+divisor

The 64-bit dividend is **r**A. The 64-bit divisor is **r**B. The 64-bit quotient of the dividend and divisor is placed into **r**D. The remainder is not supplied as a result.

Both the dividend and the divisor are interpreted as unsigned integers. The quotient is the unique unsigned integer that satisfies

dividend=(quotient∗divisor)+r

where 0 ≤ r < divisor.

If an attempt is made to perform the division

<anything> ÷ 0

then the contents of **r**D are undefined.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

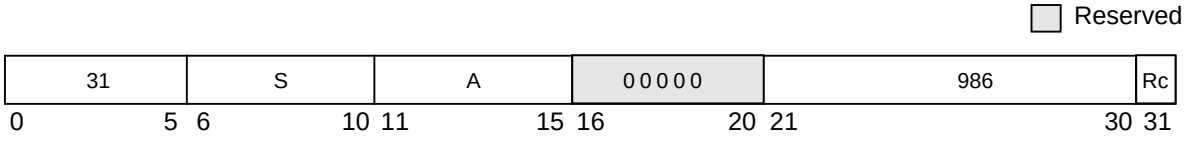
Other registers altered:

- Condition Register (CR0 Field):
  Affected: LT, GT, EQ, SO (if Rc=1)
- Exception Register:
  Affected: SO, OV (if OE=1)

extsw — Not Implemented in 601 — extsw

Extend Sign Word

extsw **r**A,**r**S (Rc=0)
extsw. **r**A,**r**S (Rc=1)



S←**r**S[32]
rA[32–63]←**r**S[32–63]
rA[0–31]←(32)S

Register **r**S[32–63] are placed into **r**A[32–63]. Bit 32 of **r**S is placed into **r**A[0–31].

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Other registers altered:

- Condition Register (CR0 Field):
  Affected: LT, GT, EQ, SO (if Rc=1)

fcfid — Not Implemented in 601 — fcfid

Floating-Point Convert from Integer Double Word — Floating-Point Unit

fcfid **fr**D,**fr**B (Rc=0)
fcfid. **fr**D,**fr**B (Rc=1)

The 64-bit signed fixed-point operand in register **fr**B is converted to an infinitely precise floating-point integer. If the result of the conversion is already in double-precision range it is placed into register **fr**D. Otherwise the result of the conversion is rounded to double-precision using the rounding mode specified by FPSCR[RN] and placed into register **fr**D.

FPSCR[FPRF] is set to the class and sign of the result. FPSCR[FR] is set if the result is incremented when rounded. FPSCR[FI] is set if the result is inexact.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Other registers altered:

- Condition Register (CR1 Field):

  Affected: LT, GT, EQ, SO (if Rc=1)
- Exception Register:

  Affected: FPRF, FR, FI, FX, XX

fctid — Not Implemented in 601 — fctid

Floating-Point Convert to Integer Double Word

fctid **fr**D,**fr**B (Rc=0)
fctid. **fr**D,**fr**B (Rc=1)

The floating-point operand in **fr**B is converted to a 64-bit signed fixed-point integer, using the rounding mode specified by FPSCR[RN], and placed into **fr**D.

If the operand in **fr**B is greater than $2^{63} - 1$, **fr**D is set to 0x'7FFF_FFFF_FFFF_FFFF'. If the operand in **fr**B is less than -2^{63}, then **fr**D is set to 0x'8000_0000_0000_0000'.

Except for enabled invalid operation exceptions, FPSCR[FPRF] is undefined. FPSCR[FR] is set if the result is incremented when rounded. FPSCR[FI] is set if the result is inexact.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Other registers altered:

- Condition Register (CR1 Field):

  Affected: LT, GT, EQ, SO (if Rc=1)
- Exception Register:

  Affected: FPRF (undefined), FR, FI, FX, XX, VXSNAN VXCVI

fctidz Not Implemented in 601 fctidz

Floating-Point Convert to Integer Double Word with Round toward Zero

fctidz **fr**D,**fr**B (Rc=0)
fctidz. **fr**D,**fr**B (Rc=1)

The floating-point operand in **fr**B is converted to a 64-bit signed fixed-point integer, using the rounding mode round toward zero, and placed into **fr**D.

If the operand in **fr**B is greater than $2^{63} - 1$, **fr**D is set to 0x'7FFF_FFFF_FFFF_FFFF'. If the operand in **fr**B is less than -2^{63}, then **fr**D is set to 0x'8000_0000_0000_0000'.

Except for enabled invalid operation exceptions, FPSCR[FPRF] is undefined. FPSCR[FR] is set if the result is incremented when rounded. FPSCR[FI] is set if the result is inexact.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Other registers altered:

- Condition Register (CR1 Field):

  Affected: LT, GT, EQ, SO (if Rc=1)
- Exception Register:

  Affected: FPRF (undefined), FR, FI, FX, XX, VXSNAN VXCVI

fres **fr**D,**fr**B (Rc=0)
fres. **fr**D,**fr**B (Rc=1)

This PowerPC instruction is not implemented by the 601. Execution of this instruction will invoke the illegal instruction handler. A description of the operation of this instruction is provided for emulation purposes.

A single-precision estimate of the reciprocal of the floating-point operand in register **fr**B is placed into register **fr**D. The estimate placed into register **fr**D is correct to a precision of one part in 256 of the reciprocal of **fr**B.

Operation with various special values of the operand is summarized below.

Operand	Result	Exception
-∞	-0	None
-0	-∞*	ZX
+0	+∞*	ZX
+∞	+0	None
SNaN	QNaN**	VXSNAN
QNaN	QNaN	None

* No result if FPSCR[ZE]=1.

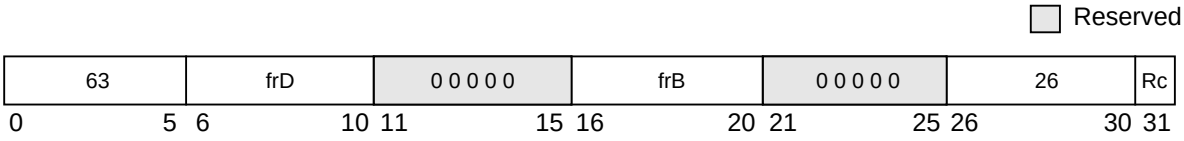
** No result if FPSCR[VE]=1.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1 and zero divide exceptions when FPSCR[ZE]=1.

Other registers altered:

- Condition Register (CR1 Field):
  Affected: FX, FEX, VX, OX (if Rc=1)
- Floating-point Status and Control Register:
  Affected: FX, OX, UX, ZX, VXSNAN, FPRF, FR (undefined), FI (undefined)

frsqrte **fr**D,**fr**B (Rc=0)
frsqrte. **fr**D,**fr**B (Rc=1)



This PowerPC instruction is not implemented by the 601. Execution of this instruction will invoke the illegal instruction handler. A description of the operation of this instruction is provided for emulation purposes.

A double-precision estimate of the reciprocal of the square root of the floating-point operand in register **fr**B is placed into register **fr**D. The estimate placed into register **fr**D is correct to a precision of one part in 32 of the reciprocal of the square root of **fr**B.

Operation with various special values of the operand is summarized below.

Operand	Result	Exception
-∞	QNaN**	VXSQRT
<0	QNaN**	VXSQRT
-0	-∞*	ZX
+0	+∞*	ZX
+∞	+0	None
SNaN	QNaN**	VXSNAN
QNaN	QNaN	None

* No result if FPSCR[ZE]=1.

** No result if FPSCR[VE]=1.

FPSCR[FPRF] is set to the class and sin of the result, except for invalid operation exceptions when FPSCR[VE] = 1 and zero divide exceptions when FPSCR[ZE] = 1.

Other registers altered:

- Condition Register (CR1 Field):

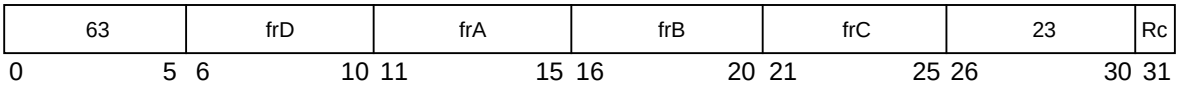
  Affected: FX, FEX, VX, OX (if Rc=1)
- Floating-point Status and Control Register:

  Affected: FX, OX, UX, ZX, VXSNAN, FPRF, FR (undefined), FI (undefined)

fselx — Not Implemented in 601 — fselx

Floating-Point Select — Floating-Point Unit

fsel **fr**D,**fr**A,**fr**C,**fr**B (Rc=0)
fsel. **fr**D,**fr**A,**fr**C,**fr**B (Rc=1)



if (**fr**A) ≥ 0.0 then **fr**D ← **fr**C
else **fr**D←(**fr**B)

This PowerPC instruction is not implemented by the 601. Execution of this instruction will invoke the illegal instruction handler. A description of the operation of this instruction is provided for emulation purposes.

The floating-point operand in register **fr**A is compared to the value zero. If the operand is greater than or equal to zero, register **fr**D is set to the contents of register **fr**C. If the operand is less than zero or is a NaN, register **fr**D is set to the contents of register **fr**B. The comparison ignores the sign of zero (i.e., regards +0 as equal to -0).

Other registers altered:

- Condition Register (CR1 Field):

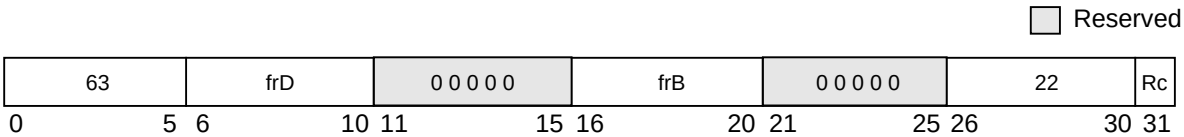
  Affected: FX, FEX, VX, OX (if Rc=1)

Care must be taken in using **fsel** if IEEE compatibility is required, or if the values being tested can be NaNs or infinities.

fsqrtx — Not Implemented in 601 — fsqrtx

Floating-Point Square Root — Floating-Point Unit

fsqrt **fr**D,**fr**B (Rc=0)
fsqrt. **fr**D,**fr**B (Rc=1)



This PowerPC instruction is not implemented by the 601. Execution of this instruction will invoke the illegal instruction handler. A description of the operation of this instruction is provided for emulation purposes.

The square root of the floating-point operand in register **fr**B is placed into register **fr**D.

If the most significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register **fr**D.

Operation with various special values of the operand is summarized below.

Operand	Result	Exception
-∞	QNaN*	VXSQRT
<0	QNaN*	VXSQRT
-0	-0	None
+∞	+∞	None
SNaN	QNaN*	VXSNAN
QNaN	QNaN	None

* No result if FPSCR[VE]=1.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1.

Other registers altered:

- Condition Register (CR1 Field):

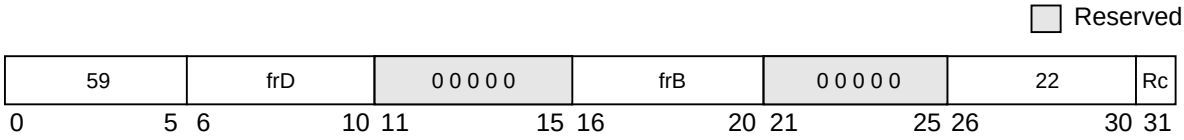
  Affected: FX, FEX, VX, OX (if Rc=1)
- Floating-point Status and Control Register:

  Affected: FX, XX, VXSQRT, VXSNAN, FPRF, FR, FI

fsqrt*x* Not Implemented in 601 **fsqrt***x*

Floating-Point Square Root Single-Precision — Floating-Point Unit

fsqrts **fr**D,**fr**B (Rc=0)
fsqrts. **fr**D,**fr**B (Rc=1)



This PowerPC instruction is not implemented by the 601. Execution of this instruction will invoke the illegal instruction handler. A description of the operation of this instruction is provided for emulation purposes.

The square root of the floating-point operand in register **fr**B is placed into register **fr**D.

If the most significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register **fr**D.

Operation with various special values of the operand is summarized below.

Operand	Result	Exception
-∞	QNaN*	VXSQRT
<0	QNaN*	VXSQRT
-0	-0	None
+∞	+∞	None
SNaN	QNaN*	VXSNAN
QNaN	QNaN	None

* No result if FPSCR[VE]=1.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1.

Other registers altered:

- Condition Register (CR1 Field):

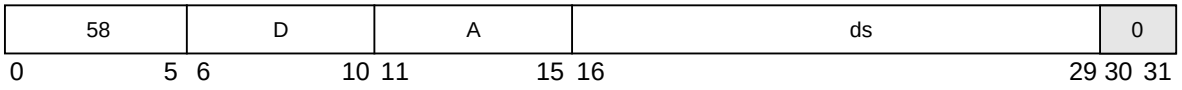
  Affected: FX, FEX, VX, OX (if Rc=1)
- Floating-point Status and Control Register:

  Affected: FX, XX, VXSQRT, VXSNAN, FPRF, FR, FI

ld — Not Implemented in 601 — ld

Load Double Word — Integer Unit

ld **r**D**,ds(r**A**)**



if **r**A=0 then b←0
else b←**r**A
EA←b+EXTS(ds||0b00)
rD←MEM(EA, 8)

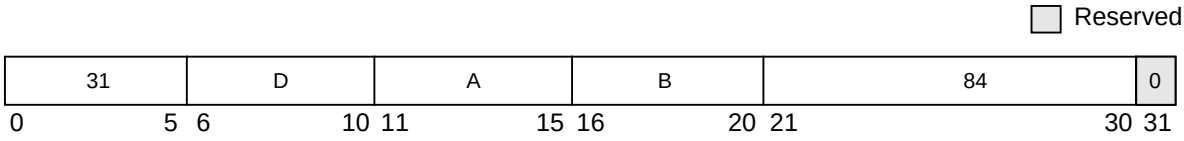
EA is the sum (**r**A|0)+(ds||0b00). The double word in memory addressed by EA is loaded into **r**D.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Other registers altered:

- None

ldarx **r**D,**r**A,**r**B



if **r**A=0 then b←0
else b←**r**A
EA←b+**r**B
RESERVE←1
RESERVE_ADDR←func(EA)
rD←MEM(EA, 8)

EA is the sum (**r**A|0)+(**r**B). The double word in memory addressed by EA is loaded into **r**D.

This instruction creates a reservation for use by a store double word conditional instruction. An address computed from the EA is associated with the reservation, and replaces any address previously associated with the reservation.

EA must be a multiple of 8. If it is not, the system alignment error handler may be invoked or the results may be boundedly undefined.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

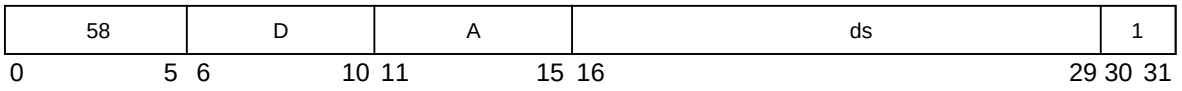
Other registers altered:

- None

ldu Not Implemented in 601 ldu

Load Double Word with Update — Integer Unit

ldu **r**D,ds(**r**A)



EA←**r**A+EXTS(ds||0b00)
rD←MEM(EA, 8)
rA←EA

EA is the sum (**r**A)+(ds||0b00). The double word in memory addressed by EA is loaded into **r**D.

EA is placed into **r**A.

If **r**A=0 or **r**A=**r**D, the instruction form is invalid.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

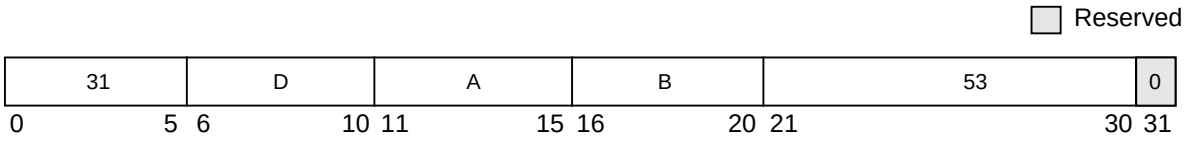
Other registers altered:

- None

ldux Not Implemented in 601 ldux

Load Double Word with Update Indexed — Integer Unit

ldux **r**D,**r**A,**r**B



EA←**r**A+ **r**B
rD←MEM(EA, 8)
rA←EA

EA is the sum (**r**A)+(**r**B). The double word in memory addressed by EA is loaded into **r**D.

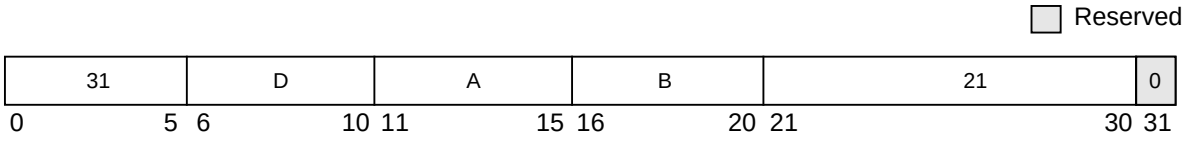
EA is placed into **r**A. If **r**A=0 or **r**A=**r**D, the instruction form is invalid.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction to be invoked.

ldx Not Implemented in 601 ldx

Load Double Word Indexed

ldx **r**D**,r**A**,r**B



if **r**A = 0 then b←0
else b←**r**A
EA←b+**r**B
rD←MEM(EA, 8)

EA is the sum (**r**A|0)+(**r**B). The double word in memory addressed by EA is loaded into **r**D.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

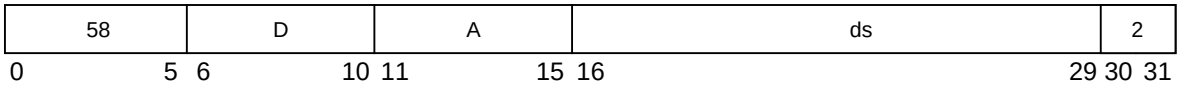
Other registers altered:

- None

lwa Not Implemented in 601 lwa

Load Word Algebraic Integer Unit

lwa **r**D**,ds(r**A**)**



if **r**A=0 then b←0
else b←**r**A
EA←b+EXTS(ds||0b00)
rD←EXTS(MEM(EA, 4))

EA is the sum (**r**A|0)+(ds||0b00). The word in memory addressed by EA is loaded into **r**D[32–63]. Register **r**D[0–31] are filled with a copy of bit 0 of the loaded word.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

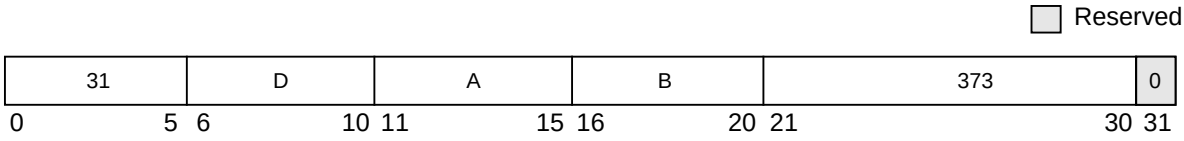
Other registers altered:

- None

lwaux Not Implemented in 601 lwaux

Load Word Algebraic with Update Indexed

lwaux **r**D,**r**A,**r**B



EA←**r**A+**r**B
rD←EXTS(MEM(EA, 4))
rA←EA

EA is the sum (**r**A)+(**r**B). The word in memory addressed by EA is loaded into **r**D[32–63]. Register **r**D[0–31] are filled with a copy of bit 0 of the loaded word. EA is placed into **r**A. If **r**A=0 or **r**A=**r**D, the instruction form is invalid.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

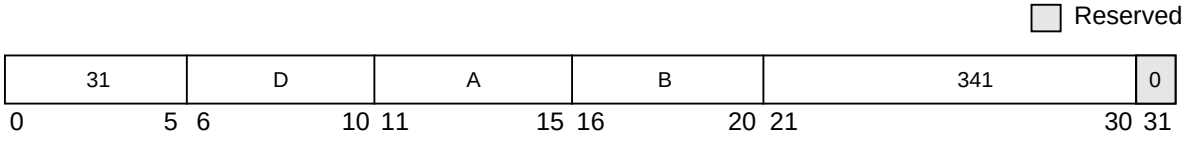
Other registers altered:

- None

lwax Not Implemented in 601 lwax

Load Word Algebraic Indexed

lwax **r**D,**r**A,**r**B



if **r**A=0 then b←0
else b←**r**A
EA←b + **r**B
rD←EXTS(MEM(EA, 4))

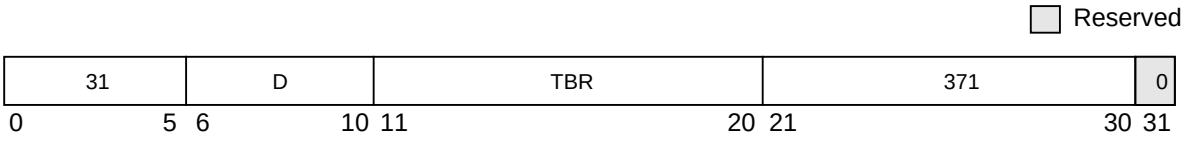
EA is the sum (**r**A|0)+(**r**B). The word in memory addressed by EA is loaded into **r**D[32–63]. Register **r**D[0–31] are filled with a copy of bit 0 of the loaded word.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Other registers altered:

- None

mftb **r**D,TBR




```
n←TBR[5–9] || TBR[0–4]
if n=268 then
  if (64-bit implementation) then
    rD←TB
  else
    rD ← TB[32–63]
else if n=269 then
  if (64-bit implementation) then
    rD←(32)0 || TB[0–31]
  else
    rD←TB[0–31]
```

The TBR field denotes either the time base or time base upper, encoded as shown in Table C-5. The contents of the designated register are copied to **r**D. When reading Time Base Upper on a 64-bit implementation, the high-order 32 bits of **r**D are set to zero.

Table C-5. TBR Encodings for mftb

Decimal	TBR*		Register Name	Access
	TBR[5–9]	TBR[0–4]		
268	01000	01100	TB	User
269	01000	01101	TBU	User

*Note that the order of the two 5-bit halves of the TBR number is reversed.

If the TBR field contains any value other than one of the values shown in Table C-5, the instruction form is invalid.

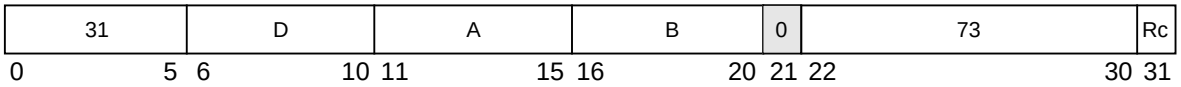
Other registers altered:

- None

mulhd Not Implemented in 601 mulhd

Multiply High Double Word — Integer Unit

mulhd **r**D,**r**A,**r**B (Rc=0)
mulhd. **r**D,**r**A,**r**B (Rc=1)



prod[0–127]←**r**A*rB
rD←prod[0–63]

The 64-bit multiplicands are **r**A and **r**B. The high-order 64 bits of the 128-bit product of the multiplicands are placed into **r**D.

Both the multiplicands and the product are interpreted as signed integers.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

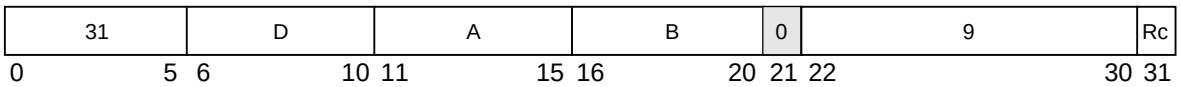
Other registers altered:

- Condition Register (CR0 Field):
  Affected: LT, GT, EQ, SO (if Rc=1)

mulhdu Not Implemented in 601 mulhdu

Multiply High Double Word Unsigned — Integer Unit

mulhdu **r**D,**r**A,**r**B (Rc=0)
mulhdu. **r**D,**r**A,**r**B (Rc=1)



prod[0–127]←**r**A*rB
rD←prod[0–63]

The 64-bit multiplicands are **r**A and **r**B. The high-order 64 bits of the 128-bit product of the multiplicands are placed into **r**D.

Both the multiplicands and the product are interpreted as unsigned integers.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Other registers altered:

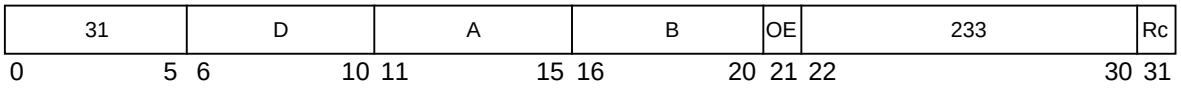
- Condition Register (CR0 Field):

  Affected: LT, GT, EQ, SO (if Rc=1)

mulld Not Implemented in 601 mulld

Multiply Low Double Word — Integer Unit

mulld	**r**D,**r**A,**r**B	(OE=0 Rc=0)
mulld.	**r**D,**r**A,**r**B	(OE=0 Rc=1)
mulldo	**r**D,**r**A,**r**B	(OE=1 Rc=0)
mulldo.	**r**D,**r**A,**r**B	(OE=1 Rc=1)



prod[0–127]←**r**A*rB
rD←prod[64–127]

The 64-bit operands are **r**A and **r**B. The low-order 64 bits of the 128-bit product of the operands are placed into **r**D.

If OE=1, then SO and OV are set to one if the product cannot be represented in 64 bits.

Both the operands and the product are interpreted as signed integers. However, the result in **r**D is independent of whether the operands are interpreted as signed or unsigned integers.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Other registers altered:

- Condition Register (CR0 Field):

  Affected: LT, GT, EQ, SO (if Rc=1)

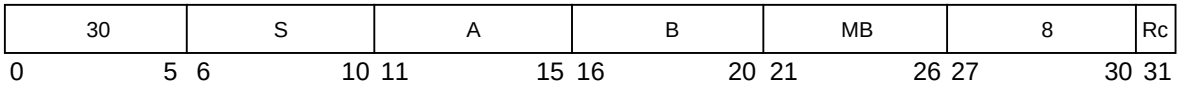
- Exception Register:

  Affected: SO OV (if OE=1)

rldcl Not Implemented in 601 rldcl

Rotate Left Double Word then Clear Left — Integer Unit

rldcl **r**A,**r**S,**r**B,MB (Rc=0)
rldcl. **r**A,**r**S,**r**B,MB (Rc=1)



n←**r**B[58–63]
r←ROTL[64](**r**S, *n*)
b←MB[5] || MB[0–4]
m←MASK(b, 63)
rA←r & m

The contents of **r**S are rotated[64] left the number of bits specified by **r**B[58–63]. A mask is generated having 1-bits from bit MB through bit 63 and 0-bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into **r**A.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

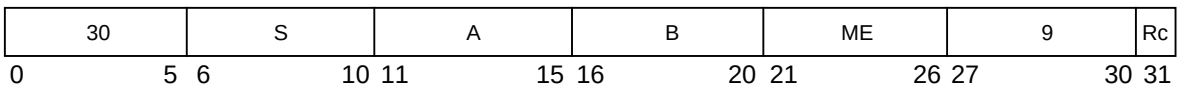
Other registers altered:

- Condition Register (CR0 Field):
  Affected: LT, GT, EQ, SO (if Rc=1)

rldcr Not Implemented in 601 rldcr

Rotate Left Double Word then Clear Right — Integer Unit

rldcr **r**A,**r**S,**r**B,ME (Rc=0)
rldcr. **r**A,**r**S,**r**B,ME (Rc=1)



n←**r**B[58–63]
r←ROTL[64](**r**S, *n*)
e←ME[5] || ME[0–4]
m←MASK(0, e)
rA←r & m

The contents of **r**S are rotated[64] left the number of bits specified by **r**B[58–63]. A mask is generated having 1-bits from bit 0 through bit ME and 0-bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into **r**A.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Other registers altered:

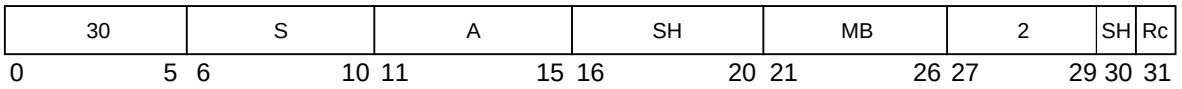
- Condition Register (CR0 Field):

  Affected: LT, GT, EQ, SO (if Rc=1)

rldic — Not Implemented in 601 — rldic

Rotate Left Double Word Immediate then Clear — Integer Unit

rldic **r**A,**r**S,SH,MB (Rc=0)
rldic. **r**A,**r**S,SH,MB (Rc=1)



n←SH[5] || SH[0–4]
r←ROTL[64](**r**S, *n*)
b←MB[5] || MB[0–4]
m←MASK(b, ¬ *n*)
rA←r & m

The contents of **r**S are rotated[64] left SH bits. A mask is generated having 1-bits from bit MB through bit 63-SH and 0-bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into **r**A.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Other registers altered:

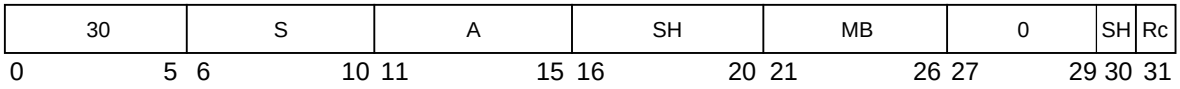
- Condition Register (CR0 Field):

  Affected: LT, GT, EQ, SO (if Rc=1)

rldicl — Not Implemented in 601 — rldicl

Rotate Left Double Word Immediate then Clear Left — Integer Unit

rldicl **r**A,**r**S,SH,MB (Rc=0)
rldicl. **r**A,**r**S,SH,MB (Rc=1)



n←SH[5] || SH[0–4]
r←ROTL[64](**r**S, *n*)
b←MB[5] || MB[0–4]
m←MASK(b, 63)
rA←r & m

The contents of **r**S are rotated[64] left SH bits. A mask is generated having 1-bits from bit MB through bit 63 and 0-bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into **r**A.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Other registers altered:

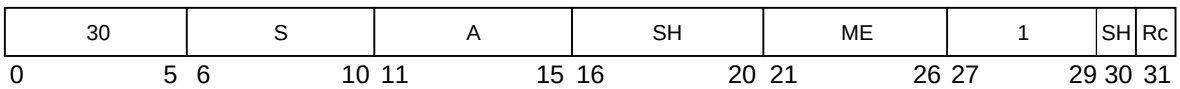
- Condition Register (CR0 Field):

  Affected: LT, GT, EQ, SO (if Rc=1)

rldicr — Not Implemented in 601 — rldicr

Rotate Left Double Word Immediate then Clear Right — Integer Unit

rldicr **r**A,**r**S,SH,ME (Rc=0)
rldicr. **r**A,**r**S,SH,ME (Rc=1)



n←SH[5] || SH[0–4]
r←ROTL[64](**r**S, *n*)
e←ME[5] || ME[0–4]
m←MASK(0, e)
rA←r & m

The contents of **r**S are rotated[64] left SH bits. A mask is generated having 1-bits from bit ME and 0-bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into **r**A.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Other registers altered:

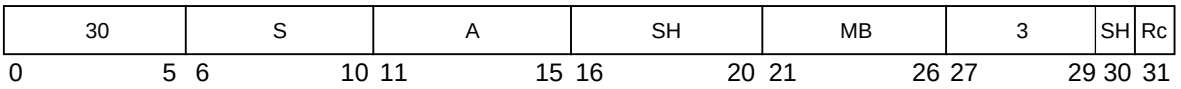
- Condition Register (CR0 Field):

  Affected: LT, GT, EQ, SO (if Rc=1)

rldimi — Not Implemented in 601 — rldimi

Rotate Left Double Word Immediate then Clear Left — Integer Unit

rldimi **r**A,**r**S,SH,MB (Rc=0)
rldimi. **r**A,**r**S,SH,MB (Rc=1)



n←SH[5] || SH[0–4]
r←ROTL[64](**r**S, *n*)
b←MB[5] || MB[0–4]
m←MASK(b, ¬ *n*)
rA←(r & m) | (**r**A & ¬ m)

The contents of **r**S are rotated[64] left SH bits. A mask is generated having 1-bits from bit MB through bit 63-SH and 0-bits elsewhere. The rotated data is inserted into **r**A under control of the generated mask.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Other registers altered:

- Condition Register (CR0 Field):

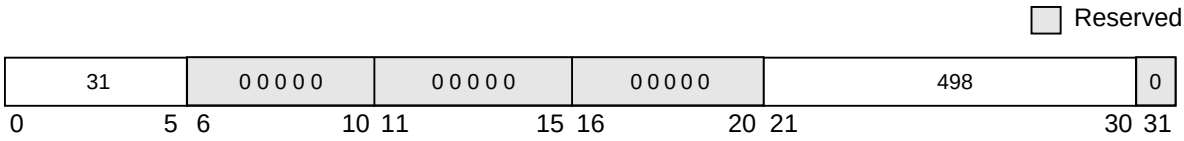
  Affected: LT, GT, EQ, SO (if Rc=1)

slbia

Not Implemented in 601

slbia

SLB Invalidate All



All SLB entries←invalid

This PowerPC instruction is not implemented by the 601. Execution of this instruction will invoke the illegal instruction handler. A description of the operation of this instruction is provided for emulation purposes.

The SLB is invalidated regardless of the settings of MSR[IT] and MSR[DT].

This instruction is supervisor-level.

This instruction is optional in PowerPC architecture.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause an illegal instruction type program exception.

It is not necessary that the ASR point to a valid segment table when issuing **slbia**.

Other registers altered:

- None

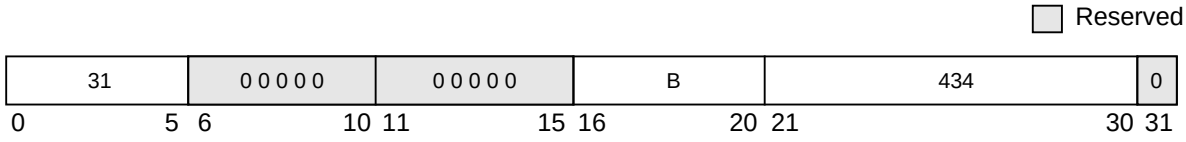
slbie

Not Implemented in 601

slbie

SLB Invalidate Entry

slbie **r**B



EA←(**r**B)
if SLB entry exists for EA, then
  SLB entry←invalid

This PowerPC instruction is not implemented by the 601. Execution of this instruction will invoke the illegal instruction handler. A description of the operation of this instruction is provided for emulation purposes.

EA is the contents of **r**B. If the segment lookaside buffer (SLB) contains an entry corresponding to EA, that entry is made invalid (i.e., removed from the SLB).

The SLB search is done regardless of the settings of MSR[IT] and MSR[DT].

Block address translation for EA, if any, is ignored.

This instruction is supervisor-level.

This instruction is optional in PowerPC architecture.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause an illegal instruction type program exception.

Other registers altered:

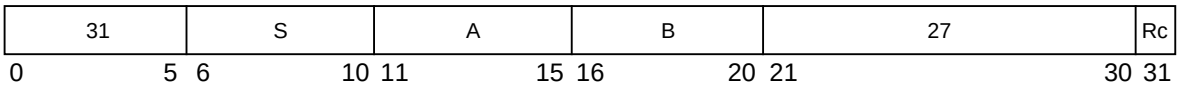
None

It is not necessary that the ASR point to a valid segment table when issuing **slbie**.

sld — Not Implemented in 601 — sld

Shift Left Double Word — Integer Unit

sld **r**A,**r**S,**r**B (Rc=0)
sld. **r**A,**r**S,**r**B (Rc=1)



n←**r**B[58–63]
r←ROTL[64](**r**S, *n*)
if **r**B[57]=0 then
    m←MASK(0, 63-*n*)
else m←(64)0
rA←r & m

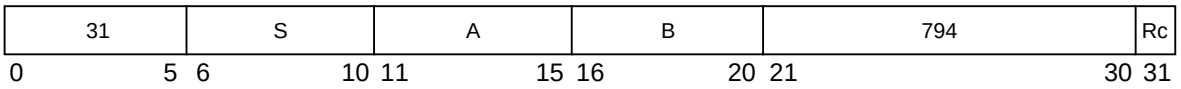
The contents of **r**S are shifted left the number of bits specified by **r**B[57–63]. Bits shifted out of position 0 are lost. Zeros are supplied to the vacated positions on the right. The result is placed into **r**A. Shift amounts from 64 to 127 give a zero result.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Other registers altered:

- Condition Register (CR0 Field):
  Affected: LT, GT, EQ, SO (if Rc=1)

srad **r**A,**r**S,**r**B (Rc=0)
srad. **r**A,**r**S,**r**B (Rc=1)



n←**r**B[58–63]
r←ROTL[64](**r**S, 64-*n*)
if **r**B[57]=0 then
  m←MASK(*n*, 63)
else m←(64)0
S←**r**S[0]
rA←(r & m) | (((64)S) & ¬ m)
CA←S & ((r& ¬ m)≠0)

The contents of **r**S are shifted right the number of bits specified by **r**B[57–63]. Bits shifted out of position 63 are lost. Bit 0 of **r**S is replicated to fill the vacated positions on the left. The result is placed into **r**A. CA is set to 1 if **r**S is negative and any 1-bits are shifted out of position 63; otherwise CA is set to 0. A shift amount of zero causes **r**A to be set equal to **r**S, and CA to be set to 0. Shift amounts from 64 to 127 give a result of 64 sign bits in **r**A, and cause CA to receive the sign bit of **r**S.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Other registers altered:

- Condition Register (CR0 Field):

  Affected: LT, GT, EQ, SO (if Rc=1)

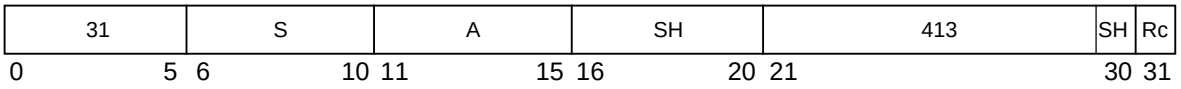
- Exception Register:

  Affected: CA

sradi — Not Implemented in 601 — sradi

Shift Right Algebraic Double Word Immediate — Integer Unit

sradi **r**A,**r**S,SH (Rc=0)
sradi. **r**A,**r**S,SH (Rc=1)



n←SH[5] || SH[0–4]
r←ROTL[64](**r**S, 64-*n*)
m←MASK(*n*, 63)
S←**r**S[0]
rA←(r & m) | (((64)S) & ¬ m)
CA←S & ((r& ¬ m)≠0)

The contents of **r**S are shifted right SH bits. Bits shifted out of position 63 are lost. Bit 0 of **r**S is replicated to fill the vacated positions on the left. The result is placed into **r**A. CA is set to 1 if **r**S is negative and any 1-bits are shifted out of position 63; otherwise CA is set to 0. A shift amount of zero causes **r**A to be set equal to **r**S, and CA to be set to 0.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Other registers altered:

- Condition Register (CR0 Field):

  Affected: LT, GT, EQ, SO (if Rc=1)

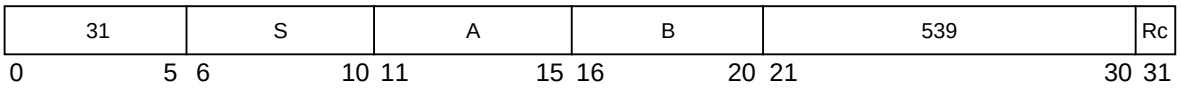
- Exception Register:

  Affected: CA

srd — Not Implemented in 601 — srd

Shift Right Double Word — Integer Unit

srd **r**A,**r**S,**r**B (Rc=0)
srd. **r**A,**r**S,**r**B (Rc=1)



n←**r**B[58–63]
r←ROTL[64](**r**S, 64-*n*)
if **r**B[57]=0 then
  m←MASK(*n*, 63)
else m←(64)0
rA←r & m

The contents of **r**S are shifted right the number of bits specified by **r**B[57–63]. Bits shifted out of position 63 are lost. Zeros are supplied to the vacated positions on the left. The result is placed into **r**A. Shift amounts from 64 to 127 give a zero result.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

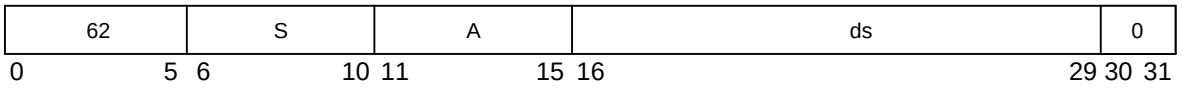
Other registers altered:

- Condition Register (CR0 Field):
  Affected: LT, GT, EQ, SO (if Rc=1)

std — Not Implemented in 601 — std

Store Double Word — Integer Unit

std **r**S,**ds(r**A**)**



if **r**A=0 then b←0
else b←**r**A
EA←b + EXTS(ds||0b00)
(MEM(EA, 8))←**r**S

EA is the sum (**r**A|0)+(ds||0b00). Register **r**S is stored into the double word in memory addressed by EA.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

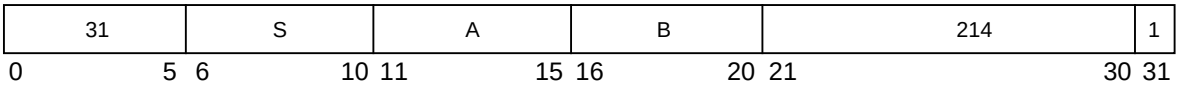
Other registers altered:

- None

stdcx. Not Implemented in 601 stdcx.

Store Double Word Conditional Indexed — Integer Unit

stdcx. **r**S,**r**A,**r**B




```
if rA=0 then b←0
else        b←rA
EA←b + rB
if RESERVE then
  (MEM(EA, 8))←rS
  RESERVE←0
CR0←0b00 || 0b1 || XER[SO]
else
  CR0←0b00 || 0b0 || XER[SO]
```

EA is the sum (**r**A|0)+(**r**B).

If a reservation exists, **r**S is stored into the double word in memory addressed by EA and the reservation is cleared.

If a reservation does not exist, the instruction completes without altering memory.

CR0 Field is set to reflect whether the store operation was performed (i.e., whether a reservation existed when the **stdcx.** instruction commenced execution), as follows:

CR0[LT GT EQ SO] + 0b00 || store_performed || XER[SO]

EA must be a multiple of 8. If it is not, the system alignment error handler may be invoked or the results may be boundedly undefined.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Other registers altered:

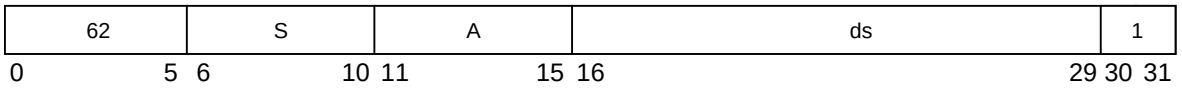
- Condition Register (CR0 Field):

  Affected: LT, GT, EQ, SO

stdu Not Implemented in 601 stdu

Store Double Word with Update — Integer Unit

stdu **r**S**,ds(r**A**)**



EA←**r**A+EXTS(ds||0b00)
(MEM(EA, 8))←**r**S
rA←EA

EA is the sum (**r**A)+(ds||0b00). Register **r**S is stored into the double word in memory addressed by EA.

EA is placed into **r**A.

If **r**A=0, the instruction form is invalid.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

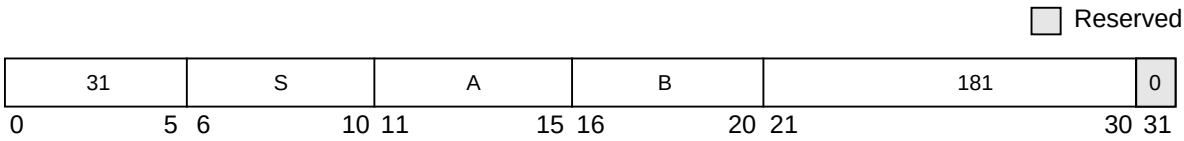
Other registers altered:

- None

stdux Not Implemented in 601 stdux

Store Double Word with Update Indexed — Integer Unit

stdux **r**S**,r**A**,r**B



EA←**r**A + **r**B
MEM(EA, 8)←**r**S
rA←EA

EA is the sum (**r**A)+(**r**B). Register **r**S is stored into the double word in memory addressed by EA.

EA is placed into **r**A.

If **r**A=0, the instruction form is invalid.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

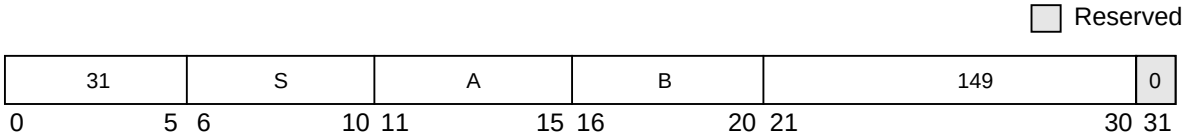
Other registers altered:

None

stdx — Not Implemented in 601 — stdx

Store Double Word Indexed — Integer Unit

stdx **r**S,**r**A,**r**B



if **r**A=0 then b←0
else b←**r**A
EA←b+**r**B
(MEM(EA, 8))←**r**S

EA is the sum (**r**A|0)+(**r**B). Register **r**S is stored into the double word in memory addressed by EA.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

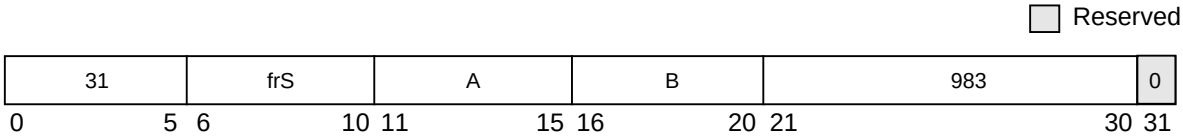
Other registers altered:

None

stfiwx — Not Implemented in 601 — stfiwx

Store Floating-Point as Integer Word — Floating-Point Unit

stfiwx **fr**S,**r**A,**r**B



if **r**A = 0 then b←0
else b←**r**A
EA←b+**r**B
MEM(EA, 4)←**fr**S[32–63]

EA is the sum (**r**A|0)+(**r**B).

The contents for the low-order 32 bits of register **fr**S are stored, without conversion, into the word in memory addressed by EA.

Other registers altered:

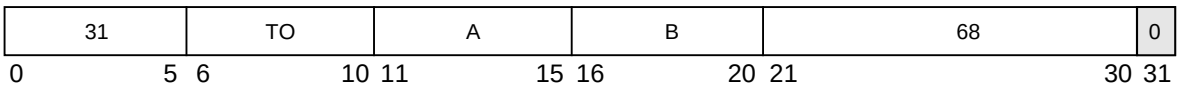
- None

td **Not Implemented in 601** **td**

Trap Double Word Integer Unit

td TO**,r**A**,r**B

Reserved



a←**r**A
b←**r**B
if (a < b) & TO[0] then TRAP
if (a > b) & TO[1] then TRAP
if (a = b) & TO[2] then TRAP
if (a <U b) & TO[3] then TRAP
if (a >U b) & TO[4] then TRAP

The contents of **r**A is compared with the contents of **r**B. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

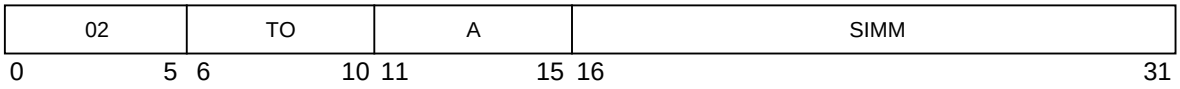
Other registers altered:

- None

tdi — Not Implemented in 601 — tdi

Trap Double Word Immediate — Integer Unit

tdi TO,**r**A,SIMM



a←**r**A
if (a < EXTS(SIMM)) & TO[0] then TRAP
if (a > EXTS(SIMM)) & TO[1] then TRAP
if (a = EXTS(SIMM)) & TO[2] then TRAP
if (a <U EXTS(SIMM)) & TO[3] then TRAP
if (a >U EXTS(SIMM)) & TO[4] then TRAP

The contents of **r**A are compared with the sign-extended SIMM field. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

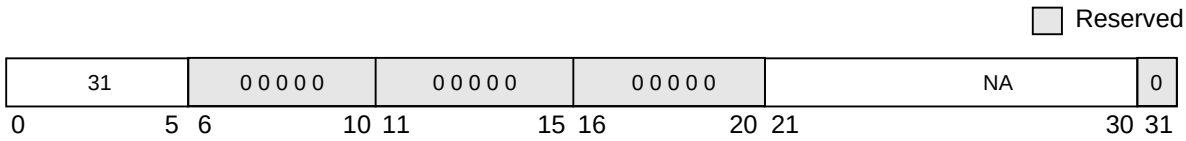
This instruction is defined only for 64-bit implementations. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

Other registers altered:

- None

tlbia — Not Implemented in 601 — tlbia

Translation Lookaside Buffer Invalidate All — Integer Unit



All TLB entries ← invalid

This PowerPC instruction is not implemented by the 601. Execution of this instruction will invoke the illegal instruction handler. A description of the operation of this instruction is provided for emulation purposes.

The entire TLB is invalidated (that is, all entries are removed).

The TLB is invalidated regardless of the settings of MSR[IT] and MSR[DT].

This is a supervisor-level instruction.

This instruction is optional in PowerPC architecture.

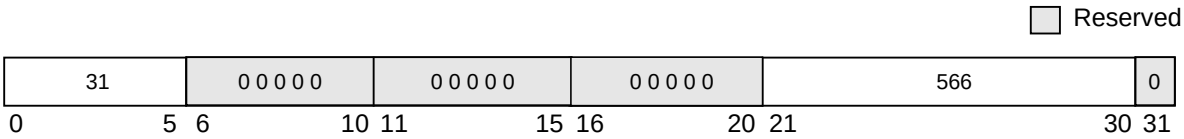
Other registers altered:

- None

It is not necessary that the ASR point to a valid segment table when issuing **tlbia**.

tlbsync — Not Implemented in 601 — tlbsync

TLB Synchronize



The **tlbsync** instruction waits until all previous **tlbie**, **tlbiex**, and **tlbia** instructions executed by the processor executing this instruction have been received and completed by all other processors.

This instruction is supervisor-level.

This instruction is optional in PowerPC architecture, but it must be implemented if any of the following are true:

- A TLB invalidation instruction that broadcasts is implemented.
- The **eciwx** or **ecowx** instructions are implemented.

Other registers altered:

- None

Appendix D
Classes of Instructions

This appendix describes how the classes of PowerPC instructions are defined. The three classifications are as follows:

- Defined
- Illegal
- Reserved

Note that while the definitions of these terms are consistent among the PowerPC processors, the assignment of these classifications is not. For example, an instruction that is specific to 64-bit implementations is considered defined for 64-bit implementations, but illegal for 32-bit implementations such as the PowerPC 601 microprocessor.

D.1 Classes of Instructions

The 601 is a 32-bit implementation of the PowerPC architecture with differences and redefinitions noted throughout this document. Differences stem largely from the different address bus sizes and compliance with POWER architecture.

All 601 instructions belong to one of the following three classes:

- Defined
- Illegal
- Reserved

The class is determined by examining the opcode and the extended opcode, if any. If the opcode, or combination of opcode and extended opcode, is not that of a defined instruction nor of a reserved instruction, the instruction is illegal.

In future versions of the PowerPC architecture, instructions that are now illegal may become defined (by being added to the architecture) or reserved (by being assigned to one of the special purposes). Likewise, reserved instructions may become defined.

D.1.1 Defined Instruction Class

Defined instructions are guaranteed to be supported in all PowerPC implementations, except as stated in the instruction descriptions in Chapter 10, “Instruction Set.” The 601

provides hardware support for most of the instructions defined for 32-bit implementations; it does not provide direct hardware support for the instructions listed in Appendix C, “PowerPC Instructions Not Implemented.”

The 601 invokes the system illegal instruction error handler (part of the program exception) when the unimplemented PowerPC instructions are encountered so they may be emulated in software, as required.

A defined instruction can have invalid forms, as described in Section D.1.1.1, “Invalid Instruction Forms.”

D.1.1.1 Invalid Instruction Forms

An instruction form is invalid if one or more operands, excluding opcodes, are coded incorrectly. Attempting to execute an invalid form of an instruction either invokes the system illegal instruction error handler (a program exception) or yields undefined results. See Chapter 10, “Instruction Set,” for individual instruction descriptions.

Invalid forms result when a bit or operand is coded incorrectly, for example, or when a reserved bit is shown as “0” but is coded as a “1”. The following instructions have invalid forms identified in their individual instruction descriptions:

- Branch conditional instructions
- Load/store with update instructions
- Load multiple instructions
- Load string instructions
- Move to/from special purpose register (**mtspr**, **mfspr**)
- Load/store floating-point with update instructions

In some cases, an invalid form of a PowerPC instruction is not an invalid form for the corresponding POWER instruction. As a result, to maintain compatibility with POWER applications, the 601 often handles PowerPC invalid forms as described in the POWER architecture. In other cases, the 601 handles the invalid form in the manner that is most convenient for that particular case. Each of the PowerPC invalid forms are addressed in this document, and a description of how 601 handles each case is provided.

D.1.2 Illegal Instruction Class

Illegal instructions can be grouped into the following categories:

- Instructions that are not implemented in the PowerPC architecture. These opcodes are available for future extensions of the PowerPC architecture; that is, future versions of the PowerPC architecture may define any of these instructions to perform new functions. The following primary opcodes are illegal:

  1, 4, 5, 6, 56, 57, 60, 61

- Instructions that are implemented in the PowerPC architecture but are not implemented in a specific PowerPC implementation (for example, instructions that can be executed on 64-bit PowerPC processors are considered illegal for 32-bit processors.

  The following opcodes are defined for 64-bit implementations only and are illegal on the 601:

  2, 30, 58, 62

- The following primary opcodes have unused extended opcodes. Their unused extended opcodes can be determined from information in Section A.2, “Complete Instruction List Sorted by Opcode,” and Section D.1.3, “Reserved Instructions.” Notice that extended opcodes for instructions that are defined only for 64-bit implementations are illegal in 32-bit implementations. All unused extended opcodes are illegal.

  19, 31, 59, 63 (opcodes 30 and 62 are illegal for all 32-bit implementations, but as 64-bit opcodes they have some unused extended opcodes).

An attempt to execute an illegal instruction invokes the illegal instruction error handler (a program exception) but has no other effect. See Section 5.4.7, “Program Exception (x'00700'),” for additional information about illegal and invalid instruction exceptions.

Note that an instruction consisting entirely of binary zeros is guaranteed to be an illegal instruction. This increases the probability that an attempt to execute data or uninitialized memory invokes the system illegal instruction error handler (a program exception). Note that if only the primary opcode consists of all zeros, the instruction is considered a reserved instruction, as described in Section D.1.3, “Reserved Instructions.”

With the exception of the instruction consisting entirely of binary zeros, the illegal instructions are available for further additions to the PowerPC architecture.

D.1.3 Reserved Instructions

Reserved instructions are allocated to specific purposes outside the scope of the PowerPC architecture. An attempt to execute a reserved instruction either causes a program exception or yields undefined results.

An attempt to execute a reserved instruction invokes the illegal instruction error handler (a program exception); however, the 601 executes many POWER architecture instructions that otherwise are not part of the PowerPC architecture. See Section 5.4.7, “Program Exception (x'00700'),” for additional information about illegal and invalid instruction exceptions.

The instructions in this class are allocated to specific purposes that are outside the scope of the PowerPC user instruction set architecture, PowerPC virtual environment architecture, and PowerPC operating environment architecture.

The following types of instructions are included in this class:

1. Instructions for the POWER architecture that have not been included in the PowerPC user instruction set architecture
2. Implementation-specific instructions used to conform to the PowerPC architecture specifications
3. The instruction with primary opcode 0, when the instruction does not consist entirely of binary zeros
4. Any other implementation-specific instructions that are not defined in the PowerPC user instruction set architecture, PowerPC virtual environment architecture, or the PowerPC operating environment architecture

Appendix E
Multiple-Precision Shifts

This appendix gives examples of how multiple precision shifts can be programmed. A multiple-precision shift is initially defined to be a shift of an *n*-word quantity, where *n*>1. The quantity to be shifted is contained in *n* registers. The shift amount is specified either by an immediate value in the instruction or by bits 27–31 of a register.

The examples shown below distinguish between the cases $n = 2$ and $n > 2$. If $n = 2$, the shift amount may be in the range 0–63, which are the maximum ranges supported by the shift instructions used. However if $n > 2$, the shift amount must be in the range 0–31, for the examples to yield the desired result. The specific instance shown for $n > 2$ is $n = 3$: extending those instruction sequences to larger *n* is straightforward, as is reducing them to the case $n = 2$ when the more stringent restriction on shift amount is met. For shifts with immediate shift amounts only the case $n = 3$ is shown, because the more stringent restriction on shift amount is always met.

In the examples it is assumed that GPRs 2 and 3 (and 4) contain the quantity to be shifted, and that the result is to be placed into the same registers. In all cases, for both input and result, the lowest-numbered register contains the highest-order part of the data and highest-numbered register contains the lowest-order part. For nonimmediate shifts, the shift amount is assumed to be in bits 27–31 (32-bit mode) of GPR6. For immediate shifts, the shift amount is assumed to be greater than 0. GPRs 0–31 are used as scratch registers. For $n > 2$, the number of instructions required is 2N – 1 (immediate shifts) or 3N – 1 (nonimmediate shifts).

E.1 Multiple-Precision Shift Examples

The examples shown here are for 32-bit mode, but they work both in 32-bit mode of a 64-bit implementation and in a 32-bit implementation. They perform the shift in units of words. If the ability to run in 32-bit implementations is not required, in a 64-bit implementation better performance can be obtained in 32-bit mode than that of the examples shown above, by using all 64 bits of GPRs 2 and 3 (and 4) to contain the quantity to be shifted, and placing the result into all 64 bits of the same registers.

Let *n* be the number of words to be shifted.

Shift Left Immediate, *n* = 3 (Shift Amount < 32)

```
rlwinm r2,r2,SH,0,31-SH
rlwimi r2,r3,SH,32-SH,31
rlwinm r3,r3,SH,0,31-SH
rlwimi r3,r4,SH,32-SH,31
rlwinm r4,r4,SH,0,31-SH
```

Shift Left, *n* = 2 (Shift Amount < 64)

```
subfic r31,r6,32
slw    r2,r2,r6
srw    r0,r3,r31
or     r2,r2,r0
addic  r31,r6,-32
slw    r0,r3,r31
or     r2,r2,r0
slw    r3,r3,r6
```

Shift Left, *n* = 3 (Shift Amount < 32)

```
subfic r31,r6,32
slw    r2,r2,r6
srw    r0,r3,r31
or     r2,r2,r0
slw    r3,r3,r6
srw    r0,r4,r31
or     r3,r3,r0
slw    r4,r4,r6
```

Shift Right Immediate, *n* = 3 (Shift Amount < 32)

```
rlwinm r4,r4,32-SH,SH,31
rlwimi r4,r3,32-SH,0,SH-1
rlwinm r3,r3,32-SH,SH,31
rlwimi r3,r2,32-SH,0,SH-1
rlwinm r2,r2,32-SH,SH,31
```

Shift Right, *n* = 2 (Shift Amount < 64)

```
subfic r31,r6,32
srw    r3,r3,r6
slw    r0,r2,r31
or     r3,r3,r0
addic  r31,r6,-32
srw    r0,r2,r31
or     r3,r3,r0
srw    r2,r2,r6
```

Shift Right, *n* = 3 (Shift Amount < 32)

```
subfic r31,r6,32
srw    r4,r4,r6
slw    r0,r3,r31
or     r4,r4,r0
srw    r3,r3,r6
slw    r0,r2,r31
or     r3,r3,r0
srw    r2,r2,r6
```

Shift Right Algebraic Immediate, *n* = 3 (Shift Amount < 32)

```
rlwinm r4,r4,32-SH,SH,31
rlwimi r4,r3,32-SH,0,SH-1
rlwinm r3,r3,32-SH,SH,31
rlwimi r3,r2,32-SH,0,SH-1
srawi  r2,r2,SH
```

Shift Right Algebraic, *n* = 2 (Shift Amount < 64)

```
subfic r31,r6,32
srw    r3,r3,r6
slw    r0,r2,r31
or     r3,r3,r0
addic. r31,r6,-32
sraw   r0,r2,r31
ble    $+8
ori    r3,r0,0
sraw   r2,r2,r6
```

Shift Right Algebraic, *n* = 3 (Shift Amount < 32)

```
subfic r31,r6,32
srw    r4,r4,r6
slw    r0,r3,r31
or     r4,r4,r0
srw    r3,r3,r6
slw    r0,r2,r31
or     r3,r3,r0
sraw   r2,r2,r6
```

Appendix F
Floating-Point Models

This appendix gives examples of how the floating-point conversion instructions can be used to perform various conversions.

F.1 Conversion from Floating-Point Number to Signed Fixed-Point Integer Word

The full convert to signed fixed-point integer word function can be implemented with the sequence shown below, assuming that the floating-point value to be converted is in FPR1, the result is returned in GPR3, and a double word at displacement “disp” from the address in GPR1 can be used as scratch space.

```
fctiw[z]  f2,f1           #convert to fx int
stfd      f2,disp(r1)     #store float
lwz       r3,disp+4(r1)   #load word and zero
```

F.2 Conversion from Floating-Point Number to Unsigned Fixed-Point Integer Word

The full convert to unsigned fixed-point integer word function can be implemented with the sequence shown below, assuming that the floating-point value to be converted is in FPR1, the value 0 is in FPR0, the value 2^{32} is in FPR3, the value x'0000 0000 7FFF FFFF' is in FPR4, the value 2^{31} is in FPR5 and GPR5, the result is returned in GPR3, and a double word at displacement “disp” from the address in GPR1 can be used as scratch space.

```
fsel      f2,f1,f1,f0     #use 0 if < 0
fsub      f5,f3,f1        #use max if > max
fsel      f2,f5,f2,f3
fsub      f5,f2,f4        #subtract 2**31
fcmpu     cr2,f2,f4       #use diff if ≥ 2**31
fsel      f2,f5,f5,f2
fctiw[z]  f2,f2           #convert to fx int
stfd      f2,disp(r1)     #store float
lwz       r3,disp+4(r1)   #load word
blt       cr2,$+8         #add 2**31 if input
add       r3,r3,r4        #was ≥ 2**31
```

F.3 Floating-Point Models

This section describes models for floating-point instructions.

F.3.1 Floating-Point Round to Single-Precision Model

The following algorithm describes the operation of the Floating-Point Round to Single-Precision (**frsp**) instruction.

```
If FRB[1–11]<897 and FRB[1–63]>0 then
   Do
      If FPSCR[UE]=0 then goto Disabled Exponent Underflow
      If FPSCR[UE]=1 then goto Enabled Exponent Underflow
   End

If FRB[1–11]>1150 and FRB[1–11]<2047 then
   Do
      If FPSCR[OE]=0 then goto Disabled Exponent Overflow
      If FPSCR[OE]=1 then goto Enabled Exponent Overflow
   End

If FRB[1–11]>896 and FRB[1–11]<1151 then goto Normal Operand

If FRB[1–63]=0 then goto Zero Operand

If FRB[1–11]=2047 then
   Do
      If FRB[12–63]=0 then goto Infinity Operand
      If FRB[12]=1 then goto QNaN Operand
      If FRB[12]=0 and FRB[13–63]>0 then goto SNaN Operand
   End
```

Disabled Exponent Underflow:

```
sign ← FRB0
If FRB[1–11]=0 then
   Do
      exp ← -1022
      frac ← b'0' || FRB[12–63]
   End
If FRB[1–11]>0 then
   Do
      exp ← FRB[1–11] - 1023
      frac ← b'1' || FRB[12–63]
   End
Denormalize operand:
   G || R || X ←b'000'
   Do while exp<-126
      exp ← exp + 1
      frac || G || R || X ← b'0' || frac || G || (R | X)
   End
FPSCR[UX] ← frac[24–52] || G || R || X>0
If frac[24–52] || G || R || X>0 then FPSCR[XX] ←1
Round single(sign,exp,frac,G,R,X)
If frac=0 then
   Do
      FRT00 ← sign
      FRT0[1–63] ← 0
      If sign=0 then FPSCR[FPRF] ← "+zero"
      If sign=1 then FPSCR[FPRF] ← "-zero"
```

```
   End
If frac>0 then
   Do
      If frac[0]=1 then
         Do
            If sign=0 then FPSCR[FPRF] ← "+normal number"
            If sign=1 then FPSCR[FPRF] ← "-normal number"
         End
      If frac[0]=0 then
         Do
            If sign=0 then FPSCR[FPRF] ← "+denormalized number"
            If sign=1 then FPSCR[FPRF] ← "-denormalized number"
         End
      Normalize operand–
         Do while frac[0]=0
            exp ← exp-1
            frac || G || R ← frac[1–52] || G || R || b'0'
         End
      FRT[0] ← sign
      FRT[1–11] ← exp + 1023
      FRT[12–63] ← frac[1–23] || b'0 0000 0000 0000 0000 0000 0000 0000'
   End
Done
```

Enabled Exponent Underflow

```
FPSCR[UX] ← 1
sign ← FRB[0]
If FRB[1–11]=0 then
   Do
      exp ← -1022
      frac ← b'0' || FRB[12–63]
   End
If FRB[1–11]>0 then
   Do
      exp ← FRB[1–11] - 1023
      frac ← b'1' || FRB[12–63]
   End
Normalize operand–
   Do while frac[0]=0
      exp ← exp - 1
      frac ← frac[1–52] || b'0'
   End
If frac[24–52]>0 then FPSCR[XX] ← 1
Round single(sign,exp,frac,0,0,0)
exp ← exp + 192
FRT[0] ← sign
FRT[1–11] ← exp + 1023
FRT[12–63] ← frac1–23 || b'0 0000 0000 0000 0000 0000 0000 0000'
If sign=0 then FPSCR[FPRF] ← "+normal number"
If sign=1 then FPSCR[FPRF] ← "-normal number"
Done
```

Disabled Exponent Overflow

```
inc ←0
FPSCR[OX] ← 1
FPSCR[XX] ← 1
If FPSCR[RN]= b’00’ then/* Round to Nearest */
  Do
    inc ←1
```

```
    If FRB[0]=0 then FRT ← x'7FF0 0000 0000 0000'
    If FRB[0]=1 then FRT ← x'FFF0 0000 0000 0000'
    If FRB[0]=0 then FPSCR[FPRF] ← "+infinity"
    If FRB[0]=1 then FPSCR[FPRF] ← "-infinity"
  End
If FPSCR[RN]= b’01’ then/* Round Truncate */
  Do
    If (b'0' || FRB[1–63]) < x'047EF FFFF E000 0000' then inc ← 1
    If FRB[0]=0 then FRT ← x'47EF FFFF E000 0000'
    If FRB[0]=1 then FRT ← x'C7EF FFFF E000 0000'
    If FRB[0]=0 then FPSCR[FPRF] ← "+normal number"
    If FRB[0]=1 then FPSCR[FPRF] ← "-normal number"
  End
If FPSCR[RN]= b'10' then /* Round to +Infinity */
  Do
    If FRB[0]=0 then inc ← 0
    If (FRB[0]=1 & (FRB > x'C7EF FFFF E000 0000' then inc ← 1)
    If FRB[0]=0 then FRT ← x'7FF0 0000 0000 0000'
    If FRB[0]=1 then FRT ← x'C7EF FFFF E000 0000'
    If FRB[0]=0 then FPSCR[FPRF] ← "+infinity"
    If FRB[0]=1 then FPSCR[FPRF] ← "-normal number"
  End
If FPSCR[RN]= b'11' then/* Round to -Infinity */
  Do
    (If FRB[0]=0 & FRB < x'47EF FFFF E000 0000') then inc ← 1
    If FRB[0]= 1 then inc ← 1
    If FRB[0]=0 then FRT ← x'47EF FFFF E000 0000'
    If FRB[0]=1 then FRT ← x'FFF0 0000 0000 0000'
    If FRB[0]=0 then FPSCR[FPRF] ← "+normal number"
    If FRB[0]=1 then FPSCR[FPRF] ← "-infinity"
  End
FPSCR[FR] ← inc
FPSCR[FI] ← 1
Done
```

Enabled Exponent Overflow

```
sign ← FRB[0]
exp ← FRB[1–11] - 1023
   frac ← b'1' || [12–63]
   If frac[24–52]>0 then FPSCR[XX] ← 1
   Round single(sign,exp,frac,0,0,0)
Enabled Overflow
   FPSCR[OX] ← 1
   exp ← exp - 192
   FRT[0] ← sign
   FRT[1–11] ← exp + 1023
   FRT[12–63] ← frac[1–23] || b'0 0000 0000 0000 0000 0000 0000 0000'
   If sign=0 then FPSCR[FPRF] ← "+normal number"
   If sign=1 then FPSCR[FPRF] ← "-normal number"
Done
```

Zero Operand

```
FRT ← FRB
If FRB[0]=0 then FPSCR[FPRF] ← "+zero"
If FRB[0]=1 then FPSCR[FPRF] ← "-zero"
FPSCR[FR FI] ← b'00'
Done
```

Infinity Operand

```
FRT ← FRB
If FRB[0]=0 then FPSCR[FPRF] ← "+infinity"
If FRB[0]=1 then FPSCR[FPRF] ← "-infinity" Done
QNaN Operand–
FRT ← FRB[0–34] || b'0 0000 0000 0000 0000 0000 0000 0000'
FPSCR[FPRF] ← "QNaN"
FPSCR[FR FI] ← b'00'
Done
```

QNaN Operand

```
FRT ← FRB[0–34] || b'0 0000 0000 0000 0000 0000 0000 0000'
FPSCR[FPRF] ← "QNaN"
FPSCR[FR FI] ← b'00'
Done
```

SNaN Operand

```
FPSCR[VXSNAN] ← 1
If FPSCR[VE]=0 then
     Do
          FRT[0–11] ← FRB[0–11]
          FRT[12] ← 1
          FRT[13–63] ← FRB[13–34] || b'0 0000 0000 0000 0000 0000 0000
0000'
          FPSCR[FPRF] ← "QNaN"
     End
FPSCR[FR FI] ← b'00'
Done
```

Normal Operand

```
sign ← FRB[0]
exp ← FRB[1–11] - 1023
frac ← b'1' || FRB[12–63]
If frac[24–52]>0 then FPSCR[XX] ← 1
Round single(sign,exp,frac,0,0,0)
If exp>+127 and FPSCR[OE]=0 then go to Disabled Exponent Overflow
If exp>+127 and FPSCR[OE]=1 then go to Enabled Overflow
FRT[0] ← sign
FRT[1–11] ← exp + 1023
FRT[12–63] ← frac[1–23] || b'0 0000 0000 0000 0000 0000 0000 0000'
If sign=0 then FPSCR[FPRF] ← "+normal number"
If sign=1 then FPSCR[FPRF] ← "-normal number"
Done
```

Round Single (sign,exp,frac,G,R,X)

```
inc ← 0
lsb ← frac[23]
gbit ← frac[24]
rbit ← frac[25]
xbit ← (frac[26–52]||G||R||X)≠0
If FPSCR[RN]=b'00' then
  Do
    If sign || lsb || gbit || rbit || xbit = b’u11uu’ then inc ←1
    If sign || lsb || gbit || rbit || xbit = b’u011u’ then inc ←1
    If sign || lsb || gbit || rbit || xbit = b’u01u1’ then inc ←1
  End
If FPSCR[RN]= b'10' then
  Do
```

```
    If sign || lsb || gbit || rbit || xbit = b'0u1uu' then inc ← 1
    If sign || lsb || gbit || rbit || xbit = b'0uu1u' then inc ← 1
    If sign || lsb || gbit || rbit || xbit = b'0uuu1' then inc ← 1
  End
If FPSCR[RN]= b'11' then
  Do
    If sign || lsb || gbit || rbit || xbit = b'1u1uu' then inc ← 1
    If sign || lsb || gbit || rbit || xbit = b'1uu1u' then inc ← 1
    If sign || lsb || gbit || rbit || xbit = b'1uuu1' then inc ← 1
  End
frac[0–23] ← frac[0–23] + inc
If carry_out=1 then
  Do
    frac[0–23] ← b'1' || frac[0–22]
    exp ← exp + 1
  End
FPSCR[FR] ← inc
FPSCR[FI] ← gbit | rbit | xbit
Return
```

F.3.2 Floating-Point Convert to Integer Model

The following algorithm describes the operation of the floating-point convert to integer instructions. In this example, u represents an undefined hexadecimal digit.

```
If Floating Convert to Integer Word
  Then Do
    Then round_mode ← FPSCR[RN]
    tgt_precision ← "32-bit integer"
  End
If Floating Convert to Integer Word with round toward Zero
  Then Do
    round_mode ← b'01'
    tgt_precision ← "32-bit integer"
  End
If Floating Convert to Integer Doubleword
  Then Do
    round_mode ← FPSCR[RN]
    tgt_precision ← "64-bit integer"
  End
If Floating Convert to Integer Doubleword with round toward Zero
  Then Do
    round_mode ← b'01'
    tgt_precision ← "64-bit integer"
  End
If FRB[1–11]=2047 and FRB[12–63]=0 then goto Infinity Operand
If FRB[1–11]=2047 and FRB12=0 then goto SNaN Operand
If FRB[1–11]=2047 and FRB12=1 then goto QNaN Operand
If FRB[1–11]>1054 then goto Large Operand


sign ← FRB0
If FRB[1–11]>0 then exp ← FRB[1–11] - 1023 /* exp - bias */
If FRB[1–11]=0 then exp ← -1022
If FRB[1–11]>0 then frac[0–64]←b'01'||FRB[12–63]||b'00000000000'
/*normal*/
If FRB[1–11]=0 then frac[0–64]←b'00'||FRB[12–63]||b'00000000000'
/*denormal*/
```

```
gbit || rbit || xbit ← b'000'
Do i=1,31-exp
  temp (0..52)← '0'|| temp (0..52) || gbit || rbit || xbit ← b'0' ||
frac[0–64] || gbit || (rbit|xbit)
End


If gbit | rbit | xbit then FPSCR[XX] ← 1
```

Round Integer (frac,gbit,rbit,xbit,round_mode)

In this example, u represents an undefined hexadecimal digit. Comparisons ignore the u bits.

```
If sign=1 then frac[0–64] ← ¬frac[0–64] + 1


If tgt_precision="32-bit integer" and frac[0–64]>+2(31)-1
  then goto Large Operand
If tgt_precision="64-bit integer" and frac[0–64]>+2(63)-1
  then goto Large Operand
If tgt_precision="32-bit integer" and frac[0–64]<-2(31) then goto Large
Operand
If tgt_precision="64-bit integer" and frac[0–64]<-2(63) then goto Large
Operand
If tgt_precision="32-bit integer"
  then FRT ← x’xuuuuuuu’ || frac[33–64]
If tgt_precision="64-bit integer" then FRT ← frac[1–64]
FPSCR[FPRF] ← undefined
Done
```

Round Integer(frac,gbit,rbit,xbit,round_mode)

In this example, u represents an undefined hexadecimal digit. Comparisons ignore the u bits.

```
inc ← 0
If round_mode= b’00’ then
  Do
    If sign || frac[64] || gbit || rbit || xbit = b'u11uu' then inc ← 1
    If sign || frac[64] || gbit || rbit || xbit = b'u011u' then inc ← 1
    If sign || frac[64] || gbit || rbit || xbit = b'u01u1' then inc ← 1
  End
If round_mode= b'10' then
  Do
      If sign || frac[64] || gbit || rbit || xbit = b'0u1uu' then inc←1
      If sign || frac[64] || gbit || rbit || xbit = b'0uu1u' then inc ← 1
      If sign || frac[64] || gbit || rbit || xbit = b'0uuu1' then inc ← 1
    End
If round_mode= b'11' then
  Do
    If sign || frac[64] || gbit || rbit || xbit = b'1u1uu' then inc ← 1
    If sign || frac[64] || gbit || rbit || xbit = b'1uu1u' then inc ← 1
    If sign || frac[64] || gbit || rbit || xbit = b'1uuu1' then inc← 1
  End
frac[0–64] ← frac[0–64] + inc
FPSCR[FR] ← inc
FPSCR[FI] ← gbit | rbit | xbit
Return
```

Infinity Operand

```
FPSCR[FR FI VXCVI] ← b'001'
If FPSCR[VE]=0 then Do
   If tgt_precision="32-bit integer" then
      Do
         If sign=0 then FRT ← x'uuuu uuuu 7FFF FFFF'
         If sign=1 then FRT ← x'uuuu uuuu 8000 0000'
      End
   Else
      Do
         If sign=0 then FRT ← x'7FFF FFFF FFFF FFFF'
         If sign=1 then FRT ← x'8000 0000 0000 0000'
      End
   FPSCR[FPRF] ← undefined
   End
Done
```

SNaN Operand

```
FPSCR[FR FI VXCVI VXSNAN] ← b'0011'
If FPSCR[VE]=0 then
   Do
      If tgt_precision="32-bit integer"
         then FRT ← x'uuuu uuuu 8000 0000'
      If tgt_precision="64-bit integer"
         then FRT ← x'8000 0000 0000 0000'
      FPSCR[FPRF] ← undefined
   End
Done
```

QNaN Operand

```
FPSCR[FR FI VXCVI] ← b'001'
If FPSCR[VE]=0 then
   Do
      If tgt_precision="32-bit integer" then FRT ← x'uuuu uuuu 8000 0000'
      If tgt_precision="64-bit integer" then FRT ← x'8000 0000 0000 0000'
      FPSCR[FPRF] ← undefined
   End
Done
```

Large Operand

```
FPSCR[FR FI VXCVI] ← b'001'
If FPSCR[VE]=0 then Do
   If tgt_precision="32-bit integer" then
      Do
         If sign=0 then FRT ← x'uuuu uuuu 7FFF FFFF'
         If sign=1 then FRT ← x'uuuu uuuu 8000 0000'
      End
   Else
      Do
         If sign=0 then FRT ← x'7FFF FFFF FFFF FFFF'
         If sign=1 then FRT ← x'8000 0000 0000 0000'
      End
   FPSCR[FPRF] ← undefined
   End
Done
```

Appendix G Synchronization Programming Examples

The examples in this appendix show how synchronization instructions can be used to emulate various synchronization primitives and how to provide more complex forms of synchronization.

For each of these examples, it is assumed that a similar sequence of instructions is used by all processes requiring synchronization of the accessed data.

G.1 General Information

The following points provide general information about the **lwarx** and **stwcx.** instructions:

- In general, **lwarx** and **stwcx.** instructions should be paired, with the same effective address used for both. The exception is an isolated **stwcx.** instruction that is used to clear any existing reservation on the processor, for which there is no paired **lwarx** and for which any (scratch) effective address can be used.
- It is acceptable to execute an **lwarx** instruction for which no **stwcx.** instruction is executed. For example, such a dangling **lwarx** instruction occurs if the value loaded in the Test and Set sequence shown in Section G.2.5, “Test and Set,” is not zero.
- To increase the likelihood that forward progress is made, it is important that looping on **lwarx**/**stwcx.** pairs be minimized. For example, in the sequence shown in Section G.2.5, “Test and Set,” this is achieved by testing the old value before attempting the store—were the order reversed, more **stwcx.** instructions might be executed, and reservations might more often be lost between the **lwarx** and the **stwcx.** instructions.
- The manner in which **lwarx** and **stwcx.** are communicated to other processors and mechanisms, and between levels of the memory subsystem within a given processor is implementation-dependent. In some implementations performance may be improved by minimizing looping on an **lwarx** instruction that fails to return a desired value. For example, in the example provided in Section G.2.5, “Test and Set,” if the programmer wishes to stay in the loop until the word loaded is zero, he could change the “**bne** $ 12” to “**bne** loop”. However, in some implementations better performance may be obtained by using an ordinary Load instruction to do the initial checking of the value, as follows:

```
loop:   lwz     r5,0(r3) #load the word
        cmpwi   r5,0     #loop back if word
        bne     loop     #not equal to 0
        lwarx   r5,0,r3  #try again, reserving
        cmpwi   r5,0     #(likely to succeed)
        bne     loop     #try to store nonzero
        stwcx.  r4,0,r3  #
        bne     loop     #loop if lost reservation
```

- In a multiprocessor, livelock is possible if a loop containing an **lwarx**/**stwcx.** pair also contains an ordinary write instruction for which any byte of the affected memory area is in the reservation granule of the reservation. For example, the first code sequence shown in Section G.5, “List Insertion,” can cause livelock if two list elements have next element pointers in the same reservation granule.

G.2 Synchronization Primitives

The following examples show how the **lwarx** and **stwcx.** instructions can be used to emulate various synchronization primitives. The sequences used to emulate the various primitives consist primarily of a loop using **lwarx** and **stwcx.**. Additional synchronization is unnecessary, because the **stwcx.** will fail, clearing the EQ bit, if the word loaded by **lwarx** has changed before the **stwcx.** is executed.

G.2.1 Fetch and No-Op

The Fetch and No-Op primitive atomically loads the current value in a word in memory. In this example it is assumed that the address of the word to be loaded is in GPR3 and the data loaded are returned in GPR4.

```
loop:   lwarx   r4,0,r3   #load and reserve
        stwcx.  r4,0,r3   #store old value if still reserved
        bne-    loop      #loop if lost reservation
```

If the **stwcx.** operation succeeds, the same value loaded by the preceding **lwarx** is stored. While the store is redundant, the use of the **stwcx.**, ensures atomic access to the location.

G.2.2 Fetch and Store

The Fetch and Store primitive atomically loads and replaces a word in memory.

In this example it is assumed that the address of the word to be loaded and replaced is in GPR3, the new value is in GPR4, and the old value is returned in GPR5.

```
loop:   lwarx   r5,0,r3   #load and reserve
        stwcx.  r4,0,r3   #store new value if still reserved
        bne-    loop      #loop if lost reservation
```

G.2.3 Fetch and Add

The Fetch and Add primitive atomically increments a word in memory.

In this example it is assumed that the address of the word to be incremented is in GPR3, the increment is in GPR4, and the old value is returned in GPR5.

```
loop:   lwarx   r5,0,r3         #load and reserve
        add     r0,r4,r5        #increment word
        stwcx.  r0,0,r3         #store new value if still reserved
        bne-    loop            #loop if lost reservation
```

G.2.4 Fetch and AND

The Fetch and AND primitive atomically ANDs a value into a word in memory.

In this example it is assumed that the address of the word to be ANDed is in GPR3, the value to AND into it is in GPR4, and the old value is returned in GPR5.

```
loop:   lwarx   r5,0,r3         #load and reserve
        and     r0,r4,r5        #AND word
        stwcx.  r0,0,r3         #store new value if still reserved
        bne-    loop            #loop if lost reservation
```

Note: This sequence can be changed to perform another Boolean operation atomically on a word in memory, simply by changing the AND instruction to the desired Boolean instruction (OR, XOR, etc.).

G.2.5 Test and Set

The Test and Set primitive atomically loads a word from memory, ensures that the word in memory contains a nonzero value, and sets the EQ bit of CR Field 0 according to whether the value loaded is zero.

In this example it is assumed that the address of the word to be tested is in GPR3, the new value (nonzero) is in GPR4, and the old value is returned in GPR5.

```
loop:   lwarx   r5,0,r3   #load and reserve
        cmpwi   r5, 0     #done if word
        bne     $+12      #not equal to 0
        stwcx.  r4,0,r3   #try to store nonzero
        bne-    loop      #loop if lost reservation
```

Notes:

1. Test and Set is shown primarily for pedagogical reasons. It is useful on machines that lack the better synchronization facilities provided by **lwarx** and **stwcx.**. Test and Set does not scale well. Using Test and Set before a critical section allows only one process to execute in the critical section at a time. Using **lwarx** and **stwcx.** to bracket the critical section allows many processes to execute in the critical section at once, but at most one will succeed in exiting from the section with its results stored.
2. Depending on the application, if Test and Set fails (that is, clears the EQ bit of CR Field 0) it may be appropriate to re-execute the Test and Set.

G.3 Compare and Swap

The Compare and Swap primitive atomically compares a value in a register with a word in memory; if they are equal, stores the value from a second register into the word in memory, if they may be unequal loads the word from memory into the first register, and sets the EQ bit of CR Field 0 to indicate the result of the comparison.

In this example it is assumed that the address of the word to be tested is in GPR3, the compared value is in GPR4, the new value is in GPR5, and the old value is returned in GPR6.

```
loop:   lwarx   r6,0,r3   #load and reserve
        cmpw    r4,r6     #first 2 operands equal ?
        bne-    exit      #skip if not
        stwcx.  r5,0,r3   #store new value if still reserved
        bne-    loop      #loop if lost reservation
exit:   mr      r4,r6     #copy 2nd operand to 1st
```

Notes:

1. The semantics in this example are based on the IBM System/370™ Compare and Swap instruction. Other architecture may define this instruction differently.
2. Compare and Swap is shown primarily for pedagogical reasons. It is useful on machines that lack the better synchronization facilities provided by **lwarx** and **stwcx.**. Although the instruction is atomic, it checks only for whether the current value matches the old value. An error can occur if the value had been changed and restored before being tested.
3. Depending on the application, if Compare and Swap fails (that is, clears the EQ bit of CR0) it may be appropriate to recompute the value potentially to be stored and then re-execute the Compare and Swap.

G.4 Lock Acquisition and Release

This example provides an algorithm for locking that demonstrates the use of synchronization with an atomic read/modify/write operation. GPR3 provides a shared memory location, the address of which is an argument of the lock and unlock procedures. This argument is used as a lock to control access to some shared resource such as a data structure. The lock is open when its value is 0 and locked when it is 1. Before accessing the shared resource, a processor sets the lock by having the lock procedure call TEST_AND_SET, which executes the code sequence shown in Section G.2.5, “Test and Set.” This atomically sets the old value of the lock, and writes the new value (1) given to it in GPR4, returning the old value in GPR5 (not used below) and setting the EQ bit in CR0 according to whether the value loaded is 0. The lock procedure repeats the test and set procedure until it successfully changes the value in the lock from 0 to 1.

The processor must not access the shared resource until it sets the lock. After the **bne-** instruction that checks for the successful test and set operation, the processor executes the **isync** instruction. This delays all subsequent instructions until all previous instructions have

completed to the extent required by context synchronization. The **sync** instruction could be used but performance would be degraded because the **sync** instruction waits for all outstanding memory accesses to complete with respect to other processors. This is not necessary here.

```
lock:   li      r4,1            #obtain lock
loop:   bl      test_and_set    #test and set
        bne-    loop            #retry until old = 0
                                #delay subsequent instructions until
                                #previous ones complete
        isync
        blr                     #return
```

The unlock procedure writes a 0 to the lock location. If the access to the shared resource includes write operations, most applications that use locking require the processor to execute a **sync** instruction to make its modification visible to all processors before releasing the lock. For this reason, the unlock procedure in the following example begins with a **sync**.

```
unlock:sync                     #delay until prior stores finish
        li      r1,0            #store zero to lock location
        stw     r1,0(r3)
        blr                     #return
```

G.5 List Insertion

The following example shows how the **lwarx** and **stwcx.** instructions can be used to implement simple LIFO (last-in-first-out) insertion into a singly-linked list. (Complicated list insertion, in which multiple values must be changed atomically, or in which the correct order of insertion depends on the contents of the elements, cannot be implemented in the manner shown below, and requires a more complicated strategy such as using locks.)

The next element pointer from the list element after which the new element is to be inserted, here called the parent element, is stored into the new element, so that the new element points to the next element in the list; this store is performed unconditionally. Then the address of the new element is conditionally stored into the parent element, thereby adding the new element to the list.

In this example it is assumed that the address of the parent element is in GPR3, the address of the new element is in GPR4, and the next element pointer is at offset 0 from the start of the element. It is also assumed that the next element pointer of each list element is in a reservation granule separate from that of the next element pointer of all other list elements.

```
loop:   lwarx   r2,0,r3         #get next pointer
        stw     r2,0(r4)        #write back new element
        sync                    #let store settle (can omit if not MP)
        stwcx.  r4,0,r3         #add new element to list
        bne-    loop            #loop if stwcx. failed
```

In the preceding example, if two list elements have next element pointers in the same reservation granule then, in a multiprocessor, livelock can occur. (Livelock is a state in which processors interact in a way such that no processor makes progress.)

If it is not possible to allocate list elements such that each element's next element pointer is in a different reservation granule, then livelock can be avoided by using the following, more complicated, code sequence.

```
        lwz     r2,0(r3)        #get next pointer
loopl:  mr      r5,r2           #keep a copy
        stw     r2,0(r4)        #write back new element
        sync                    #let store settle
loop2:  lwarx r2,0,r3           #get it again
        cmpw    r2,r5           #loop if changed (someone
        bne-    loopl           #else progressed)
        stwcx. r4,0,r3          #add new element to list
        bne-    loop2           #loop if failed
```

Appendix H
Implementation Summary for Programmers

The PowerPC 601 microprocessor is the first implementation of the PowerPC architecture. It offers a reliable platform for software and hardware developers to make products compatible with subsequent processors in the PowerPC family. In addition, the 601 provides extensions to the PowerPC architecture that allow it to function as a bridge from the POWER architecture. This appendix describes the POWER extensions as well as other differences between the 601 and the PowerPC architecture (*PowerPC Architecture, First Edition*). These differences can be categorized as follows:

- POWER extensions—Additional functionality not defined in the PowerPC architecture. For example, the 601 implements many POWER instructions that do not have PowerPC architecture equivalents.
- Variances—601 functionality that is implemented differently than as described in the PowerPC architecture. For example, there are several differences between the 601 MMU implementation and that specified by the PowerPC architecture. In general, these variances are not visible from the user level.
- Implementation-dependent extensions—These include features that are not part of but are allowed by the PowerPC architecture. For example, the 601 provides a set of implementation-dependent registers (HIDs) to control hardware features such as parity checking and instruction address breakpoint that are beyond the specifications in architecture. Software should take appropriate precautions to control use of these features.
- PowerPC architecture optional features—These include optional features defined in the PowerPC architecture that are implemented in the 601.

This appendix does not describe performance trade-offs allowed by the PowerPC architecture. For example, some implementations may provide more support for alignment than others and, therefore, they may require different amounts of assistance in the exception handler.

This appendix also does not describe variances built into the PowerPC architecture to provide some latitude in PowerPC implementations for handling reserved, invalid, and undefined conditions. These aspects are left intentionally undefined. Note that while the 601's treatment of such aspects may be predictable, taking advantage of that behavior may

cause software incompatibilities with other PowerPC implementations.

Where applicable, a reference is given to the section that describes that functionality.

Also, the tables in this appendix indicate the level of architecture at which the 601 diverges. These levels are as follows:

- PowerPC user instruction set architecture—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 uniprocessor environment.

  The tables in this appendix identify differences with this part of the architecture by listing “user” in the “Level” column of the tables.

- PowerPC virtual environment architecture—Describes the memory model for a multiprocessor environment, defines cache control instructions, and describes other aspects of virtual environments. Implementations that conform to the PowerPC virtual environment architecture also adhere to the PowerPC user instruction set architecture, but may not necessarily adhere to the PowerPC operating environment architecture.

  The tables in this appendix identify differences with this part of the architecture by listing “virtual environment” in the “Level” column of the tables.

- PowerPC operating environment architecture—Defines the memory management model, supervisor-level registers, synchronization requirements, and the exception model. Implementations that conform to the PowerPC operating environment architecture also adhere to the PowerPC user instruction set architecture and the PowerPC virtual environment architecture definition.

  The tables in this appendix identify differences with this part of the architecture by listing “operating environment” in the “Level” column of the tables.

H.1 POWER Extensions

Table H-1 lists POWER functionality supported by the 601 that is not defined in the PowerPC architecture. POWER extensions include additional functionality not defined in the PowerPC architecture.

Table H-1. POWER Extensions

Difference	Reference	Level	Type
The MQ register, provided for POWER compatibility, is not part of the PowerPC architecture.	Section 2.2.5.1, "MQ Register (MQ)"	User	POWER
The 601 implements the real-time clock feature, including POWER registers RTCU and RTCL, to provide a time reference rather than the time base feature defined by the PowerPC architecture.	Section 2.2.5.3, "Real-Time Clock (RTC) Registers (User-Level)"	Virtual and operating environment	POWER and Variance
In the 601 processor, the decrementer implementation uses the separate 7.8125 MHz RTC for its base frequency. Other PowerPC processors base the decrementer on the processor clock.	Section 2.3.3.5, "Decrementer (DEC) Register"	User and virtual environment	POWER and Variance
The decrementer register (DEC) in the 601 allows user-level read access, which is not provided in the PowerPC architecture.	Section 2.3.3.5, "Decrementer (DEC) Register"	User and operating environment	POWER
Because 601 supports the POWER registers, MQ, RTCU, and RTCL, instruction encodings to access them, **mtspr** and **mfspr**, are also provided.	Section 3.7.2, "Move to/from Special-Purpose Register Instructions"	User and operating environment	POWER
The 601 provides a group of instructions for compatibility with POWER. The relationship between the 601 and the PowerPC instruction sets are shown in Appendix C. The PowerPC architecture defines these instructions as reserved.	Appendix C, "PowerPC Instructions Not Implemented"	—	POWER

H.2 Variances

Variances include PowerPC architecture functionality that is implemented in the 601 with some differences. In general, these variances are not visible from the user level. Table H-2 lists variances to the PowerPC architecture.

Table H-2. Variances

Difference	Reference	Level	Type
The VXSOFT, VXSQRT, and NI bits (bits 21, 22, and 29, respectively) are not implemented in the 601 processor.	Section 2.2.3, "Floating-Point Status and Control Register (FPSCR)"	User and operating environment	Variance
If the Floating-Point Convert to Integer Word (**fctiw**) instruction results in a conversion exception, FPSCR[XCVI] is set causing FPSCR[VX] to be set. The PowerPC architecture specifies that when **fctiw** causes FPSCR[XCVI] to be set, FPSCR[XX] is not altered. The 601 may set both FPSCR[XX] and FPSCR[XCVI] in some circumstances.	Section 3.4.3, "Floating-Point Rounding and Conversion Instructions"	User and operating environment	Variance
The architecture requires that both FPSCR[VXCVI] and FPSCR[VXSNAN] be set when the source operand of a **fctiw** is an SNAN. 601 sets only FPSCR[VXCVI].	Section 3.4.3, "Floating-Point Rounding and Conversion Instructions"	User and operating environment	Variance
PowerPC architecture defines the following bits in the machine state register (MSR) not implemented in the 601: **Bit Description** --- 13 Power management enable (POW) 15 Exception little-endian mode (ELE) 22 Branch trace enable (BE) 30 Recoverable exception (RE) 31 Little-endian mode (LE)	Section 2.3.1, "Machine State Register (MSR)"	Operating environment	Variance
The 601 provides a bit in an implementation-specific register (HID0) for selecting between big- and little-endian modes. PowerPC architecture defines MSR[LE] for this purpose.	Section 2.4.3, "Byte and Bit Ordering"	Operating environment	Variance
The number, function, content, and format of the BAT registers implemented by the 601 is different than that specified by the PowerPC architecture.	Section 2.3.3.12, "BAT Registers"	Operating environment	Variance
The 601 clears the reservation bit set by the execution of an **lwarx** instruction when taking any type of exception. The PowerPC architecture defines that the reservation be cleared for a subset of exceptions.	Section 3.5.7, "Memory Synchronization Instructions"	User and operating environment	Variance
Because the 601 does not implement the MSR[RE] bit (recoverable exception bit), the operating system must use other criteria to determine if it is possible to recover from an asynchronous, imprecise exception.	Section 5.4.1, "Reset Exceptions (x'00100')"	Operating environment	Variance

Table H-2. Variances (Continued)

Difference	Reference	Level	Type
Instruction access exceptions due to instruction fetches from I/O controller interface segments do not set the SRR1[3] bit as defined in the PowerPC architecture. This condition clears SRR1[0–15].	Chapter 5, “Exceptions”	Operating environment	Variance
The non-IEEE mode bit, FPSCR[NI], is reserved in the 601. All floating-point results are consistent with IEEE standards.	Section 5.4.7.1, "Floating-Point Enabled Program Exceptions”	Operating environment	Variance
The 601 maps I/O controller interface error conditions to I/O controller interface exceptions instead of to the data access exception vector specified by the PowerPC architecture.	Section 5.4.10, "I/O Controller Interface Error Exception (x'00A00')”	Operating environment	Variance
Unlike exceptions that occur with memory accesses, loads, loads with update, and stores with update to I/O controller interface segments cause any target registers to be updated, regardless of whether an exception is taken.	Chapter 5, “Exceptions”	Operating environment	Variance
The 601 does not implement the trace exception as a separate exception as is defined in the PowerPC architecture (x'00D00'). The 601 vectors trace exceptions to the run-mode/trace exception (x'02000').	Section 5.4.12, "Run Mode/Trace Exception (x'02000')”	Operating environment	Variance
The 601 allows access to the I/O controller interface regardless of the setting or MSR[DT]. The PowerPC architecture does not allow these accesses when MSR[DT] is cleared.	Section 6.1.3, "Address Translation Mechanisms”	Operating environment	Variance
The 601 does not implement the PowerPC **tlbsync** instruction, but instead requires the use of a **sync** instruction to synchronize the completion of a broadcast **tlbie** instruction.	Section 10.3, "Instructions Not Implemented by the 601”	Operating environment	Variance
PowerPC architecture defines a “Guarded” memory attribute used to protect volatile memory. This attribute is associated with each virtual page (guarded bit in the page table entry) and with real memory. The 601 provides a similar function using the “Caching Inhibited” memory attribute.	—	Operating environment	Variance

H.3 Implementation-Dependent Extensions

Implementation-dependent extensions include features that are not part of, but are allowed by, the PowerPC architecture. Note that there are a number of such extensions that are described throughout this book, and there is no attempt to list them exhaustively in this appendix. Table H-3 provides a brief list of key implementation-dependent extensions supported by the 601.

Table H-3. Implementation-Dependent Extensions

Difference	Reference	Level	Type
The 601 includes the following implementation-specific registers: HID0, HID1, HID2, HID5, HID15. These may or may not be included in future implementations.	Section 2.3.3.13.1, "Checkstop Sources and Enables Register—HID0," through Section 2.3.3.13.5, "Processor Identification Register (PIR)—HID15"	Operating environment	Implementation -dependent extensions
Because the 601 automatically handles all floating-point data types, the 601 floating-point assist exception defined in the PowerPC architecture (x'00E00') would never be taken by the 601, and therefore it is not implemented.	—	Operating environment	Implementation -dependent extension
The 601 supports an additional exception called the run mode exception in addition to the exceptions defined by the PowerPC architecture.	Section 5.4.12, "Run Mode/Trace Exception (x'02000')"	Operating environment	Implementation -dependent extension
The 601 includes a feature that supports 256-Mbyte translation capability. This is enabled when the T bit is set and the BUID field equals x'07F' in the appropriate segment register(s).	Section 6.5.2.1, "I/O Controller Interface Address Translation: T = 1 in Segment Register"	Operating environment	Implementation -dependent extension

H.4 Options to the PowerPC Architecture

Table H-4 lists options to the PowerPC architecture supported by the 601. Note that because these are optional, they may not be supported by all PowerPC processors, just as the 601 does not support other optional features supported by the architecture.

Table H-4. Options to the PowerPC Architecture

Difference	Reference	Level	Type
The 601 processor implements the external access register (EAR), which is optional to the PowerPC architecture. Note that only four bits (28–31) are implemented in the 601, whereas the PowerPC architecture defines six bits.	Section 2.3.3.10, "External Access Register (EAR)"	Operating environment	Optional
The 601 implements the **eciwx** and **ecowx** instructions that are optional to the PowerPC architecture but are required for use with the EAR register.	Section 3.9, "External Control Instructions"	User	Optional

Appendix I
Instruction Timing Examples

This appendix describes timing of various code sequences in the PowerPC 601 microprocessor, showing situations where stalls might occur, and where possible, how those stalls can be avoided.

This section shows timing for combinations of instructions, showing the interactions between the various instructions whose individual timings are described in Chapter 7, “Instruction Timing.” From these examples, one can develop similar tables for other code sequences. This information is intended to show how to schedule code efficiently.

I.1 Branch Instruction Timing Examples

The examples in this section primarily illustrate branch timing issues. It is split into two main subsections—Section I.1.1, “General Branch Timing” which addresses timing issues not associated with branch prediction or condition register (CR) coherency and Section I.1.1.6, “Conditional Branch Timing,” which addresses timing issues associated with branch prediction and CR coherency.

I.1.1 General Branch Timing

This section gives timing examples showing all of the stalls that can occur for branches or be caused by branches and gives timing examples that avoid those stalls.

I.1.1.1 Dispatch Considerations

Instructions may stall in the DS stage when a branch cannot be dispatched for some reason even though the branch is in the dispatch buffer (IQ0 through IQ3). This section does not describe cases involving unresolved conditional branches. For information, see Section I.1.1.6, “Conditional Branch Timing.”

As described in Section 7.3.1.4.1, “Branch Dispatch,” dependency checking for branches occurs during the dispatch stage—thus register dependencies can cause dispatch stalls. The two registers that branch instructions use as sources are the link register (LR) and count register (CTR). In both cases, dependencies on previous branch instructions are resolved via data-forwarding in the BPU. This is illustrated in Table I-1 and Table I-2.

In Table I-1, the CTR value from the **bc** instruction in the first iteration is forwarded to the **bc** instruction in the second iteration (clock 5).

Table I-1. Two Integer Instruction Counter Loop

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Memory	FA	I1,I2,**bc**		I1,I2,**bc**		I1,I2,**bc**		I1,I2,**bc**
	CARB	I1,I2,**bc**		I1,I2,**bc**		I1,I2,**bc**		I1,I2,**bc**
	CACC		I1,I2,**bc**		I1,I2,**bc**		I1,I2,**bc**	
Dispatch	IQ2			**bc**		**bc**		**bc**
	IQ1			I2		I2		I2
	IQ0			I1	I2,Tag**bc**	I1	I2,Tag**bc**	I1
IU	ID			I1	I2,Tag**bc**	I1	I2,Tag**bc**	I1
	IE				I1	I2,Tag**bc**	I1	I2,Tag**bc**
	IC					I1	I2,Tag**bc**	I1
	IWA					I1	I2	I1
	IWL							
	FPSB							
	ISB							
BPU	BE			**bc**		**bc**		**bc**
	MR							
	BW				**bc**	**bc**	**bc**,**bc**	**bc**
Notes						CTR value from the **bc** in the first iteration is forwarded to the **bc** in the second iteration.	Decrement CTR this cycle.	

In Table I-2, the link address is forwarded to the **bclr** instruction in clock 5.

Table I-2. Two Integer Instruction Leaf Procedure

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Memory	FA	I1,I2, **bl**, I5		I3,I4, **bclr**		I5		
	CARB	I1,I2, **bl**, I5		I3,I4, **bclr**		I5		
	CACC		I1,I2,**bl**, I5		I3,I4, **bclr**		I5	
Dispatch	IQ3			I5				
	IQ2			**bl**		**bclr**		
	IQ1			I2		I4		
	IQ0			I1	I2,Tag**bl**	I3	I4	I5
IU	ID			I1	I2,Tag**bl**	I3	I4	I5
	IE				I1	I2,Tag**bl**	I3	I4
	IC					I1	I2,Tag**bl**	I3
	IWA					I1	I2	I3
	IWL							
	FPSB							
	ISB							
BPU	BE			**bl**		**bclr**		
	MR							
	BW				**bl**	**bl**	**bl**	
Notes						Forward link address to **bcr** from link shadow.	Update the LR with the address of I3.	

I.1.1.2 Integer Instruction Dependencies

If a register dependency on a branch originates from an integer instruction, such as **mtlr** or **mtctr**, the data resides in the GPR and cannot be forwarded to the BPU (there is no extra read port on the GPR for the BPU). In this case, a dispatch stall may occur, as shown in Table I-3 and Table I-4.

Table I-3 shows the instruction timing for when a **bclr** instruction follows an **mtlr** instruction. Note that **bclr** stalls in cycle 3 because of a dependency on the **mtlr** instruction. In this case the branch is taken, and the target instruction enters the ID stage in cycle 7, after a three-cycle stall.

Table I-3. Register Dependency Stalls for a Taken Branch (mtlr -> bclr)

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Memory	FA	**mtlr**, **bclr**				target		
	CARB	**mtlr**, **bclr**				target		
	CACC		**mtlr**, **bclr**				target	
Dispatch	IQ1			**bclr**				
	IQ0			**mtlr**	**bclr**	**bclr**		target
IU	ID			**mtlr**	<stall 1>	<stall 2>	<stall 3>	target
	IE				**mtlr**			
	IC					**mtlr**		
	IWA					**mtlr**		
	IWL							
	FPSB							
	ISB							
BPU	BE					**bclr**		
	MR							
	BW							
Notes				The **bclr** instruction stalls due to register dependency on the **mtlr** instruction		Register dependency on **bclr** is resolved and **bclr** is dispatched		

Table I-4 shows the timing for same instruction sequence shown in Table I-3, when the branch is not taken, showing that the instruction following the **bclr** instruction can be dispatched in cycle 6, the cycle after the **bclr** instruction is resolved.

Table I-4. Register Dependency Stalls for an Untaken Branch (mtlr -> bclr)

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Memory	FA	**mtlr**, **bclr**						
	CARB	**mtlr**, **bclr**						
	CACC		**mtlr**,**bclr**					
Dispatch	IQ2			seq. ins.				
	IQ1			**bclr**	seq. ins.	seq. ins.		
	IQ0			**mtlr**	**bclr**	**bclr**	seq. ins.	
IU	ID			**mtlr**	<stall 1>	<stall 2>	seq. ins.	
	IE				**mtlr**			seq. ins.
	IC					**mtlr**		
	IWA					**mtlr**		
	IWL							
	FPSB							
	ISB							
	CA							
BPU	BE					**bclr**		
	MR							
	BW							
Notes				The **bclr** instruction stalls due to register dependency from **mtlr** instruction		Register dependency on **bclr** is resolved and **bclr** is dispatched	Instruction following the **bclr** instruction can now be dispatched	

I.1.1.3 SPR Dependencies with LR and CTR

To avoid dispatch stalls, instructions should be placed between the **mtspr** instruction and the branch that depends upon it. The LR and the CTR have the same timing characteristics relative to **mtspr** dependencies (the following examples use the LR for clarity).

Examples of this are shown in Table I-5 and Table I-6. Table I-5 shows how the stall shown in Table I-3 is avoided by scheduling integer instructions between the **bclr** and **mtlr** instructions.

Table I-5. Instruction Sequence that Avoids Stalls Caused by a Register Dependency for a Taken Branch (mtlr -> bclr)

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Memory	FA	**mtlr**, I1, I2, I3, **bclr**				target		
	CARB	**mtlr**, I1, I2, I3, **bclr**				target		
	CACC		**mtlr**, I1, I2, I3, **bclr**				target	
Dispatch	IQ4			**bclr**				
	IQ3			I3	**bclr**			
	IQ2			I2	I3	**bclr**		
	IQ1			I1	I2	I3		
	IQ0			**mtlr**	I1	I2	I3	target
IU	ID			**mtlr**	I1	I2	I3	target
	IE				**mtlr**	I1	I2	I3
	IC					**mtlr**	I1	I2
	IWA					**mtlr**	I1	I2
	IWL							
	FPSB							
	ISB							
BPU	BE					**bclr**		
	MR							
	BW							
Notes					Integer instructions use slots that would have been stalls. Note that the branch is not dispatched any sooner, the integer stalls are simply used for useful work.			

Table I-6 shows how the stall shown in Table I-4 is avoided by scheduling integer instructions between the **bclr** and **mtlr** instructions.

Table I-6. Register Dependency Stalls for an Untaken Branch (mtlr -> bclr)

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Memory	FA	**mtlr**, I1, I2, **bclr**, seq. ins.						
	CARB	**mtlr**, I1, I2, **bclr**, seq. ins.						
	CACC		**mtlr**, I1, I2, **bcl**, seq. ins.					
Dispatch	IQ4			seq. ins.				
	IQ3			**bclr**	seq. ins.			
	IQ2			I2	**bclr**	seq. ins.		
	IQ1			I1	I2	**bclr**		
	IQ0			**mtlr**	I1	I2	seq. ins.	
IU	ID			**mtlr**	I1	I2	seq. ins.	
	IE				**mtlr**	I1	I2	seq. ins.
	IC					**mtlr**	I1	I2
	IWA					**mtlr**	I1	I2
	IWL							
	FPSB							
	ISB							
BPU	BE					**bclr**		
	MR							
	BW							
Notes					Integer instructions use slots that would have been stalls. Note that the branch is not dispatched any sooner, the integer stalls are simply used for useful work.			

I.1.1.4 Stalls Caused when the Link Shadow Register Is Full

Nonconditional branches encounter dependencies when the link shadow register is full and a branch and link instruction needs to be dispatched. This occurs when either there is a long-latency instruction, such as a multiply or divide instruction, in the IU, or there is a string of

‘not-taken’ branch and link instructions separated by very few integer instructions. This is shown in Table I-7 and Table I-8.

In Table I-7, the link shadow register is filled because of a series of branch-and-link instructions. The third **bcl** instruction stalls in cycle 5 for one cycle because the link shadow registers were filled in the previous two cycles.

Table I-7. Series of Not-Taken Branch-and-Link Instructions that Cause the Link Shadow Registers to Fill

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Memory	FA	I1, **bc**l1, **bc**l2, **bc**l3					Target**bc**l3	
	CARB	I1, **bc**l1, **bc**l2, **bc**l3					Target**bc**l3	
	CACC		I1, **bc**l1, **bc**l2, **bc**l3					Target**bc**l3
Dispatch	IQ3			**bc**l3				
	IQ2			**bc**l2				
	IQ1			**bc**l1	**bc**l3			
	IQ0			I1	**bc**l2	**bc**l3	**bc**l3	
IU	ID			I1				
	IE				I1,Tag**bc**l1	Tag**bc**l2		Tag**bc**l3
	IC					I1,Tag**bc**l1	Tag**bc**l2	
	IWA					I1		
	IWL							
	FPSB							
	ISB							
BPU	BE			**bc**l1	**bc**l2		**bc**l3	
	MR							
	BW				**bc**l1	**bc**l1,**bc**l2	**bc**l2	**bc**l3
Notes				First link shadow register used	Second link shadow register used	**bc**l3 stalls for lack of a link shadow register LR is updated for **bc**l1	Link shadow for **bc**l3 is now available	

Table I-8 shows a stall that occurs when the link shadow register is filled because of multicycle integer instructions

Table I-8. Long-Latency Integer Instruction Followed by Branch and Link Instructions that Cause the Link Shadow Registers to Fill

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Memory	FA	**muls**, **bl**,		I1, **bl**		I2, **bl**		
	CARB	**muls**, **bl**,		I1, **bl**		I2, **bl**		
	CACC		**muls**,**bl**		I1, **bl**		I2, **bl**	
Dispatch	IQ1			**bl**		**bl**		**bl**
	IQ0			**muls**		I1		I2
IU	ID			**muls**		I1	I1,Tag**bl**	I1,Tag**bl**
	IE				**muls**, Tag**bl**	**muls**, Tag**bl**	**muls**, Tag**bl**	**muls**, Tag**bl**
	IC							
	IWA							
	IWL							
	FPSB							
	ISB							
BPU	BE			**bl**		**bl**		
	MR							
	BW							
Notes				First link shadow used		Second link shadow used		Branch in Q1 stalls because there are no free link shadow registers

Table I-8. Long-Latency Integer Instruction Followed by Branch and Link Instructions that Fill the Link Shadow Registers (Continued)

Pipeline Stage		Cycle Number				
Unit	**Stage**	**8**	**9**	**10**	**11**	**12**
Memory	FA			target		
	CARB			target		
	CACC				target	
Dispatch	IQ1	**bl**	**bl**			
	IQ0	I2	I2	**bl**		target
IU	ID	I1,Tag**bl**	I2			target
	IE	**muls**,Tag**bl**	I1,Tag**bl**	I2	Tag**bl**	
	IC		**muls**,Tag**bl**	I1,Tag**bl**	I2	Tag**bl**
	IWA		**mulls**	I1	I2	
	IWL					
	FPSB					
	ISB					
BPU	BE			**bl**		**bl**
	MR					
	BW					
Notes			First **bl** completes; one link shadow becomes available	Second **bl** completes; third **bl** is executed		Third **bl** completes

I.1.1.5 Performance Considerations when an Instruction Cannot be Dispatched Out of Order—Link and Count Tag Dependencies

Branch-related stalls can occur when a branch or floating-point instruction cannot be dispatched out of-order. Such an instruction cannot be tagged to an integer instruction and is instead tagged to a bubble in the IU pipeline. These stalls are significant only if the dispatch of subsequent integer instructions is delayed. As described in Section 7.3.1.4.4, “Synchronization Tags for the Precise Exception Model,” there are three types of tags associated with branch instructions:

- Link tags—for instructions that update the LR
- Counter tags—for instructions that update the CTR
- Predicted branch tags—for unresolved conditional branches

The link and counter tags are used to synchronize the IW and BW stages and therefore

remain in the IU pipeline through the IW stage. These tags can be carried with an integer instruction or with a bubble in the integer pipeline, but only one tag of a given type can reside in a stage. Thus consecutive branch instructions that generate the same type of tag require a bubble in the integer pipeline to propagate the tag for the second instruction.

Table I-9 shows a scenario where multiple branch tags create a bubble in the IU pipeline. In this example, the **bl** instruction takes the link tag in the I3 instruction. Since the tag is occupied, the subsequent **bcl** instruction cannot tag to I3 and a bubble must be created (cycle 5) to provide a link tag for the **bcl** instruction.

Table I-9. Multiple Branch Tags Causing a Bubble in the Integer Pipeline

Pipeline Stage		**Cycle Number**						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Memory	FA	I1, I2, I3, **bl**		**bcl**, I4				
	CARB	I1, I2, I3, **bl**		**bcl**, I4				
	CACC		I1, I2, I3, **bl**		**bcl**, I4			
Dispatch	IQ3			**bl**				
	IQ2			I3		I4		
	IQ1			I2	I3,Tag**bl**	**bcl**	I4	
	IQ0			I1	I2	I3,Tag**bl**	Tag**bcl**	I4
IU	ID			I1	I2	I3,Tag**bl**	Tag**bcl**	I4
	IE				I1	I2	I3,Tag**bl**	Tag**bcl**
	IC					I1	I2	I3,Tag**bl**
	IWA					I1	I2	I3
	IWL							
	FPSB							
	ISB							
BPU	BE			**bl**		**bcl**		
	MR							
	BW				**bl**	**bl**	**bl**, **bcl**	**bl**, **bcl**
Notes				**bl** creates a link tag that is appended to I3		**bcl** cannot be tagged to I3 because it already has a link tag	The tag for **bcl** uses the ID stage in this cycle so I4 cannot be dispatched	

I.1.1.6 Conditional Branch Timing

The first set of tables in this section show timing when compare instructions are followed by dependent conditional branch instructions. There are separate tables for each of the four basic scenarios for the compare-branch instruction pair:

- Predict taken correctly
- Predict taken incorrectly
- Predict not-taken correctly
- Predict not-taken incorrectly

Within the following tables, sn is the n^{th} sequential instruction behind a branch, while tn is the n^{th} target instruction behind a branch.

Several code sequences are included to show the behavior of conditional branches in the 601 pipeline. Typically, each code sequence is followed by multiple tables, each of which shows a different branch processing scenario (e.g., taken/not-taken).

The first code sequence is a conditional instruction sequence (conditional move). In this code segment, whether an instruction occurs depends on the results of some other operation.

I.1.1.6.1 Conditional Operations—Case 1

The examples in this section illustrate the following code sequence:

```
start: cmp cr3 = r1,r3
       bc lab, cr3
       mtspr spr=r1
lab:   stw mem(r5,8)=r1
```

There are four possible timings for this sequence, depending on whether the branch is taken and on whether the prediction is correct.

Table I-10 shows the timing for the code sequence when the branch is taken and is predicted successfully.

Table I-10. Conditional Operation (Case 1)—Predict Taken Correctly

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Memory	FA	**cmp**, **bc**, **mtspr**, **stw**		**stw**				
	CARB	**cmp**, **bc**, **mtspr**, **stw**		**stw**			**stw**	
	CACC		**cmp**, **bc**, **mtspr**, **stw**		**stw**			**stw**
Dispatch	IQ3			**stw**				
	IQ2			**mtspr**				
	IQ1			**bc**	**stw**			
	IQ0			**cmp**	**mtspr**	**stw**		
IU	ID			**cmp**	**mtspr**	**stw**		
	IE				**cmp**, Tagpbr		**stw**	
	IC					**cmp**		**stw**
	IWA					**cmp**		
	IWL							
	FPSB							
	ISB							
BPU	BE			**bc**				
	MR			**bc**	**bc**			
	BW							
Notes				**bc** predicted (cannot be resolved)	**cmp** results forwarded to MR[**bc**]; prediction confirmed			

Table I-11 shows the timing for the code sequence when the branch is not taken and is predicted unsuccessfully.

Table I-11. Conditional Operation (Case 1)—Predict Taken Incorrectly

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Memory	FA	**cmp**, **bc**, **mtspr**, **stw**		**stw**				
	CARB	**cmp**, **bc**, **mtspr**, **stw**		**stw**			**stw**	
	CACC		**cmp**, **bc**, **mtspr**, **stw**		**stw**			**stw**
Dispatch	IQ3			**stw**				
	IQ2			**mtspr**				
	IQ1			**bc**	**stw**			
	IQ0			**cmp**	**mtspr**	**stw**		
IU	ID			**cmp**	**mtspr**	**stw**		
	IE				**cmp**, Tagpbr	**mtspr**	**stw**	
	IC					**cmp**	**mtspr**	**stw**
	IWA					**cmp**	**mtspr**	
	IWL							
	FPSB							
	ISB							
BPU	BE			**bc**				
	MR			**bc**	**bc**			
	BW							
Notes				**bc** predicted (cannot be resolved)	**cmp** results forwarded to MR[**bc**]; fetcher recovers from misprediction.			

Table I-12 shows the timing for the code sequence when the branch is not taken and is predicted successfully.

Table I-12. Conditional Operation (Case 1)—Predict Not-Taken Correctly

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Memory	FA	**cmp**, **bc**, **mtspr**, **stw**						
	CARB	**cmp**, **bc**, **mtspr**, **stw**					**stw**	
	CACC		**cmp**, **bc**, **mtspr**, **stw**					**stw**
Dispatch	IQ3			**stw**				
	IQ2			**mtspr**				
	IQ1			**bc**	**stw**			
	IQ0			**cmp**	**mtspr**	**stw**		
IU	ID			**cmp**	**mtspr**	**stw**		
	IE				**cmp**, Tagpbr	**mtspr**	**stw**	
	IC					**cmp**	**mtspr**	**stw**
	IWA					**cmp**	**mtspr**	
	IWL							
	FPSB							
	ISB							
BPU	BE			**bc**				
	MR			**bc**	**bc**			
	BW							
Notes				**bc** predicted (cannot be resolved)	**cmp** results forwarded to MR[**bc**]; prediction confirmed.			

Table I-13 shows the timing for the code sequence when the branch is taken and is predicted unsuccessfully.

Table I-13. Conditional Operation (Case 1)—Predict Not-Taken Incorrectly

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Memory	FA	**cmp**, **bc**, **mtspr**, **stw**			**stw**			
	CARB	**cmp**, **bc**, **mtspr**, **stw**						**stw**
	CACC		**cmp**, **bc**, **mtspr**, **stw**			**stw**		
Dispatch	IQ3			**stw**				
	IQ2			**mtspr**				
	IQ1			**bc**	**stw**			
	IQ0			**cmp**	**mtspr**		**stw**	
IU	ID			**cmp**	**mtspr**		**stw**	
	IE				**cmp**, Tagpbr			**stw**
	IC					**cmp**		
	IWA					**cmp**		
	IWL							
	FPSB							
	ISB							
BPU	BE			**bc**				
	MR			**bc**	**bc**			
	BW							
Notes				**bc** predicted (cannot be resolved)	**cmp** results forwarded to MR[**bc**]; fetcher recovers from misprediction.			

I.1.1.6.2 Conditional Operations—Case 2

The next four examples illustrate the following code sequence:

```
start: add. r1=r1+r3
       bc lab, cr0
       mtspr spr=r1
lab:   stw mem(r5,8)=r1
```

There are four possible timings for this sequence, depending on whether the branch is taken and on whether it was predicted correctly.

Table I-14 shows the timing for the code sequence when the branch is taken and is predicted successfully.

Table I-14. Conditional Operation (Case 2)—Predict Taken Correctly

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Memory	**FA**	**add., bc, mtspr, stw**		**st**				
	CARB	**add.**, **bc**, **mtspr**, **stw**		**st**			**stw**	
	CACC		**add.**, **bc**, **mtspr**, **stw**		**st**			**stw**
Dispatch	IQ3			**stw**				
	IQ2			**mtspr**				
	IQ1			**bc**	**stw**			
	IQ0			**add.**	**mtspr**	**stw**		
IU	ID			**add.**	**mtspr**	**stw**		
	IE				**add.**, Tagpbr	Tagpbr	**stw**	
	IC					**add.**		**stw**
	IWA					**add.**		
	IWL							
	FPSB							
	ISB							
BPU	BE			**bc**				
	MR			**bc**	**bc**	**bc**		
	BW							
Notes				**bc** predicted (cannot be resolved)		Prediction confirmed		

Table I-15 shows the timing for the code sequence when the branch is not taken and is predicted unsuccessfully.

Table I-15. Conditional Operation (Case 2)—Predict Taken Incorrectly

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Memory	FA	**add.**, **bc**, **mtspr**, **stw**		**st**		**mtspr**, **stw**		
	CARB	**add.**, **bc**, **mtspr**, **stw**		**st**		**mtspr**, **stw**		
	CACC		**add.**, **bc**, **mtspr**, **stw**		**st**		**mtspr**, **stw**	
Dispatch	IQ3			**stw**				
	IQ2			**mtspr**				
	IQ1			**bc**	**stw**			**stw**
	IQ0			**add.**	**mtspr**	**stw**		**mtspr**
IU	ID			**add.**	**mtspr**	**stw**		**mtspr**
	IE				**add.**, Tagpbr	Tagpbr		
	IC					**add.**		
	IWA					**add.**		
	IWL							
	FPSB							
	ISB							
BPU	BE			**bc**				
	MR			**bc**	**bc**			
	BW							
Notes				**bc** predicted (cannot be resolved)		Fetcher recovers from misprediction.		

Table I-16 shows the timing for the code sequence when the branch is not taken and is predicted successfully.

Table I-16. Conditional Operation (Case 2)—Predict Not-Taken Correctly

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Memory	FA	**add.**, **bc**, **mtspr**, **stw**						
	CARB	**add.**, **bc**, **mtspr**, **stw**						**stw**
	CACC		**add.**, **bc**, **mtspr**, **stw**					
Dispatch	IQ3			**stw**				
	IQ2			**mtspr**				
	IQ1			**bc**	**stw**	**stw**		
	IQ0			**add.**	**mtspr**	**mtspr**	**stw**	
IU	ID			**add.**	**mtspr**	**mtspr**	**stw**	
	IE				**add.**, Tagpbr	Tagpbr	**mtspr**	**stw**
	IC					**add.**		**mtspr**
	IWA					**add.**		**mtspr**
	IWL							
	FPSB							
	ISB							
BPU	BE			**bc**				
	MR			**bc**	**bc**	**bc**		
	BW							
Notes				**bc** predicted (cannot be resolved)		Prediction confirmed		

Table I-17 shows the timing for the code sequence when the branch is taken but is predicted unsuccessfully.

Table I-17. Conditional Operation (Case 2)—Predict Not-Taken Incorrectly

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Memory	FA	**add.**, **bc**, **mtspr**, **stw**				**stw**		
	CARB	**add.**, **bc**, **mtspr**, **stw**				**stw**		
	CACC		**add.**, **bc**, **mtspr**, **stw**				**stw**	
Dispatch	IQ3			**stw**				
	IQ2			**mtspr**				
	IQ1			**bc**	**stw**	**stw**		
	IQ0			**add.**	**mtspr**	**mtspr**		**stw**
IU	ID			**add.**	**mtspr**	**mtspr**		**stw**
	IE				**add.**, Tagpbr	Tagpbr		
	IC					**add.**		
	IWA					**add.**		
	IWL							
	FPSB							
	ISB							
BPU	BE			**bc**				
	MR			**bc**	**bc**	**bc**		
	BW							
Notes				**bc** predicted (cannot be resolved)		Fetcher recovers from misprediction.		

I.1.1.6.3 Conditional Operations—Case 3

The examples in this section illustrate the following code sequence:

```
start: mull. r1=r1*r3
       bc lab, cr0
       mtspr spr=r1
lab:   stw mem(r5,8)=r1
```

There are four possible timings for this sequence, depending on whether the branch is taken and on whether it is predicted correctly.

Table I-18 shows the timing for the code sequence when the branch is taken and is predicted successfully.

Table I-18. Conditional Operation (Case 3)—Predict Taken Correctly

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5-8**	**9**	**10**
Memory	FA	**mull.**, **bc**, **mtspr**, **stw**		**stw**				
	CARB	**mull.**, **bc**, **mtspr**, **stw**		**stw**				**stw**
	CACC		**mull.**, **bc**, **mtspr**, **stw**		**stw**			
Dispatch	IQ7							
	IQ6							
	IQ5							
	IQ4							
	IQ3			**stw**				
	IQ2			**mtspr**				
	IQ1			**bc**	**stw**			
	IQ0			**mull.**	**mtspr**	**stw**	**stw**	
IU	ID			**mull.**	**mtspr**	**stw**	**stw**	
	IE				**mull.**, Tagpbr	**mull.**, Tagpbr	Tagpbr	**stw**
	IC						**mull.**	
	IWA						**mull.**	
	IWL							
	FPSB							
	ISB							
BPU	BE			**bc**				
	MR			**bc**	**bc**	**bc**	**bc**	
	BW							
Notes				**bc** predicted (cannot be resolved)			Prediction confirmed	

Table I-19 shows the timing for the code sequence when the branch is not taken and is predicted unsuccessfully.

Table I-19. Conditional Operation (Case 3)—Predict Taken Incorrectly

Pipeline Stage		Cycle Number					
Unit	**Stage**	**1**	**2**	**3**	**4**	**5-8**	**9**
Memory	FA	**mull.**, **bc**, **mtspr**, **stw**		**st**			**mtspr**, **stw**
	CARB	**mull.**, **bc**, **mtspr**, **stw**		**st**			**mtspr**, **stw**
	CACC		**mull.**, **bc**, **mtspr**, **stw**		**st**		
Dispatch	IQ3			**stw**			
	IQ2			**mtspr**			
	IQ1			**bc**	**stw**		
	IQ0			**mull.**	**mtspr**	**stw**	**stw**
IU	ID			**mull.**	**mtspr**	**stw**	**stw**
	IE				**mull.**, Tagpbr	**mull.**, Tagpbr	Tagpbr
	IC						**mull.**
	IWA						**mull.**
	IWL						
	FPSB						
	ISB						
BPU	BE			**bc**			
	MR			**bc**	**bc**	**bc**	**bc**
	BW						
Notes				**bc** predicted (cannot be resolved)		Fetcher recovers from misprediction.	Fetcher recovers from misprediction.

Table I-19. Conditional Operation (Case 3)—Predict Taken Incorrectly (Continued)

Pipeline Stage		Cycle Number				
Unit	**Stage**	**10**	**11**	**12**	**13**	**14**
Memory	FA					
	CARB				**stw**	
	CACC	**mtspr**, **stw**				**stw**
Dispatch	IQ3					
	IQ2					
	IQ1		**stw**			
	IQ0		**mtspr**	**stw**		
IU	ID		**mtspr**	**stw**		
	IE			**mtspr**	**stw**	
	IC				**mtspr**	**stw**
	IWA				**mtspr**	
	IWL					
	FPSB					
	ISB					
BPU	BE					
	MR					
	BW					
Notes						

Table I-20 shows the timing for the code sequence when the branch is not taken and is predicted successfully.

Table I-20. Conditional Operation (Case 3)—Predict Not-Taken Correctly

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5-8**	**9**	**10**
Memory	FA	**mull**., **bc**, **mtspr**, **stw**						
	CARB	**mull**., **bc**, **mtspr**, **stw**						
	CACC		**mull**., **bc**, **mtspr**, **stw**					
Dispatch	IQ7							
	IQ6							
	IQ5							
	IQ4							
	IQ3			**stw**				
	IQ2			**mtspr**				
	IQ1			**bc**	**stw**	**stw**	**stw**	
	IQ0			**mull**.	**mtspr**	**mtspr**	**mtspr**	**stw**
IU	ID			**mull**.	**mtspr**	**mtspr**	**mtspr**	**stw**
	IE				**mull**., Tagpbr	**mull**., Tagpbr	Tagpbr	**mtspr**
	IC						**mull**.	
	IWA						**mull**.	
	IWL							
	FPSB							
	ISB							
BPU	BE			**bc**				
	MR			**bc**	**bc**	**bc**	**bc**	
	BW							
Notes				**bc** predicted (cannot be resolved)			Prediction confirmed	

Table I-21 shows the timing for the code sequence when the branch is not taken and is predicted unsuccessfully.

Table I-21. Conditional Operation (Case 3)—Predict Not-Taken Incorrectly

Pipeline Stage		Cycle Number						
Unit	Stage	1	2	3	4	5-8	9	10
Memory	FA	**mull**., **bc, mtspr**, **stw**					**stw**	
	CARB	**mull**., **bc, mtspr**, **stw**					**stw**	
	CACC		**mull**., **bc, mtspr**, **stw**					**stw**
Dispatch	IQ3			**stw**				
	IQ2			**mtspr**				
	IQ1			**bc**	**stw**	**stw**	**stw**	
	IQ0			**mull**.	**mtspr**	**mtspr**	**mtspr**	
IU	ID			**mull**.	**mtspr**	**mtspr**	**mtspr**	
	IE				**mull**., Tagpbr	**mull**., Tagpbr	Tagpbr	
	IC						**mull**.	
	IWA						**mull**.	
	IWL							
	FPSB							
	ISB							
BPU	BE			**bc**				
	MR			**bc**	**bc**	**bc**	**bc**	
	BW							
Notes				**bc** predicted (cannot be resolved)			Fetcher recovers from misprediction.	

Table I-21. Conditional Operation Case 3—Predict Not-Taken Incorrectly (Continued)

Pipeline Stage		Cycle Number		
Unit	**Stage**	**11**	**12**	**13**
Memory	FA			
	CARB		**stw**	
	CACC			**stw**
Dispatch	IQ0	**stw**		
IU	ID	**stw**		
	IE		**stw**	
	IC			**stw**
	IWA			
	IWL			
	FPSB			
	ISB			
BPU	BE			
	MR			
	BW			
Notes				

I.1.1.6.4 Conditional Operations—Case 4

The next four examples illustrate the following code sequence:

```
start: fmadd. f1=(f0*f1)+f3
       bc lab, cr1
       fmr f4=f1
lab:   stfd mem(r5,8)=f1
```

There are four possible timings for this sequence, depending on whether the branch is taken and on whether it was predicted correctly.

Table I-22 shows the timing for the code sequence when the branch is taken and is predicted successfully.

Table I-22. Conditional Operation (Case 4)—Predict Taken Correctly

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Memory	FA	**fmadd.**, **bc**, **fmr**, **stfd**		**stfd**				
	CARB	**fmadd.**, **bc**, **fmr**, **stfd**		**stfd**				
	CACC		**fmadd.**, **bc**, **fmr**, **stfd**		**stfd**			
Dispatch	IQ3			**stfd**				
	IQ2			**fmr**				
	IQ1			**bc**	**stfd**			
	IQ0			**fmadd.**	**fmr**	**stfd**	**stfd**	**stfd**, Tag**stfd**
IU	ID					**stfd**	**stfd**	**stfd**, Tag**stfd**
	IE				Tagpbr, Tag**fmadd**	Tagpbr	Tagpbr	Tagpbr
	IC					Tag**fmadd**	Tag**fmadd**	Tag**fmadd**
	IWA							
	IWL							
	FPSB							
	ISB							
BPU	BE			**bc**				
	MR			**bc**	**bc**	**bc**	**bc**	**bc**
	BW							
FPU	F1					**fmr**		
	FD				**fmadd.**	**fmadd.**		**stfd**
	FPM					**fmadd.**	**fmadd.**	
	FPA						**fmadd.**	**fmadd.**
	FWA							
	FWL							
Notes				**bc** predicted (cannot be resolved)		F1 is full, so **stfd** cannot dispatch		

Table I-22. Conditional Operation (Case 4)—Predict Taken Correctly (Continued)

Pipeline Stage		Cycle Number						
Unit	**Stage**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
Memory	FA							
	CARB						**stfd**	
	CACC							**stfd**
Dispatch	IQ3							
	IQ2							
	IQ1							
	IQ0	**stfd**, Tag**stfd**	**stfd**, Tag**stfd**	**stfd**, Tag**stfd**				
IU	ID	**stfd**, Tag**stfd**	**stfd**, Tag**stfd**	**stfd**, Tag**stfd**				
	IE	Tagpbr	Tagpbr		**stfd**, Tag**stfd**			
	IC	Tag**fmadd**				**stfd**, Tag**stfd**		
	IWA							
	IWL							
	FPSB					**stfd**	**stfd**	**stfd**
	ISB							
BPU	BE							
	MR	**bc**	**bc**					
	BW							
FPU	F1							
	FD	**stfd**	**stfd**					
	FPM			**stfd**				
	FPA				**stfd**			
	FWA	**fmadd.**				**stfd**	**stfd**	
	FWL							
Notes		**stfd** is stalled due to the dependency on the **fmadd.**	Branch prediction confirmed.					

Table I-23 shows the timing for the code sequence when the branch is not taken and is predicted unsuccessfully.

Table I-23. Conditional Operation (Case 4)—Predict Taken Incorrectly

Pipeline Stage		Cycle Number					
Unit	**stage**	**1**	**2**	**3**	**4**	**5**	**6**
Memory	FA	**fmadd.**, **bc**, **fmr**, **stfd**		**stfd**			
	CARB	**fmadd.**, **bc**, **fmr**, **stfd**		**stfd**			
	CACC		**fmadd.**, **bc**, **fmr**, **stfd**		**stfd**		
Dispatch	IQ3			**stfd**			
	IQ2			**fmr**			
	IQ1			**bc**	**stfd**		
	IQ0			**fmadd.**	**fmr**	**stfd**	**stfd**
IU	ID					**stfd**	**stfd**
	IE				Tagpbr, Tag**fmadd**	Tagpbr	Tagpbr
	IC					Tag**fmadd**	Tag**fmadd**
	IWA						
	IWL						
	FPSB						
	ISB						
BPU	BE			**bc**			
	MR			**bc**	**bc**	**bc**	**bc**
	BW						
FPU	F1					**fmr**	
	FD				**fmadd.**	**fmadd.**	
	FPM					**fmadd.**	**fmadd.**
	FPA						**fmadd.**
	FWA						
	FWL						
Notes				**bc** predicted (cannot be resolved)		F! is full, so **stfd** cannot be dispatched.	

Table I-23. Conditional Operation (Case 4)—Predict Taken Incorrectly (Continued)

Pipeline Stage		Cycle Number					
Unit	**Stage**	**7**	**8**	**9**	**10**	**11**	**12**
Memory	FA			**fmr**, **stfd**			
	CARB			**fmr**, **stfd**			
	CACC				**fmr**, **stfd**		
Dispatch	IQ3						
	IQ2						
	IQ1					**stfd**	
	IQ0	**stfd**, Tag**stfd**	**stfd**, Tag**stfd**	**stfd**, Tag**stfd**		**fmr**	**stfd**
IU	ID	**stfd**, Tag**stfd**	**stfd**, Tag**stfd**	**stfd**, Tag**stfd**			**stfd**
	IE	Tagpbr	Tagpbr	Tagpbr			Tag**fmr**
	IC	Tag**fmadd**	Tag**fmadd**				
	IWA						
	IWL						
	FPSB						
	ISB						
BPU	BE						
	MR	**bc**	**bc**	**bc**			
	BW						
FPU	F1						
	FD	**stfd**	**stfd**	**stfd**			**fmr**
	FPM				**stfd**		
	FPA	**fmadd.**					
	FWA		**fmadd.**				
	FWL						
Notes			**stfd** is stalled due to the dependency on the **fmadd.**	BPU recovers from mispredicted branch			

Table I-23. Conditional Operation (Case 4)—Predict Taken Incorrectly (Continued)

Pipeline Stage		Cycle Number				
Unit	**Stage**	**13**	**14**	**15**	**16**	**17**
Memory	FA					
	CARB				**stfd**	
	CACC					**stfd**
Dispatch	IQ3					
	IQ2					
	IQ1					
	IQ0					
IU	ID					
	IE	**stfd**, Tag**stfd**				
	IC	Tag**fmr**	**stfd**, Tag**stfd**			
	IWA					
	IWL					
	FPSB		**stfd**	**stfd**	**stfd**	
	ISB					
BPU	BE					
	MR					
	BW					
FPU	F1					
	FD	**stfd**				
	FPM	**fmr**	**stfd**			
	FPA		**fmr**	**stfd**		
	FWA			**fmr**	**stfd**	
	FWL					
Notes						

Table I-24 shows the timing for the code sequence when the branch is not taken and is predicted successfully.

Table I-24. Conditional Operation (Case 4)—Predict Not-Taken Correctly

Pipeline Stage		Cycle Number					
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**
Memory	FA	**fmadd.**, **bc**, **fmr**, **stfd**					
	CARB	**fmadd.**, **bc**, **fmr**, **stfd**					
	CACC		**fmadd.**, **bc**, **fmr**, **stfd**				
Dispatch	IQ3			**stfd**			
	IQ2			**fmr**			
	IQ1			**bc**	**stfd**		
	IQ0			**fmadd.**	**fmr**	**stfd**	**stfd**
IU	ID					Tag**fmr**	Tag**fmr**
	IE				Tagpbr, Tag**fmadd**	Tagpbr	Tagpbr
	IC					Tag**fmadd**	Tag**fmadd**
	IWA						
	IWL						
	FPSB						
	ISB						
BPU	BE			**bc**			
	MR			**bc**	**bc**	**bc**	**bc**
	BW						
FPU	F1					**fmr**	
	FD				**fmadd.**	**fmadd.**	**fmr**
	FPM					**fmadd.**	**fmadd.**
	FPA						**fmadd.**
	FWA						
	FWL						
Notes				**bc** predicted (cannot be resolved)		F1 is full, so **stfd** cannot be dispatched.	

Table I-24. Conditional Operation (Case 4)—Predict Not-Taken Correctly (Continued)

Pipeline Stage		Cycle Number					
Unit	**Stage**	**7**	**8**	**9**	**10**	**11**	**12**
Memory	FA						
	CARB						
	CACC						
Dispatch	IQ3						
	IQ2						
	IQ1						
	IQ0	**stfd**, Tag**stfd**	**stfd**, Tag**stfd**	**stfd**, Tag**stfd**	**stfd**, Tag**stfd**		
IU	ID	Tag**fmr**	Tag**fmr**	Tag**fmr**	Tag**fmr**	**stfd**, Tag**stfd**	
	IE	Tagpbr	Tagpbr	Tagpbr		Tag**fmr**	**stfd**, Tag**stfd**
	IC	Tag**fmadd**	Tag**fmadd**				Tag**fmr**
	IWA						
	IWL						
	FPSB						**stfd**
	ISB						
BPU	BE						
	MR	**bc**	**bc**	**bc**			
	BW						
FPU	F1						
	FD	**stfd**	**stfd**	**stfd**	**stfd**	**stfd**	**stfd**
	FPM	**fmr**					
	FPA	**fmadd.**	**fmr**				
	FWA		**fmadd.**	**fmr**	**fmr**	**fmr**	**fmr**
	FWL						
Notes		**stfd** stalls due to a RAW dependency on **fmadd.**		Branch prediction is confirmed.	**fmr** cannot leave FW until its tag completes.		

Table I-24. Conditional Operation (Case 4)—Predict Not-Taken Correctly (Continued)

Pipeline Stage		Cycle Number				
Unit	**Stage**	**13**	**14**	**15**	**16**	**17**
Memory	FA					
	CARB					**stfd**
	CACC					
Dispatch	IQ3					
	IQ2					
	IQ1					
	IQ0					
IU	ID					
	IE					
	IC	**stfd**, Tag**stfd**				
	IWA					
	IWL					
	FPSB	**stfd**	**stfd**	**stfd**	**stfd**	**stfd**
	ISB					
BPU	BE					
	MR					
	BW					
FPU	F1					
	FD	**stfd**				
	FPM		**stfd**			
	FPA			**stfd**		
	FWA	**fmr**			**stfd**	
	FWL					
Notes						

Table I-25 shows the timing for the code sequence when the branch is not taken and is predicted unsuccessfully.

Table I-25. Conditional Operation (Case 4)—Predict Not-Taken Incorrectly

Pipeline Stage		Cycle Number					
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**
Memory	FA	**fmadd.**, **bc**, **fmr**, **stfd**					
	CARB	**fmadd.**, **bc**, **fmr**, **stfd**					
	CACC		**fmadd.**, **bc**, **fmr**, **stfd**				
Dispatch	IQ3			**stfd**			
	IQ2			**fmr**			
	IQ1			**bc**	**stfd**		
	IQ0			**fmadd.**	**fmr**	**stfd**	**stfd**
IU	ID					Tag**fmr**	Tag**fmr**
	IE				Tagpbr, Tag**fmadd**	Tagpbr	Tagpbr
	IC					Tag**fmadd**	Tag**fmadd**
	IWA						
	IWL						
	FPSB						
	ISB						
BPU	BE			**bc**			
	MR			**bc**	**bc**	**bc**	**bc**
	BW						
FPU	F1					**fmr**	
	FD				**fmadd.**	**fmadd.**	**fmr**
	FPM					**fmadd.**	**fmadd.**
	FPA						**fmadd.**
	FWA						
	FWL						
Notes				**bc** predicted (cannot be resolved)		F1 is full, so **stfd** cannot be dispatched.	

Table I-25. Conditional Operation (Case 4)—Predict Not-Taken Incorrectly (Continued)

Pipeline Stage		Cycle Number					
Unit	**Stage**	**7**	**8**	**9**	**10**	**11**	**12**
Memory	FA			**stfd**			
	CARB			**stfd**			
	CACC				**stfd**		
Dispatch	IQ3						
	IQ2						
	IQ2						
	IQ0	**stfd**, Tag**stfd**	**stfd**, Tag**stfd**	**stfd**, Tag**stfd**		**stfd**	
IU	ID	Tag**fmr**	Tag**fmr**	Tag**fmr**		**stfd**	
	IE	Tagpbr	Tagpbr	Tagpbr			**stfd**, Tag**stfd**
	IC	Tag**fmadd**	Tag**fmadd**				
	IWA						
	IWL						
	FPSB						
	ISB						
BPU	BE						
	MR	**bc**	**bc**	**bc**			
	BW						
FPU	F1						
	FD	**stfd**	**stfd**	**stfd**	**stfd**		**stfd**
	FPM	**fmr**					
	FPA	**fmadd.**	**fmr**				
	FWA		**fmadd.**	**fmr**	**fmr**		
	FWL						
Notes		**stfd** stalls due to a RAW dependency on the **fmadd.**		BPU recovers from mispredicted branch.			

Table I-25. Conditional Operation (Case 4)—Predict Not-Taken Incorrectly (Continued)

Pipeline Stage		Cycle Number			
Unit	**Stage**	**13**	**14**	**15**	**16**
Memory	FA				
	CARB			**stfd**	
	CACC				**stfd**
Dispatch	IQ3				
	IQ2				
	IQ1				
	IQ0				
IU	ID				
	IE				
	IC	**stfd**, Tag**stfd**			
	IWA				
	IWL				
	FPSB	**stfd**	**stfd**	**stfd**	
	ISB				
BPU	BE				
	MR				
	BW				
FPU	F1				
	FD				
	FPM	**stfd**			
	FPA		**stfd**		
	FWA			**stfd**	
	FWL				
Notes					

I.1.1.6.5 Conditional Operations—Case 5

The next four examples illustrate the following code sequence. There are four possible timings for this sequence, depending on whether the branch is taken and on whether the prediction is correct.

```
start: fmadds. f1=(f0*f1)+f3
       bc lab, cr1
```

```
        fmr f4=f1
lab:    stfd mem(r5,8)=f1
```

Table I-26 shows the timing for the code sequence when the branch is taken and is predicted successfully.

Table I-26. Conditional Operation (Case 5)—Predict Taken Correctly

Pipeline Stage		Cycle Number					
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**
Memory	FA	**fmadds.**, **bc**, **fmr**, **stfs**		**stfs**			
	CARB	**fmadds.**, **bc**, **fmr**, **stfs**		**stfs**			
	CACC		**fmadds.**, **bc**, **fmr**, **stfs**		**stfs**		
Dispatch	IQ3			**stfs**			
	IQ2			**fmr**			
	IQ1			**bc**	**stfs**		
	IQ0			**fmadds.**	**fmr**	**stfs**	**stfs**,Tag**stfs**
IU	ID					**stfs**	**stfs**,Tag**stfs**
	IE				Tagpbr, Tag**fmadds**	Tagpbr	Tagpbr
	IC					Tag**fmadds**	Tag**fmadds**
	IWA						
	IWL						
	FPSB						
	ISB						
BPU	BE			**bc**			
	MR			**bc**	**bc**	**bc**	**bc**
	BW						
FPU	F1						
	FD				**fmadds.**	**fmr**	**stfs**
	FPM					**fmadds.**	
	FPA						**fmadds.**
	FWA						
	FWL						
Notes				**bc** predicted (cannot be resolved)			

Table I-26. Conditional Operation (Case 5)—Predict Taken Correctly (Continued)

Pipeline Stage		Cycle Number						
Unit	**Stage**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
Memory	FA							
	CARB						**stfs**	
	CACC							**stfs**
Dispatch	IQ3							
	IQ2							
	IQ1							
	IQ0	**stfs**, Tag**stfs**	**stfs**, Tag**stfs**	**stfs**, Tag**stfs**				
IU	ID	**stfs**, Tag**stfs**	**stfs**, Tag**stfs**	**stfs**, Tag**stfs**				
	IE	Tagpbr	Tagpbr		**stfs**, Tag**stfs**			
	IC	Tag**fmadds**				**stfs**, Tag**stfs**		
	IWA							
	IWL							
	FPSB					**stfs**	**stfs**	
	ISB							
BPU	BE							
	MR	**bc**	**bc**					
	BW							
FPU	F1							
	FD	**stfs**	**stfs**					
	FPM			**stfs**				
	FPA				**stfs**			
	FWA	**fmadds.**				**stfs**		
	FWL							
Notes		**stfs** stalls if FD because of the RAW hazard with the **fmadds**.	Branch prediction confirmed.					

Table I-27 shows the timing for the code sequence when the branch is taken and is predicted unsuccessfully.

Table I-27. Conditional Operation (Case 5)—Predict Taken Incorrectly

Pipeline Stage		Cycle Number					
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**
Memory	FA	**fmadds.**, **bc**, **fmr**, **stfs**		**stfs**			
	CARB	**fmadds.**, **bc**, **fmr**, **stfs**		**stfs**			
	CACC		**fmadds.**, **bc**, **fmr**, **stfs**		**stfs**		
Dispatch	IQ3			**stfs**			
	IQ2			**fmr**			
	IQ1			**bc**	**stfs**		
	IQ0			**fmadds.**	**fmr**	**stfs**	**stfs**, Tag**stfs**
IU	ID					**stfs**	**stfs**, Tag**stfs**
	IE				Tagpbr, Tag**fmadds**	Tagpbr	Tagpbr
	IC					Tag**fmadds**	Tag**fmadds**
	IWA						
	IWL						
	FPSB						
	ISB						
BPU	BE			**bc**			
	MR			**bc**	**bc**	**bc**	**bc**
	BW						
FPU	F1						
	FD				**fmadds.**	**fmr**	**stfs**
	FPM					**fmadds.**	
	FPA						**fmadds.**
	FWA						
	FWL						
Notes				**bc** predicted (cannot be resolved)			

Table I-27. Conditional Operation Case 5—Predict Taken Incorrectly (Continued)

Pipeline Stage		Cycle Number					
Unit	**Stage**	**7**	**8**	**9**	**10**	**11**	**12**
Memory	FA		**fmr**, **stfs**				
	CARB		**fmr**, **stfs**				
	CACC			**fmr**, **stfs**			
Dispatch	IQ3						
	IQ2						
	IQ1				**stfs**		
	IQ0	**stfs**, Tag**stfs**	**stfs**, Tag**stfs**		**fmr**	**stfs**	
IU	ID	**stfs**, Tag**stfs**	**stfs**, Tag**stfs**			**stfs**	
	IE	Tagpbr	Tagpbr			Tag**fmr**	**stfs**, Tag**stfs**
	IC	Tag**fmadds**					Tag**fmr**
	IWA						
	IWL						
	FPSB						
	ISB						
BPU	BE						
	MR	**bc**	**bc**				
	BW						
FPU	F1						
	FD	**stfs**	**stfs**			**fmr**	**stfs**
	FPM			**stfs**			**fmr**
	FPA						
	FWA	**fmadds.**					
	FWL						
Notes		**stfs** stalls if FD because of the RAW hazard with the **fmadds**.	BPU recovers from branch misprediction.				

Table I-27. Conditional Operation Case 5—Predict Taken Incorrectly (Continued)

Pipeline Stage		Cycle Number			
Unit	**Stage**	**13**	**14**	**15**	**16**
Memory	FA				
	CARB			**stfs**	
	CACC				**stfs**
Dispatch	IQ3				
	IQ2				
	IQ1				
	IQ0				
IU	ID				
	IE				
	IC	**stfs**, Tag**stfs**			
	IWA				
	IWL				
	FPSB	**stfs**	**stfs**	**stfs**	
	ISB				
BPU	BE				
	MR				
	BW				
FPU	F1				
	FD				
	FPM	**stfs**			
	FPA	**fmr**	**stfs**		
	FWA		**fmr**	**stfs**	
	FWL				
Notes					

Table I-28 shows the timing for the code sequence when the branch is not taken and is predicted successfully.

Table I-28. Conditional Operation (Case 5)—Predict Not-Taken Correctly

Pipeline Stage		Cycle Number				
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**
Memory	FA	**fmadds.**, **bc**, **fmr**, **stfs**		**stfs**		
	CARB	**fmadds.**, **bc**, **fmr**, **stfs**		**stfs**		
	CACC		**fmadds.**, **bc**, **fmr**, **stfs**		**stfs**	
Dispatch	IQ3			**stfs**		
	IQ2			**fmr**		
	IQ1			**bc**	**stfs**	
	IQ0			**fmadds.**	**fmr**	**stfs**
IU	ID					Tag**fmr**
	IE				Tagpbr, Tag**fmadds**	Tagpbr
	IC					Tag**fmadds**
	IWA					
	IWL					
	FPSB					
	ISB					
BPU	BE			**bc**		
	MR			**bc**	**bc**	**bc**
	BW					
FPU	F1					
	FD				**fmadds.**	**fmr**
	FPM					**fmadds.**
	FPA					
	FWA					
	FWL					
Notes				**bc** predicted (cannot be resolved)		

Table I-28. Conditional Operation (Case 5)—Predict Not-Taken Correctly (Continued)

Pipeline Stage		Cycle Number					
Unit	**Stage**	**6**	**7**	**8**	**9**	**10**	**11**
Memory	FA						
	CARB						
	CACC						
Dispatch	IQ3						
	IQ2						
	IQ1						
	IQ0	**stfs**, Tag**stfs**	**stfs**, Tag**stfs**	**stfs**, Tag**stfs**	**stfs**, Tag**stfs**		
IU	ID	Tag**fmr**	Tag**fmr**	Tag**fmr**	Tag**fmr**	**stfs**, Tag**stfs**	
	IE	Tagpbr	Tagpbr	Tagpbr		Tag**fmr**	**stfs**, Tag**stfs**
	IC	Tag**fmadds**	Tag**fmadds**				Tag**fmr**
	IWA						
	IWL						
	FPSB						
	ISB						
BPU	BE						
	MR	**bc**	**bc**	**bc**			
	BW						
FPU	F1						
	FD	**stfs**	**stfs**	**stfs**	**stfs**	**stfs**	**stfs**
	FPM	**fmr**					
	FPA	**fmadds.**	**fmr**				
	FWA		**fmadds.**	**fmr**	**fmr**	**fmr**	**fmr**
	FWL						
Notes			**stfs** stalls if FD because of the RAW dependency with the **fmadds.**	The branch prediction is confirmed	**fmr** cannot leave FW until its tag completes		

Table I-28. Conditional Operation (Case 5)—Predict Not-Taken Correctly (Continued)

Pipeline Stage		Cycle Number				
Unit	**Stage**	**12**	**13**	**14**	**15**	**16**
Memory	FA					
	CARB				**stfs**	
	CACC					**stfs**
Dispatch	IQ3					
	IQ2					
	IQ1					
	IQ0					
IU	ID					
	IE					
	IC					
	IWA					
	IWL					
	FPSB	**stfs**	**stfs**	**stfs**	**stfs**	**stfs**
	ISB					
BPU	BE					
	MR					
	BW					
FPU	F1					
	FD	**stfs**				
	FPM		**stfs**			
	FPA			**stfs**		
	FWA	**fmr**			**stfs**	
	FWL					
Notes						

Table I-29 shows the timing for the code sequence when the branch is taken and is predicted unsuccessfully.

Table I-29. Conditional Operation (Case 5)—Predict Not-Taken Incorrectly

Pipeline Stage		Cycle Number					
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**
Memory	FA	**fmadds.**, **bc**, **fmr**, **stfs**		**stfs**			
	CARB	**fmadds.**, **bc**, **fmr**, **stfs**		**stfs**			
	CACC		**fmadds.**, **bc**, **fmr**, **stfs**		**stfs**		
Dispatch	IQ3			**stfs**			
	IQ2			**fmr**			
	IQ1			**bc**	**stfs**		
	IQ0			**fmadds.**	**fmr**	**stfs**, Tag**stfs**	**stfs**
IU	ID					Tag**fmr**	Tag**fmr**
	IE				Tagpbr, Tag**fmadds**	Tagpbr	Tagpbr
	IC					Tag**fmadds**	Tag**fmadds**
	IWA						
	IWL						
	FPSB						
	ISB						
BPU	BE			**bc**			
	MR			**bc**	**bc**	**bc**	**bc**
	BW						
FPU	F1						
	FD				**fmadds.**	**fmr**	**stfs**
	FPM					**fmadds.**	**fmr**
	FPA						**fmadds.**
	FWA						
	FWL						
Notes				**bc** predicted (cannot be resolved)			

Table I-29. Conditional Operation (Case 5)—Predict Not-Taken Incorrectly (Continued)

Pipeline Stage		Cycle Number								
Unit	**Stage**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Memory	FA		**stfs**							
	CARB		**stfs**						**stfs**	
	CACC			**stfs**						**stfs**
Dispatch	IQ3									
	IQ2									
	IQ1									
	IQ0	**stfs,** Tag**stfs**	**stfs**, Tag**stfs**		**stfs**					
IU	ID	Tag**fmr**	Tag**fmr**		**stfs**					
	IE	Tagpbr	Tagpbr			**stfs**, Tag**stfs**				
	IC	Tag**fmadds**					**stfs**, Tag**stfs**			
	IWA									
	IWL									
	FPSB						**stfs**	**stfs**	**stfs**	
	ISB									
BPU	BE									
	MR	**bc**	**bc**							
	BW									
FPU	F1									
	FD	**stfs**	**stfs**	**stfs**		**stfs**				
	FPM						**stfs**			
	FPA	**fmr**						**stfs**		
	FWA	**fmadds.**	**fmr**	**fmr**					**stfs**	
	FWL									
Notes		**stfs** stalls if FD because of the RAW dependency with the **fmadds.**	BPU recovers from incorrect prediction.							

I.1.1.7 Conditional Loop Closure Examples

The examples in this section show the timings for code sequences that perform conditional loop closure. In the tables illustrating this type of code, the loop is run until the pipeline behavior stabilizes, at which point the loop is exited. These examples show the behavior of the pipeline in three different conditions—loop start-up, stable looping code, and loop completion.

I.1.1.7.1 Conditional Loop Closure—Case 1

The tables in this section show the timings for the following code sequence:

```
start:  add r1=r1+r3
        cmp cr3=r1,r4
        bc start, cr3
exit:   add r1=r4,r3
```

Table I-30 shows the timing when the conditional branch is predicted as taken. Note that in this example, (op), which begin appearing in clock cycle 10, are the instructions along sequential path beyond the second add (**add**2) instruction. These instructions are shown here because the sequential path is executed.

Table I-30. Conditional Loop Closure (Case 1)—Predict Taken

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Memory	FA	start:0		start:1			start:2	
	CARB	**add**1, **cmp**, **bc**,**add**2		**add**1, **cmp**, **bc**,**add**2			**add**1, **cmp**, **bc**,**add**2	
	CACC		**add**1, **cmp**, **bc**,**add**2		**add**1, **cmp**, **bc**,**add**2			**add**1, **cmp**, **bc**,**add**2
Dispatch	IQ3			**add**2		**add**2		
	IQ2			**bc**		**bc**	**add**2	
	IQ1			**cmp**	**add**2	**cmp**	**bc**	
	IQ0			**add**1	**cmp**, Tagpbr	**add**1	**cmp**	**add**2
IU	ID			**add**1	**cmp**, Tagpbr	**add**1	**cmp**	**add**2
	IE				**add**1	**cmp**, Tagpbr	**add**1	**cmp**, Tagpbr
	IC					**add**1	**cmp**	**add**1
	IWA					**add**1	**cmp**	**add**1
	IWL							
	FPSB							
	ISB							
BPU	BE			**bc**			**bc**	
	MR			**bc**	**bc**	**bc**	**bc**	**bc**
	BW							
Notes				**bc** predicted taken.		**cmp** result is forwarded to MR; **bc** resolved: predicted correctly; **bc** in IQ2 stalls because MR is busy.	**bc** predicted taken.	**cmp** result is forwarded to MR; **bc** resolved: predicted correctly.

Table I-30. Conditional Loop Closure (Case 1)—Predict Taken (Continued)

Pipeline Stage		Cycle Number					
Unit	**Stage**	**8**	**9**	**10**	**11**	**12**	**13**
Memory	FA	start:3	Exit+5(*4)	Exit+6(*4)	start:4	Exit+8(*4)	
	CARB	**add**1,**cmp**, **bc**,**add**2	Exit+5(*4)	Exit+6(*4)	**add**1,**cmp**, **bc**,**add**2	Exit+8(*4)	
	CACC		**add**1,**cmp**, **bc**,**add**2	Exit+5(*4)	Exit+6(*4)	**add**1,**cmp**, **bc**,**add**2	Exit+8(*4)
Dispatch	IQ7			(op)	(op)	(op)	
	IQ6			(op)	(op)	(op)	(op)
	IQ5			(op)	(op)	(op)	(op)
	IQ4			(op)	(op)	(op)	(op)
	IQ3	**add**2		**add**2	(op)	(op)	(op)
	IQ2	**bc**		**bc**	**add**2	(op)	(op)
	IQ1	**cmp**	**add**2	**cmp**	**bc**	(op)	(op)
	IQ0	**add**1	**cmp**, Tagpbr	**add**1	**cmp**	**add**2	(op)
IU	ID	**add**1	**cmp**, Tagpbr	**add**1	**cmp**	**add**2	(op)
	IE		**add**1	**cmp**, Tagpbr	**add**1	**cmp**, Tagpbr	**add**2
	IC	**cmp**			**cmp**		**cmp**
	IWA	**cmp**			**cmp**		**cmp**
	IWL						
	FPSB						
	ISB						
BPU	BE	**bc**			**bc**		
	MR	**bc**	**bc**	**bc**	**bc**		
	BW						
Notes		**bc** predicted taken.	State in this cycle is the same as the state in cycle 4. (2.5 cycle/ loop)	**cmp** result is forwarded to MR; **bc** resolved: predicted correctly; **bc** in IQ2 stalls because MR is busy.	**bc** predicted taken.	**cmp** result is forwarded to MR; **bc** resolved: predicted incorrectly. Target instructions are not loaded into DS.	

Table I-31 shows the timing when the conditional branch is predicted as not-taken.

Table I-31. Conditional Loop Closure (Case 1)—Predict Not-Taken

Pipeline Stage		Cycle Number					
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**
Memory	FA	start:0				start:1	
	CARB	**add**1,**cmp**, **bc**,**add**2				**add**1,**cmp**, **bc**,**add**2	
	CACC		**add**1,**cmp**, **bc**,**add**2				**add**1,**cmp**, **bc**,**add**2
Dispatch	IQ3			**add**2			
	IQ2			**bc**			
	IQ1			**cmp**	**add**2		
	IQ0			**add**1	**cmp**, Tagpbr	**add**2	
IU	ID			**add**1	**cmp**, Tagpbr	**add**2	
	IE				**add**1	**cmp**, Tagpbr	
	IC					**add**1	**cmp**
	IWA					**add**1	**cmp**
	IWL						
	FPSB						
	ISB						
BPU	BE			**bc**			
	MR			**bc**	**bc**	**bc**	
	BW						
Notes				**bc** predicted not-taken.		**cmp** result is forwarded to MR; **bc** resolved: predicted incorrectly.	

Table I-32. Conditional Loop Closure (Case 1)—Predict Not-Taken (Continued)

Pipeline Stage		Cycle Number			
Unit	**Stage**	**7**	**8**	**9**	**10**
Memory	FA			Exit+8(*4)	
	CARB			Exit+8(*4)	
	CACC				Exit+8(*4)
Dispatch	IQ7			(op)	
	IQ6			(op)	(op)
	IQ5			(op)	(op)
	IQ4			(op)	(op)
	IQ3	**add**2		(op)	(op)
	IQ2	**bc**		(op)	(op)
	IQ1	**cmp**	**add**2	(op)	(op)
	IQ0	**add**1	**cmp**, Tagpbr	**add**2	(op)
IU	ID	**add**1	**cmp**, Tagpbr	**add**2	(op)
	IE		**add**1	**cmp**, Tagpbr	**add**2
	IC			**add**1	**cmp**
	IWA			**add**1	**cmp**
	IWL				
	FPSB				
	ISB				
BPU	BE	**bc**			
	MR	**bc**	**bc**	**bc**	
	BW				
Notes		State in this cycle is the same as the state in cycle 3 (4 cycles/loop)		**cmp** result is forwarded to MR; **bc** resolved: predicted correctly.	

I.1.1.7.2 Conditional Loop Closure—Case 2

The tables in this section show the timings for the following code sequence:

```
start: mull r1=r1*r3
       cmp cr3=r1,r4
       bc start, cr3
exit:  mull r1=r1*r4
```

Table I-33 shows the timing when the conditional branch is predicted as taken.

Table I-33. Conditional Loop Closure (Case 2)—Predict Taken

Pipeline Stage		Cycle Number					
Unit	**Stage**	**1**	**2**	**3**	**4**	**5-8**	**9**
Memory	FA	start:0		start:1			
	CARB	**mull**1,**cmp**, **bc**,**mull**2		**mull**1,**cmp**, **bc**,**mull**2			
	CACC		**mull**1,**cmp**, **bc**,**mull**2		**mull**1,**cmp**, **bc**,**mull**2		
Dispatch	IQ3			**mull**2		**mull**2	**mull**2
	IQ2			**bc**		**bc**	**bc**
	IQ1			**cmp**	**mull**2	**cmp**	**cmp**
	IQ0			**mull**1	**cmp**, Tagpbr	**mull**1	**mull**1
IU	ID			**mull**1	**cmp**, Tagpbr	**cmp**, Tagpbr	**mull**1
	IE				**mull**1	**mull**1	**cmp**, Tagpbr
	IC						**mull**1
	IWA						**mull**1
	IWL						
	FPSB						
	ISB						
BPU	BE			**bc**			
	MR			**bc**	**bc**	**bc**	**bc**
	BW						
Notes				**bc** predicted taken.			**cmp** result is forwarded to MR; **bc** resolved: predicted correctly; **bc** in IQ2 stalls because MR is busy.

Table I-33. Conditional Loop Closure (Case 2)—Predict Taken (Continued)

Pipeline Stage		Cycle Number					
Unit	**Stage**	**10**	**11**	**12-14**	**15**	**16**	**17**
Memory	FA	start:2				start:3	
	CARB	**mull**1,**cmp**, **bc**,**mull**2				**mull**1,**cmp**, **bc**,**mull**2	
	CACC		**mull**1,**cmp**, **bc**,**mull**2				**mull**1,**cmp**, **bc**,**mull**2
Dispatch	IQ3			**mull**2	**mull**2		
	IQ2	**mull**2		**bc**	**bc**	**mull**2	
	IQ1	**bc**		**cmp**	**cmp**	**bc**	
	IQ0	**cmp**	**mull**2	**mull**1	**mull**1	**cmp**	**mull**2
IU	ID	**cmp**	**cmp**, Tagpbr	**cmp**, Tagpbr	**mull**1	**cmp**	**cmp**, Tagpbr
	IE	**mull**1	**mull**1	**mull**1	**cmp**, Tagpbr	**mull**1	**mull**1
	IC	**cmp**			**mull**1	**cmp**	
	IWA	**cmp**			**mull**1	**cmp**	
	IWL						
	FPSB						
	ISB						
BPU	BE	**bc**				**bc**	
	MR	**bc**	**bc**		**bc**	**bc**	**bc**
	BW						
Notes		**bc** predicted taken.		State in this cycle (14) is the same as the state in cycle 8. (6 cycles/ loop)	**cmp** result is forwarded to MR; **bc** resolved: predicted correctly; **bc** in IQ2 stalls because MR is busy.	**bc** predicted taken.	

Table I-33. Conditional Loop Closure (Case 2)—Predict Taken (Continued)

Pipeline Stage		Cycle Number					
Unit	**Stage**	**18-20**	**21**	**22**	**23**	**24-28**	**29**
Memory	FA		Exit:				
	CARB		**mull**2				
	CACC			**mull**2			
Dispatch	IQ3	**mull**2	**mull**2				
	IQ2	**bc**	**bc**				
	IQ1	**cmp**	**cmp**				
	IQ0	**mull**1	**mull**1		**mull**2		
IU	ID	**cmp**, Tagpbr	**mull**1		**mull**2		
	IE	**mull**1	**cmp**, Tagpbr			**mull**2	
	IC		**mull**1	**cmp**			**mull**2
	IWA		**mull**1	**cmp**			**mull**2
	IWL						
	FPSB						
	ISB						
BPU	BE						
	MR		**bc**	**bc**			
	BW						
Notes		State in this cycle (14) is the same as the state in cycle 8. (6 cycles/loop)	**cmp** result is forwarded to MR; **bc** resolved: predicted incorrectly; instructions from start: three are purged and Exit: is fetched (FA).				

Table I-34 shows the timing when the conditional branch is predicted as not-taken.

Table I-34. Conditional Loop Closure (Case 2)—Predict Not-Taken

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5-8**	**9**	**10**
Memory	FA	start:0					start:1	
	CARB	**mull**1, **cmp**, **bc**,**mull**2					**mull**1, **cmp**, **bc**,**mull**2	
	CACC		**mull**1, **cmp**, **bc**,**mull**2					**mull**1, **cmp**, **bc**,**mull**2
Dispatch	IQ3			**mull**2				
	IQ2			**bc**				
	IQ1			**cmp**	**mull**2			
	IQ0			**mull**1	**cmp**, Tagpbr	**mull**2	**mull**2	
IU	ID			**mull**1	**cmp**, Tagpbr	**cmp**, Tagpbr	**mull**2	
	IE				**mull**1	**mull**1	**cmp**, Tagpbr	
	IC						**mull**1	**cmp**
	IWA						**mull**1	**cmp**
	IWL							
	FPSB							
	ISB							
BPU	BE			**bc**				
	MR			**bc**	**bc**	**bc**	**bc**	
	BW							
Notes				**bc** predicted not-taken.			**cmp** result is forwarded to MR; **bc** resolved: predicted incorrectly; fetcher recovers from mis-predicted branch.	

Table I-34. Conditional Loop Closure (Case 2)—Predict Not-Taken (Continued)

Pipeline Stage		Cycle Number						
Unit	**Stage**	**11**	**12**	**13-16**	**17**	**18**	**19**	**20**
Memory	FA				start:1			
	CARB				**mull**1, **cmp**, **bc**,**mull**2			
	CACC					**mull**1, **cmp**, **bc**,**mull**2		
Dispatch	IQ3	**mull**2					**mull**2	
	IQ2	**bc**					**bc**	
	IQ1	**cmp**	**mull**2				**cmp**	**mull**2
	IQ0	**mull**1	**cmp**, Tagpbr	**mull**2	**mull**2		**mull**1	**cmp**, Tagpbr
IU	ID	**mull**1	**cmp**, Tagpbr	**cmp**, Tagpbr	**mull**2		**mull**1	**cmp**, Tagpbr
	IE		**mull**1	**mull**1	**cmp**, Tagpbr			**mull**1
	IC				**mull**1	**cmp**		
	IWA				**mull**1	**cmp**		
	IWL							
	FPSB							
	ISB							
BPU	BE						**bc**	
	MR	**bc**	**bc**	**bc**	**bc**		**bc**	**bc**
	BW							
Notes		**bc** predicted not-taken	state in this cycle is the same as the state in cycle 4. (8 cycle/ loop)		**cmp** result is forwarded to MR; **bc** resolved: predicted incorrectly; fetcher recovers from mis-predicted branch.		**bc** predicted not-taken	

Table I-34. Conditional Loop Closure (Case 2)—Predict Not-Taken (Continued)

Pipeline Stage		Cycle Number				
Unit	**Stage**	**21**	**22**	**23**	**24-27**	**28**
Memory	FA					
	CARB					
	CACC	**mull**2				
Dispatch	IQ3					
	IQ2					
	IQ1					
	IQ0	**mull**2	**mull**2			
IU	ID	**cmp**, Tagpbr	**mull**2			
	IE	**mull**1	**cmp**, Tagpbr	**mull**2	**mull**2	
	IC		**mull**1	**cmp**		**mull**2
	IWA		**mull**1	**cmp**		**mull**2
	IWL					
	FPSB					
	ISB					
BPU	BE					
	MR	**bc**	**bc**			
	BW					
Notes			**cmp** result is forwarded to MR; **bc** resolved: predicted correctly.			

I.1.1.7.3 Conditional Loop Closure—Case 3

The tables in this section show the timings for the following code sequence:

```
start:  fmadd f1=(f2*f3)+f1
        fcmpu cr3=f1,f5
        bc start, cr3
exit:   fmadd f1=(f2*f4)+f1
```

Table I-35 shows the timing when the conditional branch is predicted as taken.

Table I-35. Conditional Loop Closure (Case 3)—Predict Taken

Pipeline Stage		Cycle Number				
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**
Memory	FA	start:0		start:1		
	CARB	**fmadd**1, **fcmpu**,**bc**, **fmadd**2		**fmadd**1, **fcmpu**,**bc**, **fmadd**2		
	CACC		**fmadd**1, **fcmpu**,**bc**, **fmadd**2		**fmadd**1, **fcmpu**,**bc**, **fmadd**2	
Dispatch	IQ3			**fmadd**2		**fmadd**2
	IQ2			**bc**		**bc**
	IQ1			**fcmpu**	**fmadd**2	**fcmpu**
	IQ0			**fmadd**1	**fcmpu**, Tagpbr	**fmadd**1
IU	ID					
	IE				Tag**fmadd**1	Tag**fcmpu**, Tagpbr
	IC					Tag**fmadd**1
	IWA					
	IWL					
	FPSB					
	ISB					
BPU	BE			**bc**		
	MR			**bc**	**bc**	**bc**
	BW					
FPU	F1					**fcmpu**
	FD				**fmadd**1	**fmadd**1
	FPM					**fmadd**1
	FPA					
	FWA					
	FWL					
Notes				**bc** predicted taken.		**bc** stalls because MR is busy.

Table I-35. Conditional Loop Closure (Case 3)—Predict Taken (Continued)

Pipeline Stage		Cycle Number				
Unit	**Stage**	**6**	**7**	**8**	**9**	**10**
Memory	FA					
	CARB					
	CACC					
Dispatch	IQ3					
	IQ2	**fmadd**2	**fmadd**2	**fmadd**2	**fmadd**2	**fmadd**2
	IQ1	**bc**	**bc**	**bc**	**bc**	**bc**
	IQ0	**fcmpu**	**fcmpu**	**fcmpu**	**fcmpu**	**fcmpu**
IU	ID	Tag**fmadd**1	Tag**fmadd**1	Tag**fmadd**1	Tag**fmadd**1	Tag**fmadd**1
	IE	Tagpbr	Tagpbr	Tagpbr	Tagpbr	Tagpbr
	IC	Tag**fcmpu**	Tag**fcmpu**	Tag**fcmpu**	Tag**fcmpu**	Tag**fcmpu**
	IWA					
	IWL					
	FPSB					
	ISB					
BPU	BE					
	MR	**bc**	**bc**	**bc**	**bc**	**bc**
	BW					
FPU	F1	**fmadd**1	**fmadd**1	**fmadd**1	**fmadd**1	
	FD	**fcmpu**	**fcmpu**	**fcmpu**	**fcmpu**	**fmadd**1
	FPM	**fmadd**1				**fcmpu**
	FPA	**fmadd**1	**fmadd**1			
	FWA		**fmadd**1	**fmadd**1		
	FWL					
Notes		Floating-point compare (IC) instructions synchronize between FW and IC. **fcmpu** in IQ0 stalls because F1 is busy.	**fcmpu** stalls in FD due to a data dependency.			

Table I-35. Conditional Loop Closure (Case 3)—Predict Taken (Continued)

Pipeline Stage		Cycle Number					
Unit	**Stage**	**11**	**12**	**13**	**14**	**15**	**16**
Memory	FA				start:2		
	CARB				**fmadd**1, **fcmpu**,**bc**, **fmadd**2		
	CACC					**fmadd**1, **fcmpu**,**bc**, **fmadd**2	
Dispatch	IQ3						**fmadd**2
	IQ2	**fmadd**2	**fmadd**2	Tag**fmadd**2	Tag**fmadd**2		**bc**
	IQ1	**bc**	**bc**	**bc**	**bc**		**fcmpu**
	IQ0	Tag**fcmpu**	Tag**fcmpu**	Tag**fcmpu**	Tag**fcmpu**	Tag**fmadd**2	**fmadd**1
IU	ID	Tag**fmadd**1	Tag**fmadd**1	Tag**fmadd**1	Tag**fcmpu**	Tag**fmadd**2	
	IE	Tagpbr	Tagpbr	Tagpbr	Tag**fmadd**1	Tag**fcmpu**, Tagpbr	Tagpbr
	IC	Tag**fcmpu**	Tag**fcmpu**			Tag**fmadd**1	Tag**fcmpu**
	IWA						
	IWL						
	FPSB						
	ISB						
BPU	BE				**bc**		
	MR	**bc**	**bc**	**bc**	**bc**	**bc**	**bc**
	BW						
FPU	F1	**fcmpu**		**fmadd**2	**fmadd**2	**fmadd**2	
	FD	**fmadd1**	**fcmpu**	**fcmpu**	**fcmpu**	**fcmpu**	**fmadd**2
	FPM	**fmadd**1	**fmadd**1				**fcmpu**
	FPA	**fcmpu**	**fmadd**1	**fmadd**1			
	FWA		**fcmpu**		**fmadd**1		
	FWL						
Notes				CR information is sent to the BPU; branch is resolved as predicted correctly.	**bc** predicted taken	Tag**fmadd**2 is purged from the pipeline.	**fmadd**2 is purged from the FPU.

Table I-35. Conditional Loop Closure (Case 3)—Predict Taken (Continued)

Pipeline Stage		Cycle Number					
Unit	**Stage**	**17**	**18**	**19**	**20**	**21**	**22**
Memory	FA				Start:3		
	CARB				**fmadd**1, **fcmpu**,**bc**, **fmadd**2		
	CACC					**fmadd**1, **fcmpu**,**bc**, **fmadd**2	
Dispatch	IQ3						**fmadd**2
	IQ2	**fmadd**2	**fmadd**2	**fmadd**2	Tag**fmadd**2		**bc**
	IQ1	**bc**	**bc**	**bc**	**bc**		**fcmpu**
	IQ0	**fcmpu**	Tag**fcmpu**	Tag**fcmpu**	Tag**fcmpu**	Tag**fmadd**2	**fmadd**1
IU	ID	Tag**fmadd**1	Tag**fmadd**1	Tag**fmadd**1	Tag**fcmpu**	Tag**fmadd**2	
	IE	Tagpbr	Tagpbr	Tagpbr	Tag**fmadd**1	Tag**fcmpu**, Tagpbr	Tagpbr
	IC	Tag**fcmpu**	Tag**fcmpu**			Tag**fmadd**1	Tag**fcmpu**
	IWA						
	IWL						
	FPSB						
	ISB						
BPU	BE				**bc**		
	MR	**bc**	**bc**	**bc**	**bc**	**bc**	**bc**
	BW						
FPU	F1		**fcmpu**		**fmadd**2	**fmadd**2	**fmadd**2
	FD	**fmadd**1	**fmadd**1	**fcmpu**	**fcmpu**	**fcmpu**	**fcmpu**
	FPM		**fmadd**1	**fmadd**1			
	FPA	**fcmpu**		**fmadd**1	**fmadd**1		
	FWA		**fcmpu**			**fmadd**1	
	FWL						
Notes				CR data is sent to the BPU; branch is resolved as predicted correctly.	**bc** predicted taken		**fmadd**2 is purged from the FPU.

Table I-35. Conditional Loop Closure (Case 3)—Predict Taken (Continued)

Pipeline Stage		Cycle Number					
Unit	**Stage**	**23**	**24**	**25**	**26**	**27**	**28**
Memory	FA				Exit:		
	CARB				**fmadd**2		
	CACC					**fmadd**2	
Dispatch	IQ3	**fmadd**2					
	IQ2	**bc**	**fmadd**2	**fmadd**2	**fmadd**2		
	IQ1	**fcmpu**	**bc**	**bc**	**bc**		
	IQ0	**fmadd**1	**fcmpu**	Tag**fcmpu**	Tag**fcmpu**		**fmadd**2
IU	ID		Tag**fmadd**1	Tag**fmadd**1	Tag**fmadd**1		
	IE	Tagpbr	Tagpbr	Tagpbr	Tagpbr		
	IC	Tag**fcmpu**	Tag**fcmpu**	Tag**fcmpu**			
	IWA						
	IWL						
	FPSB						
	ISB						
BPU	BE						
	MR	**bc**	**bc**	**bc**	**bc**		
	BW						
FPU	F1			**fcmpu**		**fmadd**2	
	FD		**fmadd**1	**fmadd**1	**fcmpu**	**fcmpu**	
	FPM	**fcmpu**		**fmadd**1	**fmadd**1		
	FPA		**fcmpu**		**fmadd**1	**fmadd**1	
	FWA			**fcmpu**			
	FWL						
Notes			The state in this cycle is the same as the state in cycle 17 (7 cyc/loop).		CR information is sent to the BPU; branch is resolved as predicted **incorrectly**.	**fmadd**1, **fcmpu**, and **fmadd**2 are purged from the FPU.	

Table I-35. Conditional Loop Closure (Case 3)—Predict Taken (Continued)

Pipeline Stage		Cycle Number			
Unit	**Stage**	**29**	**30**	**31**	**32**
Memory	FA				
	CARB				
	CACC				
Dispatch	IQ3				
	IQ2				
	IQ1				
	IQ0				
IU	ID				
	IE	Tag**fmadd**2			
	IC		Tag**fmadd**2		
	IWA				
	IWL				
	FPSB				
	ISB				
BPU	BE				
	MR				
	BW				
FPU	F1				
	FD	**fmadd**2			
	FPM		**fmadd**2		
	FPA			**fmadd**2	
	FWA				**fmadd**2
	FWL				
Notes					

Table I-36 shows the timing when the conditional branch is predicted as not-taken.

Table I-36. Conditional Loop Closure (Case 3)—Predict Not-Taken

Pipeline Stage		Cycle Number					
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**
Memory	FA	start:0					
	CARB	**fmadd**1, **fcmpu**,**bc**, **fmadd**2					
	CACC		**fmadd**1, **fcmpu**,**bc**, **fmadd**2				
Dispatch	IQ3			**fmadd**2			
	IQ2			**bc**			
	IQ1			**fcmpu**	**fmadd**2		
	IQ0			**fmadd**1	**fcmpu**, Tagpbr	**fmadd**2	**fmadd**2
IU	ID						
	IE				Tag**fmadd**1	Tag**fcmpu**, Tagpbr	Tagpbr
	IC					Tag**fmadd**1	Tag**fcmpu**
	IWA						
	IWL						
	FPSB						
	ISB						
BPU	BE			**bc**			
	MR			**bc**	**bc**	**bc**	**bc**
	BW						
FPU	F1					**fcmpu**	
	FD				**fmadd**1	**fmadd**1	**fcmpu**
	FPM					**fmadd**1	**fmadd**1
	FPA						**fmadd**1
	FWA						
	FWL						
Notes				**bc** predicted not-taken.		**fmadd**2 stalls in IQ0 because F1 is busy.	Floating-point compare (IC) instructions synchronize between FW and IC.

Table I-36. Conditional Loop Closure (Case 3)—Predict Not-Taken (Continued)

Pipeline Stage		Cycle Number					
Unit	**Stage**	**7**	**8**	**9**	**10**	**11**	**12**
Memory	FA						
	CARB						
	CACC						
Dispatch	IQ3						
	IQ2						
	IQ1						
	IQ0						
IU	ID	Tag**fmadd**2	Tag**fmadd**2	Tag**fmadd**2	Tag**fmadd**2	Tag**fmadd**2	Tag**fmadd**2
	IE	Tagpbr	Tagpbr	Tagpbr	Tagpbr	Tagpbr	Tagpbr
	IC	Tag**fcmpu**	Tag**fcmpu**	Tag**fcmpu**	Tag**fcmpu**	Tag**fcmpu**	Tag**fcmpu**
	IWA						
	IWL						
	FPSB						
	ISB						
BPU	BE						
	MR	**bc**	**bc**	**bc**	**bc**	**bc**	**bc**
	BW						
FPU	F1	**fmadd**2	**fmadd**2	**fmadd**2			
	FD	**fcmpu**	**fcmpu**	**fcmpu**	**fmadd**2	**fmadd**2	
	FPM				**fcmpu**	**fmadd**2	**fmadd**2
	FPA	**fmadd**1				**fcmpu**	**fmadd**2
	FWA		**fmadd**1				**fcmpu**
	FWL						
Notes		**fcmpu** stalls in FD due to a data dependency.					

Table I-36. Conditional Loop Closure (Case 3)—Predict Not-Taken (Continued)

Pipeline Stage		Cycle Number					
Unit	**Stage**	**13**	**14**	**15**	**16**	**17**	**18**
Memory	FA	start:1					
	CARB	**fmadd**1, **fcmpu**,**bc**, **fmadd**2					
	CACC		**fmadd**1, **fcmpu**,**bc**, **fmadd**2				
Dispatch	IQ3			**fmadd**2			
	IQ2			**bc**			
	IQ1			**fcmpu**	**fmadd**2		
	IQ0			**fmadd**1	**fcmpu**, Tagpbr	**fmadd**2	**fmadd**2
IU	ID	Tag**fmadd**2					
	IE	Tagpbr			Tag**fmadd**1	Tag**fcmpu**, Tagpbr	Tagpbr
	IC					Tag**fmadd**1	Tag**fcmpu**
	IWA						
	IWL						
	FPSB						
	ISB						
BPU	BE			**bc**			
	MR	**bc**		**bc**	**bc**	**bc**	**bc**
	BW						
FPU	F1					**fcmpu**	
	FD				**fmadd**1	**fmadd**1	**fcmpu**
	FPM					**fmadd**1	**fmadd**1
	FPA	**fmadd**2					**fmadd**1
	FWA		**fmadd**2				
	FWL						
Notes		CR information is sent to the BPU; branch is resolved as predicted incorrectly.	**fmadd**2 is purged from the FPU.	**bc** predicted not-taken.	State in this cycle is the same as in cycle 4 (12 cycle /loop).	**fmadd**2 stalls in IQ0 because F1 is busy.	Floating-point compare (IC) instructions synchronize between FW and IC.

Table I-36. Conditional Loop Closure (Case 3)—Predict Not-Taken (Continued)

Pipeline Stage		Cycle Number					
Unit	**Stage**	**19**	**20**	**21**	**22**	**23**	**24**
Memory	FA						
	CARB						
	CACC						
Dispatch	IQ3						
	IQ2						
	IQ1						
	IQ0						
IU	ID	Tag**fmadd**2	Tag**fmadd**2	Tag**fmadd**2	Tag**fmadd**2	Tag**fmadd**2	Tag**fmadd**2
	IE	Tagpbr	Tagpbr	Tagpbr	Tagpbr	Tagpbr	Tagpbr
	IC	Tag**fcmpu**	Tag**fcmpu**	Tag**fcmpu**	Tag**fcmpu**	Tag**fcmpu**	Tag**fcmpu**
	IWA						
	IWL						
	FPSB						
	ISB						
BPU	BE						
	MR	**bc**	**bc**	**bc**	**bc**	**bc**	**bc**
	BW						
FPU	F1	**fmadd**2	**fmadd**2	**fmadd**2			
	FD	**fcmpu**	**fcmpu**	**fcmpu**	**fmadd**2	**fmadd**2	
	FPM				**fcmpu**	**fmadd**2	**fmadd**2
	FPA	**fmadd**1				**fcmpu**	**fmadd**2
	FWA		**fmadd**1				**fcmpu**
	FWL						
Notes		**fcmpu** stalls in FD due to a data dependency.					

Table I-36. Conditional Loop Closure (Case 3)—Predict Not-Taken (Continued)

Pipeline Stage		Cycle Number		
Unit	**Stage**	**25**	**26**	**27**
Memory	FA			
	CARB			
	CACC			
Dispatch	IQ3			
	IQ2			
	IQ1			
	IQ0			
IU	ID	Tag**fmadd**2		
	IE	Tagpbr	Tag**fmadd**2	
	IC			Tag**fmadd**2
	IWA			
	IWL			
	FPSB			
	ISB			
BPU	BE			
	MR	**bc**		
	BW			
FPU	F1			
	FD			
	FPM			
	FPA	**fmadd**2		
	FWA		**fmadd**2	**fmadd**2
	FWL			
Notes		CR information is sent to the BPU; branch is resolved as predicted **correctly**.	**fmadd**2 stalls in FW waiting for its tag to clear IC.	

I.1.1.7.4 Conditional Loop Closure—Case 1

The tables in this section show the timings for the following code sequence:

```
start: fmadds f1=(f2*f3)+f1
       fcmpu cr3=f1,f5
       bc start, cr3
exit:  fmadds f1=(f2*f4)+f1
```

Table I-37 shows the timing when the conditional branch is predicted as taken.

Table I-37. Conditional Loop Closure (Case 4)—Predict Taken

Pipeline Stage		Cycle Number				
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**
Memory	FA	start:0		start:1		
	CARB	**fmadds**1 **fcmpu**,**bc**, **fmadds**2		**fmadds**1, **fcmpu**,**bc**, **fmadds**2		
	CACC		**fmadds**1, **fcmpu**,**bc**, **fmadds**2		**fmadds**1, **fcmpu**,**bc**, **fmadds**2	
Dispatch	IQ3			**fmadds**2		**fmadds**2
	IQ2			**bc**		**bc**
	IQ1			**fcmpu**	**fmadds**2	**fcmpu**
	IQ0			**fmadds**1	**fcmpu**, Tagpbr	**fmadds**1
IU	ID					
	IE				Tag**fmadds**1	Tag**fcmpu**, Tagpbr
	IC					Tag**fmadds**1
	IWA					
	IWL					
	FPSB					
	ISB					
BPU	BE			**bc**		
	MR			**bc**	**bc**	**bc**
	BW					
FPU	F1					
	FD				**fmadds**1	**fcmpu**
	FPM					**fmadds**1
	FPA					
	FWA					
	FWL					
Notes				**bc** predicted taken.		**bc** stalls because MR is busy; **fcmpu** stalls in FD due to a data dependency.

Table I-37. Conditional Loop Closure (Case 4)—Predict Taken (Continued)

Pipeline Stage		Cycle Number				
Unit	**Stage**	**6**	**7**	**8**	**9**	**10**
Memory	FA					
	CARB					
	CACC					
Dispatch	IQ3					
	IQ2	**fmadds**2	**fmadds**2	**fmadds**2	**fmadds**2	**fmadds**2
	IQ1	**bc**	**bc**	**bc**	**bc**	**bc**
	IQ0	**fcmpu**	**fcmpu**	**fcmpu**	**fcmpu**	Tag**fcmpu**
IU	ID	Tag**fmadds**1	Tag**fmadds**1	Tag**fmadds**1	Tag**fmadds**1	Tag**fmadds**1
	IE	Tagpbr	Tagpbr	Tagpbr	Tagpbr	Tagpbr
	IC	Tag**fcmpu**	Tag**fcmpu**	Tag**fcmpu**	Tag**fcmpu**	Tag**fcmpu**
	IWA					
	IWL					
	FPSB					
	ISB					
BPU	BE					
	MR	**bc**	**bc**	**bc**	**bc**	**bc**
	BW					
FPU	F1	**fmadds**1	**fmadds**1	**fmadds**1		
	FD	**fcmpu**	**fcmpu**	**fcmpu**	**fmadds**1	**fcmpu**
	FPM				**fcmpu**	**fmadds**1
	FPA	**fmadds**1				**fcmpu**
	FWA		**fmadds**1			
	FWL					
Notes		Floating-point compare (IC) instructions synchronize between FW and IC. **fcmpu** in IQ0 stalls because F1 is busy.				

Table I-37. Conditional Loop Closure (Case 4)—Predict Taken (Continued)

Pipeline Stage		Cycle Number				
Unit	**Stage**	**11**	**12**	**13**	**14**	**15**
Memory	FA			start:2		
	CARB			**fmadds**1, **fcmpu**,**bc**, **fmadds**2		
	CACC				**fmadds**1, **fcmpu**,**bc**, **fmadds**2	
Dispatch	IQ3					**fmadds**2
	IQ2	**fmadds**2	Tag**fmadds**2	Tag**fmadds**2		**bc**
	IQ1	**bc**	**bc**	**bc**		**fcmpu**
	IQ0	Tag**fcmpu**	Tag**fcmpu**	Tag**fcmpu**	Tag**fmadds**2	**fmadds**1
IU	ID	Tag**fmadds**1	Tag**fmadds**1	Tag**fcmpu**	Tag**fmadds**2	
	IE	Tagpbr	Tagpbr	Tag**fmadds**1	Tag**fcmpu**, Tagpbr	Tagpbr
	IC	Tag**fcmpu**			Tag**fmadds**1	Tag**fcmpu**
	IWA					
	IWL					
	FPSB					
	ISB					
BPU	BE			**bc**		
	MR	**bc**	**bc**	**bc**	**bc**	**bc**
	BW					
FPU	F1		**fmadds**2	**fmadds**2		
	FD	**fcmpu**	**fcmpu**	**fcmpu**	**fmadds**2	
	FPM				**fcmpu**	**fmadds**2
	FPA	**fmadds**1				**fcmpu**
	FWA	**fcmpu**	**fmadds**1			
	FWL					
Notes			CR information is sent to the BPU; branch is resolved as predicted correctly.	**bc** predicted taken	Tag**fmadd**2 is purged from the pipeline.	**fmadds**2 is purged from the FPU.

Table I-37. Conditional Loop Closure (Case 4)—Predict Taken (Continued)

Pipeline Stage		Cycle Number				
Unit	**Stage**	**16**	**17**	**18**	**19**	**20**
Memory	FA				Start:3	
	CARB				**fmadds**1, **fcmpu**,**bc**, **fmadds**2	
	CACC					**fmadds**1, **fcmpu**,**bc**, **fmadds**2
Dispatch	IQ3					
	IQ2	**fmadds**2	**fmadds**2	**fmadds**2	Tag**fmadds**2	
	IQ1	**bc**	**bc**	**bc**	**bc**	
	IQ0	**fcmpu**	Tag**fcmpu**	Tag**fcmpu**	Tag**fcmpu**	Tag**fmadds**2
IU	ID	Tag**fmadds**1	Tag**fmadds**1	Tag**fmadds**1	Tag**fcmpu**	Tag**fmadds**2
	IE	Tagpbr	Tagpbr	Tagpbr	Tag**fmadds**1	Tag**fcmpu**, Tagpbr
	IC	Tag**fcmpu**	Tag**fcmpu**			Tag**fmadds**1
	IWA					
	IWL					
	FPSB					
	ISB					
BPU	BE				**bc**	
	MR	**bc**	**bc**	**bc**	**bc**	**bc**
	BW					
FPU	F1				**fmadds**2	**fmadds**2
	FD	**fmadds**1	**fcmpu**	**fcmpu**	**fcmpu**	**fcmpu**
	FPM		**fmadds**1			
	FPA			**fmadds**1		
	FWA	**fcmpu**			**fmadds**1	
	FWL					
Notes				CR information is sent to the BPU; branch is resolved as predicted correctly	**bc** predicted taken	

Table I-37. Conditional Loop Closure (Case 4)—Predict Taken (Continued)

Pipeline Stage		Cycle Number				
Unit	**Stage**	**21**	**22**	**23**	**24**	**25**
Memory	FA					Start:3
	CARB					**fmadds**1, **fcmpu**,**bc**, **fmadds**2
	CACC					
Dispatch	IQ2	**fmadds**2				
	IQ1	**bc**	**fmadds**2	**fmadds**2	**fmadds**2	Tag**fmadds**2
	IQ0	**fcmpu**	**bc**	**bc**	**bc**	**bc**
IU	ID	**fmadds**1	**fcmpu**	Tag**fcmpu**	Tag**fcmpu**	Tag**fcmpu**
	IE		Tag**fmadds**1	Tag**fmadds**1	Tag**fmadds**1	Tag**fcmpu**
	IC	Tagpbr	Tagpbr	Tagpbr	Tagpbr	Tag**fmadds**1
	IWA	Tag**fcmpu**	Tag**fcmpu**	Tag**fcmpu**		
	IWL					
	FPSB					
	ISB					
BPU	BE					
	MR					**bc**
	BW	**bc**	**bc**	**bc**	**bc**	**bc**
FPU	F1					
	FD					**fmadds**2
	FPM	**fmadds**2	**fmadds**1	**fcmpu**	**fcmpu**	**fcmpu**
	FPA	**fcmpu**		**fmadds**1		
	FWA		**fcmpu**		**fmadds**1	
	FWL			**fcmpu**		**fmadds**1
Notes						

Table I-37. Conditional Loop Closure (Case 4)—Predict Taken (Continued)

Pipeline Stage		Cycle Number				
Unit	**Stage**	**26**	**27**	**28**	**29**	**30**
Memory	FA					Exit:
	CARB					**fmadds**2
	CACC	**fmadds**1, **fcmpu**,**bc**, **fmadds**2				
Dispatch	IQ2		**fmadds**2		**fmadds**2	**fmadds**2
	IQ1		**bc**	**fmadds**2	**bc**	**bc**
	IQ0		**fcmpu**	**bc**	Tag**fcmpu**	Tag**fcmpu**
IU	ID	Tag**fmadds**2	**fmadds**1	**fcmpu**	Tag**fmadds**1	Tag**fmadds**1
	IE	Tag**fmadds**2		Tag**fmadds**1	Tagpbr	Tagpbr
	IC	Tag**fcmpu**, Tagpbr	Tagpbr	Tagpbr	Tag**fcmpu**	
	IWA	Tag**fmadds**1	Tag**fcmpu**	Tag**fcmpu**		
	IWL					
	FPSB					
	ISB					
BPU	BE					
	MR				**bc**	**bc**
	BW	**bc**	**bc**	**bc**		
FPU	F1					
	FD	**fmadds**2			**fcmpu**	**fcmpu**
	FPM	**fcmpu**	**fmadds**2	**fmadds**1	**fmadds**1	
	FPA		**fcmpu**			**fmadds**1
	FWA			**fcmpu**	**fcmpu**	
	FWL					
Notes						CR information is sent to the BPU; branch is resolved as predicted incorrectly.

Table I-37. Conditional Loop Closure (Case 4)—Predict Taken (Continued)

Pipeline Stage		Cycle Number				
Unit	**Stage**	**31**	**32**	**33**	**34**	**35**
Memory	FA					
	CARB					
	CACC	**fmadds**2				
Dispatch	IQ2					
	IQ1					
	IQ0		**fmadds**2			
IU	ID					
	IE			Tag**fmadds**2		
	IC				Tag**fmadds**2	
	IWA					
	IWL					
	FPSB					
	ISB					
BPU	BE					
	MR					
	BW					
FPU	F1	**fmadds**2				
	FD	**fcmpu**		**fmadds**2		
	FPM				**fmadds**2	
	FPA					**fmadds**2
	FWA	**fmadds**1				
	FWL					
Notes		**fmadds**1, **fcmpu**, and **fmadds**2 are purged from the FPU.				

Table I-38 shows the timing when the conditional branch is predicted as not-taken.

Table I-38. Conditional Loop Closure (Case 4)—Predict Not-Taken

Pipeline Stage		Cycle Number					
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**
Memory	FA	start:0					
	CARB	**fmadds**1, **fcmpu**,**bc**, **fmadds**2					
	CACC		**fmadds**1, **fcmpu**,**bc**, **fmadds**2				
Dispatch	IQ3			**fmadds**2			
	IQ2			**bc**			
	IQ1			**fcmpu**	**fmadds**2		
	IQ0			**fmadds**1	**fcmpu**, Tagpbr	**fmadds**2	
IU	ID						Tag**fmadds**2
	IE				Tag**fmadds**1	Tag**fcmpu**, Tagpbr	Tagpbr
	IC					Tag**fmadds**1	Tag**fcmpu**
	IWA						
	IWL						
	FPSB						
	ISB						
BPU	BE			**bc**			
	MR			**bc**	**bc**	**bc**	**bc**
	BW						
FPU	F1						**fmadds**2
	FD				**fmadds**1	**fcmpu**	**fcmpu**
	FPM					**fmadds**1	
	FPA						**fmadds**1
	FWA						
	FWL						
Notes				**bc** predicted not-taken.		**fcmpu** stalls in FD due to a data dependency.	Floating-point compare (IC) instructions synchronize between FW and IC.

Table I-38. Conditional Loop Closure (Case 4)—Predict Not-Taken (Continued)

Pipeline Stage		Cycle Number				
Unit	**Stage**	**7**	**8**	**9**	**10**	**11**
Memory	FA					
	CARB					
	CACC					
Dispatch	IQ3					
	IQ2					
	IQ1					
	IQ0					
IU	ID	Tag**fmadds**2	Tag**fmadds**2	Tag**fmadds**2	Tag**fmadds**2	Tag**fmadds**2
	IE	Tagpbr	Tagpbr	Tagpbr	Tagpbr	Tagpbr
	IC	Tag**fcmpu**	Tag**fcmpu**	Tag**fcmpu**	Tag**fcmpu**	Tag**fcmpu**
	IWA					
	IWL					
	FPSB					
	ISB					
BPU	BE					
	MR	**bc**	**bc**	**bc**	**bc**	**bc**
	BW					
FPU	F1	**fmadds**2	**fmadds**2			
	FD	**fcmpu**	**fcmpu**	**fmadds**2		
	FPM			**fcmpu**	**fmadds**2	
	FPA				**fcmpu**	**fmadds**2
	FWA	**fmadds**1				**fcmpu**
	FWL					
Notes						

Table I-38. Conditional Loop Closure (Case 4)—Predict Not-Taken (Continued)

Pipeline Stage		Cycle Number				
Unit	**Stage**	**12**	**13**	**14**	**15**	**16**
Memory	FA	start:1				
	CARB	**fmadds**1, **fcmpu**,**bc**, **fmadds**2				
	CACC		**fmadds**1, **fcmpu**,**bc**, **fmadds**2			
Dispatch	IQ3			**fmadds**2		
	IQ2			**bc**		
	IQ1			**fcmpu**	**fmadds**2	
	IQ0			**fmadds**1	**fcmpu**, Tagpbr	**fmadds**2
IU	ID	Tag**fmadds**2				
	IE	Tagpbr			Tag**fmadds**1	Tag**fcmpu**, Tagpbr
	IC					Tag**fmadds**1
	IWA					
	IWL					
	FPSB					
	ISB					
BPU	BE			**bc**		
	MR	**bc**		**bc**	**bc**	**bc**
	BW					
FPU	F1					
	FD				**fmadds**1	**fcmpu**
	FPM					**fmadds**1
	FPA					
	FWA	**fmadds**2	**fmadds**2			
	FWL					
Notes		CR information is sent to the BPU; branch is resolved as predicted incorrectly.		**bc** predicted not-taken.		**fcmpu** stalls in FD due to a data dependency.

Table I-38. Conditional Loop Closure (Case 4)—Predict Not-Taken (Continued)

Pipeline Stage		Cycle Number				
Unit	**Stage**	**17**	**18**	**19**	**20**	**21**
Memory	FA					
	CARB					
	CACC					
Dispatch	IQ3					
	IQ2					
	IQ1					
	IQ0					
IU	ID	Tag**fmadds**2	Tag**fmadds**2	Tag**fmadds**2	Tag**fmadds**2	Tag**fmadds**2
	IE	Tagpbr	Tagpbr	Tagpbr	Tagpbr	Tagpbr
	IC	Tag**fcmpu**	Tag**fcmpu**	Tag**fcmpu**	Tag**fcmpu**	Tag**fcmpu**
	IWA					
	IWL					
	FPSB					
	ISB					
BPU	BE					
	MR	**bc**	**bc**	**bc**	**bc**	**bc**
	BW					
FPU	F1	**fmadds**2	**fmadds**2	**fmadds**2		
	FD	**fcmpu**	**fcmpu**	**fcmpu**	**fmadds**2	
	FPM				**fcmpu**	**fmadds**2
	FPA	**fmadds**1				**fcmpu**
	FWA		**fmadds**1			
	FWL					
Notes		Floating-point compare (IC) instructions synchronize between FW and IC. **fcmpu** in IQ0 stalls because F1 is busy.		The state in this cycle is the same as that of cycle 8 (11 cycle/ loop).		

Table I-39 describes the conditional loop closure when timing is predicted as not-taken.

Table I-39. Conditional Loop Closure (Case 4)—Predict Not-Taken (Continued)

Pipeline Stage		Cycle Number			
Unit	**Stage**	**22**	**23**	**24**	**25**
Memory	FA				
	CARB				
	CACC				
Dispatch	.IQ3				
	IQ2				
	IQ1				
	IQ0				
IU	ID	Tag**fmadds**2	Tag**fmadds**2		
	IE	Tagpbr	Tagpbr	Tag**fmadds**2	
	IC	Tag**fcmpu**			Tag**fmadds**2
	IWA				
	IWL				
	FPSB				
	ISB				
BPU	BE				
	MR	**bc**	**bc**		
	BW				
FPU	F1				
	FD				
	FPM				
	FPA	**fmadds**2			
	FWA		**fmadds**2		
	FWL			**fmadds**2	**fmadds**2
Notes			CR information is sent to the BPU; branch is resolved as predicted correctly.		

I.2 Integer Instruction Timing Examples

There are two groups of integer instructions, ALU instructions and load/store operations. The ALU instruction timing examples are given in Section I.2.1, “ALU Instruction Timings,” and the load/store timing examples are given in Section I.2.2, “Load/Store Instruction Timings.”

In many cases, the 601 has extra hardware to reduce or eliminate the number of stall cycles between instructions with register dependencies. Most stalls can be avoided by reordering the instructions to place instructions without dependencies where a stall cycle would occur. The details of these scenarios are discussed in the following sections. Timing diagrams illustrating the stall are shown along with examples showing how stall cycles can be eliminated. Note that some stalls, such as those caused by a context-synchronizing instruction, cannot be avoided by rescheduling code.

Some stalls are caused by hardware resource conflicts, such as when the cache has more than one outstanding miss.

I.2.1 ALU Instruction Timings

ALU instructions can stall due to GPR and SPR dependencies and when there is contention for hardware resources, such as the cache.

I.2.1.1 GPR Data Dependencies

The most basic data dependency occurs when an instruction has an operand that is the result of the previous instruction. Instructions flowing through the IU must complete in order, and ALU instructions always write back in order with respect to one another. That eliminates write-after-read (WAR) and write-after-write (WAW) dependencies.

Special logic in the IU handles read-after-write (RAW) dependencies to prevent stalls due to RAW dependencies. Table I-40 shows this scenario, using the following code sequence:

```
Start: add r1=r2,r3
       subf r4=r5,r1
```

Table I-40. Adjacent IU ALU Instructions with a GPR Dependency (RAW)

Pipeline Stage		Cycle Number					
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**
Memory	FA	**add**, **subf**					
	CARB	**add**, **subf**					
	CACC		**add**, **subf**				
Dispatch	IQ1			**subf**			
	IQ0			**add**			
IU	ID			**add**	**subf**		
	IE				**add**	**subf**	
	IC					**add**	**subf**
	IWA					**add**	**subf**
	IWL						
	FPSB						
	ISB						

No stall cycles occur in this code sequence because a forward path brings the **add** results directly from IE to ID in the same cycle, since the data does not reach the GPR for another cycle (IWA). This example can be extended to any pair of arithmetic, shift/rotate, or logical operations executed by the IU.

The following code sequence shows a similar scenario:

```
Start: add r1=r2,r3
       oril r10=r11,r12
       subf r4=r5,r1
```

The instruction in the IWA stage writes to a GPR being read by the instruction in the ID stage without encountering stalls, because the register data is not read during the ID stage until the instruction in the IWA stage writes its results. The timing for this sequence is shown in Table I-41.

Table I-41. GPR Dependency (RAW) between First and Third ALU Operation

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Memory	FA	**add**, **oril**, **subf**						
	CARB	**add**, **oril**, **subf**						
	CACC		**add**, **oril**, **subf**					
Dispatch	IQ2			**subf**				
	IQ1			**oril**				
	IQ0			**add**	**subf**			
IU	ID			**add**	**oril**	**subf**		
	IE				**add**	**oril**	**subf**	
	IC					**add**	**oril**	**subf**
	IWA					**add**	**oril**	**subf**
	IWL							
	FPSB							
	ISB							

A GPR dependency can also occur when an **mfspr** instruction is followed by an instruction that is trying to use that GPR. Some SPRs can be read in a single cycle, while others take several cycles, in which case, the IU pipeline stalls. Typically these stalls cannot be avoided by rescheduling code. Timings for this scenario differ according to the register being accessed, and are shown as follows.

The following code sequence is used for the four examples:

```
Start: mfspr r3=SPR
       add r5=r4,r3
```

Table I-42 shows timings for **mfcr** and **mfspr** for the following SPRs: XER, MQ, PIR, DABR, EAR, RTCL, DEC, LR, CTR, and the BAT registers. There is GPR dependency, but there is no pipeline stall.

Table I-42. mfspr Instruction Followed by a Dependent ALU Instruction (No Stall)

Pipeline Stage		Cycle Number					
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**
Memory	FA	**mfspr**, **add**					
	CARB	**mfspr**, **add**					
	CACC		**mfspr**,**add**				
Dispatch	IQ1			**add**			
	IQ0			**mfspr**	**add**		
IU	ID			**mfspr**	**add**		
	IE				**mfspr**	**add**	
	IC					**mfspr**	**add**
	IWA					**mfspr**	**add**
	IWL						
	FPSB						
	ISB						

Table I-43 shows the timings for **mfsr** and **mfsrin** instructions and for the **mfspr** instruction for SDR1. The stall cycle here is caused by a GPR dependency and can be avoided by inserting a nondependent instruction between the **mfspr** and the **add** instruction.

Table I-43. mfspr Followed by a Dependent ALU Instruction (Avoidable Stall)

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Memory	FA	**mfspr**, **add**						
	CARB	**mfspr**, **add**						
	CACC		**mfspr**,**add**					
Dispatch	IQ1			**add**				
	IQ0			**mfspr**	**add**			
IU	ID			**mfspr**	**add**			
	IE				**mfspr**	**add**	**add**	
	IC					**mfspr**		**add**
	IWA					**mfspr**		**add**
	IWL							
	FPSB							
	ISB							

Table I-44 shows how advantage can be taken of the stall by the placement of a nondependent instruction (**oril**) between the **mfspr** and **add** instructions. Again, these timings are the same for registers accessed in Table I-43—**mfsr**, **mfsrin**, and **mfspr** for SDR1.

Table I-44. mfspr Followed by a Dependent ALU Instruction Showing Code Rescheduling

Pipeline Stage		Cycle Number				
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**
Memory	FA					
	CARB					
	CACC					
Dispatch	IQ2	**add**				
	IQ1	**oril**	**add**			
	IQ0	**mfspr**	**oril**	**add**		
IU	ID	**mfspr**	**oril**	**add**		
	IE		**mfspr**	**oril**	**add**	
	IC			**mfspr**	**oril**	**add**
	IWA			**mfspr**	**oril**	**add**
	IWL					
	FPSB					
	ISB					

Table I-45 shows the timing for the **mfmsr** and **mfspr** instructions for the following SPRs—SPRGn, DSISR, DAR, HID0–HID2, RTCU, SRR0, SRR1, and PVR. For **mfmsr**, 5, *n* is 0; for PVR, *n* is 5; and for the other SPRs, *n* is 2.

Table I-45. mfspr Followed by a Dependent ALU Instruction (Multicycle Stall)

Pipeline Stage		Cycle Number					
Unit	**Stage**	**1**	**2**	**3**	**4**	**5–(5+n)**	**6 + n**
Memory	FA	**mfspr**, **add**					
	CARB	**mfspr**, **add**					
	CACC		**mfspr**,**add**				
Dispatch	IQ1	**add**					
	IQ0	**mfspr**	**add**	**add**			
IU	ID	**mfspr**	**add**	**add**			
	IE		**mfspr**	**mfspr**	**add**	**add**	
	IC				**mfspr**		**add**
	IWA				**mfspr**	**mfspr**	**add**
	IWL						
	FPSB						
	ISB						

I.2.1.2 Condition Register (CR) Dependencies

In general, instructions can read and write the CR without causing IU pipeline stalls. Examples of common sequences that do not cause pipeline stalls are shown in the following examples.

Table I-46 shows an ALU instruction with the record option followed by a CR logical instruction, using the following code sequence:

```
Start: add. r5=r6 + x'3'
       crnand cr1=cr2, cr0.
```

Note that despite the dependency on the CR, there is no stall.

Table I-46. ALU Instruction with CR Update Followed by a CR Logical Instruction

Pipeline Stage		Cycle Number			
Unit	**Stage**	**1**	**2**	**3**	**4**
Memory	FA				
	CARB				
	CACC				
Dispatch	IQ1	**crnand**			
	IQ0	**add.**	**crnand**		
IU	ID	**add.**	**crnand**		
	IE		**add.**	**crnand**	
	IC			**add.**	**crnand**
	IWA			**add.**	**crnand**
	IWL				
	FPSB				
	ISB				

Table I-47 shows the timing for a CR logical instruction followed by another CR logical instruction, using the following code sequence:

```
Start: crand cr28=cr3, cr4
       cror cr1=cr28, cr2
```

Note that despite the dependency on the CR, there is no stall.

Table I-47. Consecutive CR Logical Instructions

Pipeline Stage		Cycle Number			
Unit	**Stage**	**1**	**2**	**3**	**4**
Memory	FA				
	CARB				
	CACC				
Dispatch	IQ1	**cror**			
	IQ0	**crand**	**cror**		
IU	ID	**crand**	**cror**		
	IE		**crand**	**cror**	
	IC			**crand**	**cror**
	IWA			**crand**	**cror**
	IWL				
	FPSB				
	ISB				

Table I-48 shows the timing for a **mtcrf** instruction followed by a CR logical instruction, using the following code sequence:

```
Start: mtcrf CR=x'ff', r5
       crand cr20=cr30, cr10
```

Note that despite the dependency on the CR, there is no stall.

Table I-48. mtcrf Instruction Followed by a CR Logical Instruction

Pipeline Stage		Cycle Number			
Unit	**Stage**	**1**	**2**	**3**	**4**
Memory	FA				
	CARB				
	CACC				
Dispatch	IQ1	**crand**			
	IQ0	**mtcrf**	**crand**		
IU	ID	**mtcrf**	**crand**		
	IE		**mtcrf**	**crand**	
	IC			**mtcrf**	**crand**
	IWA			**mtcrf**	**crand**
	IWL				
	FPSB				
	ISB				

Table I-49 shows the timing for a **mcrf** instruction followed by a CR logical instruction, using the following code sequence:

```
Start: mcrf CR[8:11]=CR[0:3]
       crxor cr15=cr9, cr10
```

Note that despite the dependency on the CR, there is no stall.

Table I-49. mcrf Instruction Followed by a CR Logical Instruction

Pipeline Stage		Cycle Number			
Unit	**Stage**	**1**	**2**	**3**	**4**
Memory	FA				
	CARB				
	CACC				
Dispatch	IQ1	**crxor**			
	IQ0	**mcrf**	**crxor**		
IU	ID	**mcrf**	**crxor**		
	IE		**mcrf**	**crxor**	
	IC			**mcrf**	**crxor**
	IWA			**mcrf**	**crxor**
	IWL				
	FPSB				
	ISB				

Table I-50 shows the timing for two consecutive ALU instructions that update the CR, using the following code sequence:

```
Start: add. r1=r2 + r3
       subf. r4=r5 - r6
```

Note that despite the dependency on the CR, there is no stall.

Table I-50. Consecutive ALU Instructions that Update the CR

Pipeline Stage		Cycle Number			
Unit	**Stage**	**1**	**2**	**3**	**4**
Memory	FA				
	CARB				
	CACC				
Dispatch	IQ1	**subf.**			
	IQ0	**add.**	**subf.**		
IU	ID	**add.**	**subf.**		
	IE		**add.**	**subf.**	
	IC			**add.**	**subf.**
	IWA			**add.**	**subf.**
	IWL				
	FPSB				
	ISB				

Table I-51 shows the timing for a Move XER to CR (**mcrxr**) instruction followed by a CR logical instruction, using the following code sequence:

```
Start: mcrxr CR[8:11]=XER[0:3]
       crand cr15=cr9,cr10
```

Note that despite the dependency on the CR, there is no stall.

Table I-51. mcrxr Instruction Followed by a CR Logical Instruction

Pipeline Stage		Cycle Number			
Unit	**Stage**	**1**	**2**	**3**	**4**
Memory	FA				
	CARB				
	CACC				
Dispatch	IQ1	**crand**			
	IQ0	**mcrxr**	**crand**		
IU	ID	**mcrxr**	**crand**		
	IE		**mcrxr**	**crand**	
	IC			**mcrxr**	**crand**
	IWA			**mcrxr**	**crand**
	IWL				
	FPSB				
	ISB				

However, some instruction sequences generate stalls in the pipeline due to a read-after-write dependency on the CR.

mcrf is the only instruction that reads the CR from the ID stage. An **mcrf** instruction that immediately follows any instruction that updates the CR stalls in the ID stage for one cycle. A nondependent instruction can be scheduled in between the instruction that writes the CR and the **mcrf** to allow the IU processor to do useful work during the cycle that the **mcrf** would be held in ID. The following examples show both the stall and how the stall can be eliminated with proper code scheduling.

Table I-52 shows the **mcrf** instruction stalling for one cycle when it follows and ALU instruction that updates the CR. The code sequence for this example is as follows:

```
Start: add. r5=r6 + x'1'
       mcrf CR[8:11], CR[12:15]
```

Table I-52. ALU Instruction Updating the CR Followed by an mcrf Instruction

Pipeline Stage		Cycle Number				
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**
Memory	FA					
	CARB					
	CACC					
Dispatch	IQ1	**mcrf**				
	IQ0	**add.**	**mcrf**	**mcrf**		
IU	ID	**add.**	**mcrf**	**mcrf**		
	IE		**add.**		**mcrf**	
	IC			**add.**		**mcrf**
	IWA			**add.**		**mcrf**
	IWL					
	FPSB					
	ISB					

Table I-53 shows how the throughput is improved by inserting a nondependent instruction (**ori**) between the **add.** instruction and the **mcrf** instruction, and eliminating the stall in the pipeline. The code sequence for this example is as follows:

```
Start: add. r5=r6 + x'1'
       oril r7=r5, x'00ff'
       mcrf CR[8:11], CR[12:15]
```

Table I-53. Scheduling Code to Eliminate a Stall between an ALU Instruction Updating the CR and an mcrf Instruction

Pipeline Stage		Cycle Number				
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**
Memory	FA					
	CARB					
	CACC					
Dispatch	IQ2	**mcrf**				
	IQ1	**oril**	**mcrf**			
	IQ0	**add.**	**oril**	**mcrf**		
IU	ID	**add.**	**oril**	**mcrf**		
	IE		**add.**	**oril**	**mcrf**	
	IC			**add.**	**oril**	**mcrf**
	IWA			**add.**	**oril**	**mcrf**
	IWL					
	FPSB					
	ISB					

The **stwcx.** instruction updates the CR from the IWA stage—which causes pipeline stalls if the next instruction needs to read the CR. The **stwcx.** instruction timing depends upon the state of the reservation bit when the **stwcx.** instruction enters IWA. If the reservation is cleared, the **stwcx**. instruction spends one cycle in IWA and IE is held one cycle. If the reservation is set, the **stwcx**. instruction spends two cycles in IWA causing IE to be held for two cycles. These pipeline stalls cannot be avoided. If the **stwcx**. instruction is followed by an **mcrf** instruction, an additional stall cycle is generated which, however, can be filled with a nondependent instruction.

The following examples show timings for the **stwcx**. instruction.

Table I-54 and Table I-55 show the timing for when an **stwcx.** instruction is followed by an ALU instruction and the reservation is set and cleared respectively. These two examples use the following code sequence:

```
Start: stwcx. r5, r6, r7
       addic r8=r8 + x'1'
```

Table I-54 shows the timing for an **stwcx.** instruction followed by an ALU instruction when the reservation bit has been cleared. Note that in this example, the stall of the **add** instruction in the IE stage is a function of the way the **stwcx**. instruction is implemented. Any instruction that follows an **stwcx.** stalls for one cycle in IE even though the **stwcx.** has left the pipeline.

Table I-54. stwcx. Followed by an ALU Instruction (Reservation Cleared)

Pipeline Stage		Cycle Number				
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**
Memory	FA					
	CARB		**stwcx.**			
	CACC			**stwcx.**		
Dispatch	IQ1	**addic**				
	IQ0	**stwcx.**	**addic**			
IU	ID	**stwcx.**	**addic**			
	IE		**stwcx.**	**addic**	**addic**	
	IC			**stwcx.**		**addic**
	IWA			**stwcx.**		**addic**
	IWL					
	FPSB					
	ISB					

Table I-55 shows the timing for an **stwcx.** instruction followed by an ALU instruction when the reservation bit has been set.

Table I-55. stwcx. Instruction Followed by an ALU Instruction (Reservation Set)

Pipeline Stage		Cycle Number					
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**
Memory	FA						
	CARB						
	CACC						
Dispatch	IQ1	**addic**					
	IQ0	**stwcx.**	**addic**				
IU	ID	**stwcx.**	**addic**				
	IE		**stwcx.**	**addic**	**addic**	**addic**	
	IC				**stwcx.**		**addic**
	IWA			**stwcx.**	**stwcx.**		**addic**
	IWL						
	FPSB						
	ISB						

Table I-56 and Table I-57 show the timings when an **mcrf** instruction follows an **stwcx.** instruction when the reservation bit has been cleared and set respectively.

```
Start: stwcx. r5, r6, r7
       mcrf CR[8:11], CR[12:15]
```

Table I-56. stwcx. Followed by an mcrf Instruction (Reservation Cleared)

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Memory	FA							
	CARB							
	CACC							
Dispatch	IQ1	**mcrf**						
	IQ0	**stwcx.**	**mcrf**					
IU	ID	**stwcx.**	**mcrf**	**mcrf**	**mcrf**	**mcrf**		
	IE		**stwcx.**				**mcrf**	
	IC				**stwcx.**			**mcrf**
	IWA			**stwcx.**	**stwcx.**			**mcrf**
	IWL							
	FPSB							
	ISB							

The stall of the add instruction in the IE stage is a function of how the **stwcx.** instruction is implemented. Any instruction that follows an **stwcx.** stalls for two cycles in the IE stage when the reservation is set for the **stwcx.** instruction. The **mcrf** is held in ID because it has a CR dependency on the **stwcx.**

Table I-57. stwcx. Instruction Followed by an mcrf Instruction (Reservation Set)

Pipeline Stage		Cycle Number					
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**
Memory	FA						
	CARB						
	CACC						
Dispatch	IQ1	**mcrf**					
	IQ0	**stwcx.**	**mcrf**				
IU	ID	**stwcx.**	**mcrf**	**mcrf**	**mcrf**		
	IE		**stwcx.**			**mcrf**	
	IC			**stwcx.**			**mcrf**
	IWA			**stwcx.**			**mcrf**
	IWL						
	FPSB						
	ISB						

I.2.1.3 XER Dependencies

Four instructions implemented in the 601 (**lscbx**, **lswx**, **stswx**, and **mcrxr**) read the XER from the ID stage, and, as a result, can stall in the ID stage for a cycle when they immediately follow an instruction that updates the XER. Unlike the CR, the XER dependency is checked on the entire register and not on a field-by-field basis.

The XER is updated by arithmetic instructions in which the OE bit is set, by arithmetic and shift operations that set the CA bit, by the **lscbx** and **mcrxr** instructions, and by an **mtspr** instructions that specify the XER. Code scheduling can effectively eliminate the stall cycle by moving a nondependent instruction into the stall position. The sequences shown below do not generate a stall.

Table I-58 shows an ALU instruction that updates the XER followed by an ALU instruction that reads the XER, using the following code sequence:

```
Start: addic r1=r2 + x'1'
       addme r4=r5 + XER[CA] - 1
```

Note that despite the dependency on the CA bit of the XER, there is no stall.

Table I-58. addic Instruction Followed by an addme Instruction

Pipeline Stage		Cycle Number			
Unit	**Stage**	**1**	**2**	**3**	**4**
Memory	FA				
	CARB				
	CACC				
Dispatch	IQ1	**addme**			
	IQ0	**addic**	**addme**		
IU	ID	**addic**	**addme**		
	IE		**addic**	**addme**	
	IC			**addic**	**addme**
	IWA			**addic**	**addme**
	IWL				
	FPSB				
	ISB				

Table I-59 shows the timing when a shift instruction that updates the XER is followed by an ALU instruction that reads the XER, using the following code sequence:

```
Start: sraw r1=r2, r3
       addze r4=r5 + XER[CA]
```

Note that despite the dependency on the CA bit of the XER, there is no stall.

Table I-59. sraw Instruction Followed by an addze Instruction

Pipeline Stage		Cycle Number			
Unit	**Stage**	**1**	**2**	**3**	**4**
Memory	FA				
	CARB				
	CACC				
Dispatch	IQ1	**addze**			
	IQ0	**sraw**	**addze**		
IU	ID	**sraw**	**addze**		
	IE		**sraw**	**addze**	
	IC			**sraw**	**addze**
	IWA			**sraw**	**addze**
	IWL				
	FPSB				
	ISB				

Table I-60 shows the timing for a shift instruction that updates the XER followed by a move from XER instruction, using the following code sequence:

```
Start: sraw r1 = r2, r3
       mfspr r4=XER
```

Note that despite the dependency on the CA bit of the XER, there is no stall.

Table I-60. sraw Instruction Followed by an mfxer Instruction

Pipeline Stage		Cycle Number			
Unit	**Stage**	**1**	**2**	**3**	**4**
Memory	FA				
	CARB				
	CACC				
Dispatch	IQ1	**mfspr**			
	IQ0	**sraw**	**mfspr**		
IU	ID	**sraw**	**mfspr**		
	IE		**sraw**	**mfspr**	
	IC			**sraw**	**mfspr**
	IWA			**sraw**	**mfspr**
	IWL				
	FPSB				
	ISB				

The first, third, and fifth code sequences below generate a stall cycle. The second, fourth, and sixth code sequences below demonstrate how an instruction can be moved into the stall slot to prevent a dead cycle.

Table I-61 shows the stall that occurs when timing for an ALU instruction that updates the XER followed by a load string instruction. The following code sequence is shown:

```
Start: addic r4=r4 + x'1'
       lswx r28, r5, r6
```

Note that because of the dependency on the CA bit of the XER, the **lswx** instruction stalls for one cycle in the ID stage.

Table I-61. addic Instruction Followed by a lswx Instruction

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5–5 + (n-2)**	**6 + (n-2)**	**7 + (n-2)**
Memory	FA							
	CARB				**lswx**	**lswx**		
	CACC					**lswx**	**lswx**	
Dispatch	IQ1	**lswx**						
	IQ0	**addic**	**lswx**					
IU	ID	**addic**	**lswx**	**lswx**				
	IE		**addic**		**lswx**	**lswx**		
	IC			**addic**		**lswx**	**lswx**	
	IWA			**addic**				
	IWL						**lswx**	**lswx**
	FPSB							
	ISB							
Notes						n is the number of registers to be loaded.		

Table I-62 shows the timing how stall is avoided in the previous example by inserting a nondependent instruction (**oril**) between the ALU instruction that updates the XER and the load string instruction. The following code example is used:

```
Start: addic r4=r4+x'1'
       oril r10=r11, r12
       lswx r28, r5, r6
```

Note that the **oril** instruction executes during the cycle where the **lswx** would stall while the XER(CA) is being updated.

Table I-62. Avoiding the Stall between an addic Instruction and an lswx Instruction

Pipeline Stage		Cycle Number						
Unit	Stage	1	2	3	4	5	6	7
Memory	FA							
	CARB				**lswx**	**lswx**		
	CACC					**lswx**	**lswx**	
Dispatch	IQ2	**lswx**						
	IQ1	**oril**	**lswx**					
	IQ0	**addic**	**oril**	**lswx**				
IU	ID	**addic**	**oril**	**lswx**				
	IE		**addic**	**oril**	**lswx**[a]	**lswx**		
	IC			**addic**	**oril**	**lswx**		
	IWA			**addic**	**oril**			
	IWL						**lswx**	**lswx**
	FPSB							
	ISB							

a. The XER byte count value is assumed to be 8 (two registers of data). The operand is assumed to be aligned.

Table I-63 shows the **mcrxr** instruction stalling in the ID stage for one cycle because of a dependency on the OV and SO bits of the XER, which are set by the previous **mulo** instruction. The following code sequence is used:

```
Start: mulo r1=r2 * r3
       mcrxr CR[8:11]=XER[0:3]
```

Table I-63. mulo Instruction Followed by an mcrxr Instruction

Pipeline Stage		Cycle Number				
Unit	**Stage**	**1**	**2–6**	**7**	**8**	**9**
Memory	FA					
	CARB					
	CACC					
Dispatch	IQ1	**mcrxr**				
	IQ0	**mulo**	**mcrxr**			
IU	ID	**mulo**	**mcrxr**	**mcrxr**		
	IE		**mulo**		**mcrxr**	
	IC			**mulo**		**mcrxr**
	IWA			**mulo**		**mcrxr**
	IWL					
	FPSB					
	ISB					
Notes			It is assumed that the **r**B operand requires only five cycles in IE.			

The timing example in Table I-64 shows how the stall shown in the previous example is avoided by scheduling a nondependent instruction (**oril**) between the **mulo** and **mcrxr** instructions. The following code sequence is shown:

```
Start: mulo r1=r2 * r3
       oril r10=r11, r12
       mcrxr CR[8:11]=XER[0:3]
```

Table I-64. Eliminating the Stall between an mulo and an mcrxr Instruction

Pipeline Stage		Cycle Number				
Unit	**Stage**	**1**	**2–6**	**7**	**8**	**9**
Memory	FA					
	CARB					
	CACC					
Dispatch	IQ2	**mcrxr**				
	IQ1	**oril**	**mcrxr**			
	IQ0	**mulo**	**oril**	**mcrxr**		
IU	ID	**mulo**	**oril**	**mcrxr**		
	IE		**mulo**	**oril**	**mcrxr**	
	IC			**mulo**	**oril**	**mcrxr**
	IWA			**mulo**	**oril**	**mcrxr**
	IWL					
	FPSB					
	ISB					
Notes			It is assumed that the RB operand requires only 5 cycles in IE.			

Table I-65 shows the stall that occurs when a Move to XER instruction is followed by a load string instruction. Dependency on the byte count field of the XER causes the **lswx** instruction to stall for one cycle in the ID stage. The following code sequence is shown:

```
Start: mtspr XER=r5
       lswx r4, r5, r6
```

Table I-65. mtxer Instruction Followed by an lswx Instruction

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5–5 + (n - 2)**	**6 + (n-2)**	**7 + (n-2)**
Memory	FA							
	CARB				**lswx**	**lswx**		
	CACC					**lswx**	**lswx**	
1.552"	IQ1	**lswx**						
	IQ0	**mfspr**	**lswx**					
IU	ID	**mfspr**	**lswx**	**lswx**				
	IE		**mfspr**		**lswx**	**lswx**		
	IC			**mfspr**		**lswx**	**lswx**	
	IWA			**mfspr**				
	IWL						**lswx**	**lswx**
	FPSB							
	ISB							
Notes						n is the number of registers to be loaded		

Table I-66 shows how the stall in the previous example is eliminated by scheduling a nondependent instruction (**oril**) between the **mtspr** and **lswx** instructions.

The following code sequence is shown:

```
Start: mtspr XER=r5
       oril r10=r11, r12
       lswx r4, r5, r6
```

Table I-66. Eliminating the Stall between an mtxer and an lswx Instruction

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Memory	FA							
	CARB				**lswx**	**lswx**		
	CACC					**lswx**	**lswx**	
Dispatch	IQ2	**lswx**						
	IQ1	**oril**	**lswx**					
	IQ0	**mfspr**	**oril**	**lswx**				
IU	ID	**mfspr**	**oril**	**lswx**				
	IE		**mfspr**	**oril**	**lswx** [1]	**lswx**		
	IC			**mfspr**	**oril**	**lswx**	**lswx**	
	IWA			**mfspr**	**oril**			
	IWL						**lswx**	**lswx**
	FPSB							
	ISB							
Notes					[1] The XER byte count value is assumed to be 8 (2 registers worth of data). The operand is assumed to be aligned.			

I.2.1.4 MQ Dependencies

The MQ register is written and read by certain multiply, divide, and shift instructions. There are no stall cycles associated with reading or writing this register for any code sequence. Two examples are given below.

Table I-67 shows the timing for a Move to MQ instruction followed by an instruction that reads the MQ. The following code sequence is shown:

```
Start: mtspr MQ=r5
       mul r6=r7 * r8
```

Table I-67. Move to MQ Followed by an mul Instruction

Pipeline Stage		Cycle Number				
Unit	**Stage**	**1**	**2**	**3**	**4 - 7**	**8**
Memory	FA					
	CARB					
	CACC					
Dispatch	IQ1	**mul**				
	IQ0	**mtspr**	**mul**			
IU	ID	**mtspr**	**mul**			
	IE		**mtspr**	**mul**	**mul**	
	IC			**mtspr**		**mul**
	IWA			**mtspr**		**mul**
	IWL					
	FPSB					
	ISB					
Notes					It is assumed that the RB operand requires only five cycles in the IE stage.	

Table I-68 shows the timing for an instruction that writes the MQ followed by an instruction that reads the MQ. Note that despite the dependency on MQ, there is no stall.

The following code sequence is shown:

```
Start: slq r5, r6, 4
       slliq r7, r8, 4
```

Table I-68. Consecutive Shift Left with MQ Instructions

Pipeline Stage		Cycle Number			
Unit	**Stage**	**1**	**2**	**3**	**4**
Memory	FA				
	CARB				
	CACC				
Dispatch	IQ1	**slliq**			
	IQ0	**slq**	**slliq**		
IU	ID	**slq**	**slliq**		
	IE		**slq**	**slliq**	
	IC			**slq**	**slliq**
	IWA			**slq**	**slliq**
	IWL				
	FPSB				
	ISB				

I.2.1.5 Other Register Dependencies on ALU Instructions

Timings to access the other SPRs than those discussed above depends on the SPR being referenced. However, there is only one scenario involving other SPRs where code can be scheduled to improve performance. For some SPRs that are implemented in the sequencer, the **mtspr** instruction runs on the sequencer, while the IU is running subsequent instructions. The **mtspr** accesses to SRR0, SRR1, DSISR, DAR, SPRGn, and RTCU execute in parallel with subsequent IU instructions, and therefore cause a two-cycle stall in the subsequent instruction if the subsequent instruction requires the sequencer.

However, if the subsequent instruction wants to use the sequencer, (**mfspr**(SRR0, SRR1, DSISR, DAR, SPRGn, RTCU, HID0, HID1, HID2, PVR), **mfmsr**, **clcs**, **sc**, **isync**, **tlbie**, and **rfi**) it must wait until the sequencer finishes processing the **mtspr** instruction. The following operations on the 601 use the sequencer and are subject to the two-cycle stall described above—**mtspr**(SRR0, SRR1, DSISR, DAR, SPRGn, RTCU, HID0, HID1, HID2), **mtmsr**.

There is potentially an additional two-cycle stall for this sequencer dependency. Code scheduling can be used to move nondependent operations into the two stall cycles. Instruction sequences illustrating this behavior are shown below.

Table I-69 shows a **mtspr** instruction that requires the use of the sequencer followed by an instruction that has no dependency and therefore, no stall.

The following code sequence is shown:

```
Start: mtspr SRR0=r5
```

```
add r4=r4 + r3
```

Table I-69. Move to SPR Followed by ALU Instruction

Pipeline Stage		Cycle Number				
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**
Memory	FA					
	CARB					
	CACC					
Dispatch	IQ1	**add**				
	IQ0	**mfspr**	**add**	**add**		
IU	ID	**mfspr**	**add**	**add**		
	IE		**mfspr**	**mfspr**	**add**	
	IC				**mfspr**	**add**
	IWA				**mfspr**	**add**
	IWL					
	FPSB					
	ISB					

Table I-70 shows the timing for an **mtspr** instruction that requires the sequencer. In this example, the second instruction stalls in IE for three cycles because there is a hardware resource dependency.

The following code sequence is shown:

```
Start: mtspr SRR0=r5
       rfi
```

Table I-70. mtspr Instruction Followed by an rfi Instruction

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5 - 6**	**7 - 8**	**9 - 21**
Memory	FA							
	CARB							
	CACC							
Dispatch	IQ1	**rfi**						
	IQ0	**mfspr**	**rfi**	**rfi**				
IU	ID	**mfspr**	**rfi**	**rfi**				
	IE		**mfspr**	**mfspr**	**rfi**	**rfi**	**rfi**	
	IC				**mfspr**		**rfi**	**rfi**
	IWA				**mfspr**		**rfi**	**rfi**
	IWL							
	FPSB							
	ISB							
Notes								13 cycles in IWA

The **mtmsr** instruction and **mtspr** access to HID0, HID1, HID2 instructions are all context synchronizing; executing one of these instructions causes all following instructions in the pipeline to be purged and refetched.In addition to the length of time it takes the **mtmsr** instruction to execute, three cycles are required to refetch the next instruction and have it progress to IE. Table I-71 shows the timing for the following code sequence:

```
Start: mtmsr msr=r5
       add r1=r2 + r3
```

Table I-71. mtmsr Instruction Followed by Any Instruction

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4 - 16**	**17 - 18**	**19**	**20**
Memory	FA							
	CARB							
	CACC							
Dispatch	IQ1	**add**						
	IQ0	**mtmsr**	**add**				**add**	
IU	ID	**mtmsr**	**add**	**add**			**add**	
	IE		**mtmsr**	**mtmsr**	**add**			**add**
	IC				**mtmsr**			
	IWA				**mtmsr**			
	IWL							
	FPSB							
	ISB							
Notes					Shown for FPSCR (FEX) = 0. Additional four-cycle delay for FEX = 1.	**add** and any instructions in the pipeline are purged & refetched.		

The segment registers can be written by consecutive instructions; the adjacent **mtsr** (or **mtsrin**) instructions flow through the pipeline without any stalls in between them.

Table I-72 shows the timing for two consecutive **mtsr** instructions, using the following code sequence:

```
Start: mtsr SEG2=r3
       mtsr SEG3=r4
```

Table I-72. Consecutive mtsr Instructions

Pipeline Stage		Cycle Number			
Unit	**Stage**	**1**	**2**	**3**	**4**
Memory	FA				
	CARB				
	CACC				
Dispatch	IQ1	**mtsr** 2			
	IQ0	**mtsr** 1	**mtsr** 2		
IU	ID	**mtsr** 1	**mtsr** 2		
	IE		**mtsr** 1	**mtsr** 2	
	IC			**mtsr** 1	**mtsr** 2
	IWA			**mtsr** 1	**mtsr** 2
	IWL				
	FPSB				
	ISB				

I.2.2 Load/Store Instruction Timings

Loads and stores have different timing characteristics than other IU instructions—primarily because there is an additional stage in the pipeline for accessing the cache. The following sections describe the interaction of loads and stores with other instructions as they flow through the pipeline.

I.2.2.1 Load Target Dependencies

Load instructions write a GPR with data from memory. Instructions that follow the load and refer to the load target GPR stall in the IE stage until the GPR data is available. If the load hits in the cache and the dependent instruction immediately follows the load, the IU pipeline stalls for one cycle. This cycle can be eliminated by scheduling a nondependent instruction between the load and the dependent operation. If the instruction following the load is not dependent upon the load target, the instruction does not stall in the execute stage.

If the load misses in the cache, several IU instructions may execute before the load target data is available. In this case, each subsequent instruction is checked for a register dependency with a previous load target and held in the IE stage if a dependency exists.

I.2.2.1.1 Load Instructions Followed by an Independent Operation Body

The following code sequence shows a load instruction followed by a nondependent instruction. The instruction timing for this example in Table I-73 shows how no stall is encountered because there is no register dependency:

```
Start:  lwz r5=MEM[x'4'(r6)]
        add r7=r7 + r8
```

Table I-73. Load Followed by an ALU Instruction with No Register Dependency

Pipeline Stage		Cycle Number			
Unit	**Stage**	**1**	**2**	**3**	**4**
Memory	FA				
	CARB		**lwz**		
	CACC			**lwz**	
Dispatch	IQ1	**add**			
	IQ0	**lwz**	**add**		
IU	ID	**lwz**	**add**		
	IE		**lwz**	**add**	
	IC			**lwz**	**add**
	IWA				**add**
	IWL				**lwz**
	FPSB				
	ISB				

I.2.2.1.2 Load Instruction Followed by a Dependent Operation

Table I-74 shows the one-cycle stall that occurs when a load instruction is followed by a dependent instruction.

```
Start: lwz r5=MEM[x'4'(r6)]
       add r5=r5 + r8
```

In this example the **add** instruction is dependent on the load target. The add stalls 1 cycle in the IE stage.

Table I-74. Load Followed by an ALU Instruction with a Register Dependency

Pipeline Stage		Cycle Number				
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**
Memory	FA					
	CARB		**lwz**			
	CACC			**lwz**		
Dispatch	IQ1	**add**				
	IQ0	**lwz**	**add**			

Table I-74. Load Followed by an ALU Instruction with a Register Dependency (Continued)

Pipeline Stage		Cycle Number				
IU	ID	**lwz**	**add**			
	IE		**lwz**	**add**	**add**	
	IC			**lwz**		**add**
	IWA					**add**
	IWL				**lwz**	
	FPSB					
	ISB					

I.2.2.1.3 Load Instruction Followed by a Dependent Operation with an Intervening Independent Instruction

Table I-75 shows how scheduling a nondependent instruction between the load and a dependent instruction eliminates the one-cycle stall illustrated in the previous example.

```
Start: lwz r5=MEM[x'4'(r6)]
       subf MEM[x’10’(r11)]=r10
       add r5=r5 + x'1'
```

In this example, the **addic** instruction is dependent on the load target; however, the data is available for the **addic** at the start of the IE stage.

Table I-75. Avoiding a Stall in the Pipeline on a Dependent ALU from a Load

Pipeline Stage		Cycle Number				
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**
Memory	FA					
	CARB		**lwz**			
	CACC			**lwz**		
Dispatch	IQ2	**add**				
	IQ1	**subf**	**add**			
	IQ0	**lwz**	**subf**	**add**		

Table I-75. Avoiding a Stall in the Pipeline on a Dependent ALU from a Load

Pipeline Stage		Cycle Number				
IU	ID	**lwz**	**subf**	**add**		
	IE		**lwz**	**subf**	**add**	
	IC			**lwz**	**subf**	**add**
	IWA				**subf**	**add**
	IWL				**lwz**	
	FPSB					
	ISB					

I.2.2.1.4 Consecutive Load Instructions without Register Dependencies

Table I-76 shows that no stall occurs when a load instruction is followed by another load instruction that does not have data dependencies. The following code sequence is used:

```
Start: lwz r3=MEM[r4 + r5]
       lwz r8=MEM[r9, r10]
```

In this example the second **lwz** instruction (**lwz** 2) needs the data sent to the target register by **lwz** 1, but it is not available in cycle three, when **lwz** 2 is ready for it. Therefore, the second **lwz** instruction stalls for one cycle in the IE stage until the target data is available.

Table I-76. Consecutive Load Instructions with No Register Dependencies

Pipeline Stage		Cycle Number				
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**
Memory	FA					
	CARB		**lwz** 1	**lwz** 2		
	CACC			**lwz** 1	**lwz** 2	
Dispatch	IQ1	**lwz** 2				
	IQ0	**lwz** 1	**lwz** 2			
IU	ID	**lwz** 1	**lwz** 2			
	IE		**lwz** 1	**lwz** 2		
	IC			**lwz** 1	**lwz** 2	
	IWA					
	IWL				**lwz** 1	**lwz** 2
	FPSB					
	ISB					

I.2.2.1.5 Consecutive Load Instructions with a Dependency

Table I-77 shows stall occurring when a load instruction has a dependency on the

immediately previous load instruction. The following code sequence is used:

```
Start:  lwz r5=MEM[x'10'(r6)]
        lwz r6=MEM[x0'(r5)]
```

Note that in this example, the second **lwz** instruction stalls in the IE stage for an extra cycle (cycle 4).

Table I-77. Load Instruction Followed by a Dependent Load Instruction

Pipeline Stage		Cycle Number					
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**
Memory	FA						
	CARB		**lwz** 1	**lwz** 2	**lwz** 2		
	CACC			**lwz** 1		**lwz** 2	
Dispatch	IQ1	**lwz** 2					
	IQ0	**lwz** 1	**lwz** 2				
IU	ID	**lwz** 1	**lwz** 2				
	IE		**lwz** 1	**lwz** 2	**lwz** 2		
	IC			**lwz** 1		**lwz** 2	
	IWA						
	IWL				**lwz** 1		**lwz** 2
	FPSB						
	ISB						

The following code sequence shows how the stall encountered in the previous example can be eliminated by scheduling a nondependent instruction (**ori**) before the second, dependent load instruction with an intervening independent instruction.

```
Start: lwz r5=MEM[x'10'(r6)]
       oril r0=r1, r1
       lwz r5=MEM[x'0'(r5)]
```

Table I-78 shows the instruction timing for this code sequence. Note that because the **ori** instruction

Table I-78. Avoiding a Pipeline Stall from Two Loads with an rD Dependency

Pipeline Stage		Cycle Number					
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**
Memory	FA						
	CARB		**lwz** 1		**lwz** 2		
	CACC			**lwz** 1		**lwz** 2	
Dispatch	IQ2	**lwz** 2					
	IQ1	**oril**	**lwz** 2				
	IQ0	**lwz** 1	**oril**	**lwz** 2			
IU	ID	**lwz** 1	**oril**	**lwz** 2			
	IE		**lwz** 1	**oril**	**lwz** 2		
	IC			**lwz** 1	**oril**	**lwz** 2	
	IWA				**oril**		
	IWL				**lwz** 1		**lwz** 2
	FPSB						
	ISB						

Load multiple and load string instructions write one GPR at a time. Therefore a subsequent dependent operation does not have to wait for a cycle for the load data to become available unless it is dependent upon the last register to be written by the load. The sample code sequences is shown below.

The following code sequence illustrates a load multiple instruction followed by an instruction with a dependency on something other than GPR31:

```
Start: lmw r29, x'0'(r3)
       add r30=r30 + x'0001'
```

The timing for this example is shown in Table I-79. Note that in this example, the **add** is stalled in the ID stage while the **lmw** instruction is in the IE stage. Because there is only one execute unit, the **add** instruction must wait until the **lmw** instruction completes. In this example, the **add** instruction also is given a register dependency on the **lmw** instruction; however, because it is not the last register loaded by the **lmw** instruction, the data is available for the **add** in time to prevent a pipeline stall.

Table I-79. Load Multiple Instruction Followed by an ALU Instruction with a Dependency on a Register Besides GPR31

Pipeline Stage		Cycle Number					
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**
Memory	FA						
	CARB		**lmw**	**lmw**	**lmw**		
	CACC			**lmw**	**lmw**	**lmw**	
Dispatch	IQ1	**add**					
	IQ0	**lmw**	**add**				
IU	ID	**lmw**	**add**	**add**	**add**		
	IE		**lmw**	**lmw**	**lmw**	**add**	
	IC			**lmw**	**lmw**	**lmw**	**add**
	IWA						**add**
	IWL				**lmw**	**lmw**	**lmw**
	FPSB						
	ISB						

In the following code sequence, a load multiple instruction is followed by an instruction with a dependency on register 31:

```
Start:  lmw r29, x'0'(r3)
        add r31=r31 + x'0001'
```

Table I-80 shows the instruction timing for this example. Note that the **add** instruction stalls for two cycles in the ID stage because the IE stage is busy and stalls for one cycle in the IE stage because of the GPR dependency.

Table I-80. Load Multiple Instruction Followed by an ALU Instruction with a Register Dependency on GPR31

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5,6**	**7**	**8**
Memory	FA							
	CARB		**lmw**	**lmw**	**lmw**			
	CACC			**lmw**	**lmw**	**lmw**		
Dispatch	IQ1	**add**						
	IQ0	**lmw**	**add**					
IU	ID	**lmw**	**add**	**add**	**add**			
	IE		**lmw**	**lmw**	**lmw**	**add**	**add**	
	IC			**lmw**	**lmw**	**lmw**		**add**
	IWA							**add**
	IWL				**lmw**	**lmw**	**lmw**	
	FPSB							
	ISB							
Notes			**add** stalls because IE is busy.			**add** stalls due to a dependency on GPR31.		

I.2.3 Store Source Dependencies

Store instructions do not write the register file like loads do, so subsequent instructions do not have register hazards with a store. A subsequent instruction will not stall behind a store because of a register hazard.

An interesting code sequence is shown below; a store immediately follows an ALU instruction (**add**) and writes the ALU result to memory. There is no stall in the pipeline for this code sequence.

The following code sequence shows an ALU instruction followed by a dependent store:

```
Start: add r6=r6 + x'1'
       stw MEMx'4'(r3)=r6
```

Table I-81 shows the instruction timing for this example. Note that despite the fact that the **stw** instruction is dependent on the result of the **add** instruction, there is no stall.

Table I-81. ALU Instruction Followed by a Store Instruction Containing a Dependency on the ALU Target

Pipeline Stage		Cycle Number			
Unit	**Stage**	**1**	**2**	**3**	**4**
Memory	FA				
	CARB			**stw**	
	CACC				**stw**
Dispatch	IQ1	**stw**			
	IQ0	**add**	**stw**		
IU	ID	**add**	**stw**		
	IE		**add**	**stw**	
	IC			**add**	**stw**
	IWA			**add**	
	IWL				
	FPSB				
	ISB				

Stores may occur back-to-back without any pipeline stalls. An example is shown in the following code sequence:

```
Start: stw MEM[x'0'(r2)]=r1
       stw MEM[x'0'(r3)]=r2
```

Table I-82 shows the instruction timing for this example. Note that in this example there are no register dependencies and therefore there are no stalls.

Table I-82. Two Stores with No Register Dependencies

Pipeline Stage		Cycle Number			
Unit	**Stage**	**1**	**2**	**3**	**4**
Memory	FA				
	CARB		**stw** 1	**stw** 2	
	CACC			**stw** 1	stw 2
Dispatch	IQ1	**stw** 2			
	IQ0	**stw** 1	**stw** 2		
IU	ID	**stw** 1	**stw** 2		
	IE		**stw** 1	**stw** 2	
	IC			**stw** 1	**stw** 2
	IWA			**stw** 1	**stw** 2
	IWL				
	FPSB				
	ISB				

I.2.3.1 Load and Store Operations that Use the Update Option

Load and store operations that use the update option have timings similar to both ALU instructions and regular load and store operations. Update form loads/stores write the addressing register (**r**A) with the effective address; however because of feed-forwarding, no pipeline stalls occur when an instruction that immediately follows needs to use **r**A. There is a one-cycle stall for instructions that immediately follow a load and use the load target data, just as for regular loads.

The following code sequence shows a load with update instruction followed by an operation that is dependent on update register:

```
Start: lwzu r1=MEM[r2, r3]
       add r4=r2, r0
```

Table I-83 shows the instruction timing for this code sequence. Note that although there is a dependency on the GPR, there is no stall.

Table I-83. Load with Update Followed by an ALU Dependent on the Load rA

Pipeline Stage		Cycle Number			
Unit	**Stage**	**1**	**2**	**3**	**4**
Memory	FA				
	CARB		**lwzu**		
	CACC			**lwzu**	
Dispatch	IQ1	**add**			
	IQ0	**lwzu**	**add**		
IU	ID	**lwzu**	**add**		
	IE		**lwzu**	**add**	
	IC			**lwzu**	**add**
	IWA			**lwzu**	**add**
	IWL				**lwzu**
	FPSB				
	ISB				

The following code sequence shows consecutive store with update (**stwu**) instructions. In this example, the second instruction is dependent on the update register:

```
Start: stwu MEM[x'0'(r6)]=r2
       stwu MEM[x'0'(r6)]=r4
```

Table I-84 shows the instruction timing for this example. Note that although there is a GPR dependency, there is no stall.

Table I-84. Consecutive Store with Update Instruction with a Dependency on the Updated Register from the First Store Instruction

Pipeline Stage		Cycle Number			
Unit	**Stage**	**1**	**2**	**3**	**4**
Memory	FA				
	CARB		**stwu** 1	**stwu** 2	
	CACC			**stwu** 1	**stwu** 2
Dispatch	IQ1	**stwu** 2			
	IQ0	**stwu** 1	**stwu** 2		
IU	ID	**stwu** 1	**stwu** 2		
	IE		**stwu** 1	**stwu** 2	
	IC			**stwu** 1	**stwu** 2
	IWA			**stwu** 1	**stwu** 2
	IWL				
	FPSB				
	ISB				

The following code sequence shows a store with update instruction (**stwo**) followed by an instruction that is dependent on the update register:

```
Start: stwu MEM[r2, r3]=r1
       add r4=r2 + r4
```

Table I-85 shows the instruction timing for this example. Note that despite the dependency on the GPR, there is no stall.

Table I-85. Store with Update Followed by an ALU Instruction with a Dependency on the Updated Register from the Store

Pipeline Stage		Cycle Number			
Unit	**Stage**	**1**	**2**	**3**	**4**
Memory	FA				
	CARB		**stwu**		
	CACC			**stwu**	
Dispatch	IQ1	**add**			
	IQ0	**stwu**	**add**		

Table I-85. Store with Update Followed by an ALU Instruction with a Dependency on the Updated Register from the Store (Continued)

Pipeline Stage		Cycle Number			
IU	ID	**stwu**	**add**		
	IE		**stwu**	**add**	
	IC			**stwu**	**add**
	IWA			**stwu**	**add**
	IWL				
	FPSB				
	ISB				

The following code sequence illustrates a load with update instruction followed by an operation that is dependent on load target register):

```
Start: lwux r1=MEM[r2, r3]
       xor. r10=r1, r6
```

Table I-86 shows the instruction timing for this example. Note that because of the dependency on the GPR that is the target of the **lwux** instruction, the **xor.** instruction stalls for one cycle in the IE stage.

Table I-86. Load with Update Followed by an ALU Operation with a GPR Hazard on Load Target

Pipeline Stage		Cycle Number				
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**
Memory	FA					
	CARB		**lwux**			
	CACC			**lwux**		
Dispatch	IQ1	**xor.**				
	IQ0	**lwux**	**xor.**			
IU	ID	**lwux**	**xor.**			
	IE		**lwux**	**xor.**	**xor.**	
	IC			**lwux**		**xor.**
	IWA			**lwux**		**xor.**
	IWL				**lwux**	
	FPSB					
	ISB					

I.2.3.2 Load/Store Collisions

Load or store operations that access the same memory location (and hit in the cache) can be next to each other in program order and not cause a pipeline stall or have any additional latency.

The following code sequence shows a load instruction followed by a store instruction using the same address:

```
Start:  lwz r1=MEM[x'0'(r2)]
        stw MEM[x'0'(r2)]=r6
```

Table I-87 shows the instruction timing for this example. Note that although both instructions use the same address there is no pipeline stall.

Table I-87. A Load Operation Followed by a Store Operation to the Same Address (Cache Hit)

Pipeline Stage		Cycle Number			
Unit	**Stage**	**1**	**2**	**3**	**4**
Memory	FA				
	CARB		**lwz**	**stw**	
	CACC			**lwz**	**stw**
Dispatch	IQ1	**stw**			
	IQ0	**lwz**	**stw**		
IU	ID	**lwz**	**stw**		
	IE		**lwz**	**stw**	
	IC			**lwz**	**stw**
	IWA				
	IWL				**lwz**
	FPSB				
	ISB				

The following code example shows a store instruction followed by a load instruction from the same address:

```
Start:  stw MEM[r4, r6]=r2
        lwz r7=MEM[r4, r6]
```

Table I-88 shows the instruction timing for this example. Note that although both instructions use the same address there is no pipeline stall.

Table I-88. Store Operation Followed by a Load to the Same Address (Cache Hit)

Pipeline Stage		Cycle Number				
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**
Memory	FA					
	CARB		**stw**	**lwz**		
	CACC			**stw**	**lwz**	
Dispatch	IQ1	**lwz**				
	IQ0	**stw**	**lwz**			
IU	ID	**stw**	**lwz**			
	IE		**stw**	**lwz**		
	IC			**stw**	**lwz**	
	IWA					
	IWL					**lwz**
	FPSB					
	ISB					

I.3 Floating-Point Instruction Timing Examples

This section shows the timing for code sequences that contain sequences of floating-point instructions.

I.3.1 Floating-Point Data Dependency Stalls

The following timing examples involve only floating-point instructions. Table I-89 shows the instruction timing for two consecutive double-precision floating-point accumulate (**fmadd**) instructions. Note that these instructions require two cycles in each of the FD, FPM, and FPA stages, but are self-pipelining. That is, the second cycle of the FD stage is simultaneous with the first cycle of the FPM stage. This allows the second instruction to enter the FPM stage in cycle 3. However, because there is a RAW dependency, the second instruction cannot enter the FPM stage until the second clock cycle after the first instruction completes the write-back stage. Without this dependency, the second instruction would have entered the FPM stage in cycle 4.

Table I-89. Double-Precision Accumulate Instructions with RAW Data Dependency (Two fmadd Instructions)

Cycle Number	1	2	3	4	5
F1		**fmadd** 2			
FD	**fmadd** 1	**fmadd** 1	**fmadd** 2	**fmadd** 2	**fmadd** 2
FPM		**fmadd** 1	**fmadd** 1		
FPA			**fmadd** 1	**fmadd** 1	
FW					**fmadd** 1
Resources required nonexclusively	**fr**A1, **fr**B1, **fr**C1	**fr**A1, **fr**B1, **fr**C1			
Resources required exclusively					**fr**A2

Table I-89. Double-Precision Accumulate Instructions with RAW Data Dependency (Two fmadd Instructions) (Continued)

Cycle Number	6	7	8	9	10
F1					
FD	**fmadd** 2	**fmadd** 2			
FPM		**fmadd** 2	**fmadd** 2		
FPA			**fmadd** 2	**fmadd** 2	
FW					**fmadd** 2
Resources required nonexclusively	**fr**A2, **fr**B2, **fr**C2	**fr**A2, **fr**B2, **fr**C2			
Resources required exclusively					**fr**D2

Table I-90 shows the instruction timing for two consecutive single-precision add (**fadds**) instructions with a RAW dependency. This scenario is similar to the previous example in that the second example cannot enter the FPM cycle until the second cycle after the first instruction completes the FW stage.

Table I-90. Single-Precision Add Instructions with RAW Data Dependency (Two fadds Instructions)

Cycle Number	1	2	3	4	5	6	7	8
F1								
FD	**fadds** 1	**fadds** 2	**fadds** 2	**fadds** 2	**fadds** 2			
FPM		**fadds** 1				**fadds** 2		
FPA			**fadds** 1				**fadds** 2	
FW				**fadds** 1				**fadds** 2
Resources required nonexclusively	**fr**A1, **fr**B1				**fr**A2, **fr**B2			
Resources required exclusively				**fr**A2				**fr**D2

Table I-91 shows the instruction timing when a RAW dependency exists between a single-precision add instruction (**fmadds**) and a move instruction (**fmr**). Note that, similar to the previous examples, this dependency prevents the second instruction from entering the FPM stage until the second cycle after the **fmadds** instruction completes the FW stage.

Table I-91. Move Instruction with RAW Data Dependency on Single-Precision Accumulate Instruction (fmadds, fmr)

Cycle Number	1	2	3	4	5	6	7	8
F1								
FD	**fmadds**	**fmr**	**fmr**	**fmr**	**fmr**			
FPM		**fmadds**				**fmr**		
FPA			**fmadds**				**fmr**	
FW				**fmadds**				**fmr**
Resources required nonexclusively	**fr**A1, **fr**B1, **fr**C1				**fr**B2			
Resources required exclusively				**fr**B2				**fr**D2

Table I-92 shows the instruction timing for when a RAW data dependency exists between a single-precision accumulate instruction and a store instruction. Note that although there is a dependency, there is no stall.

Table I-92. Store Instruction with RAW Data Dependency on Immediately Previous Single-Precision Accumulate Instruction (fmadds, stfs)

Cycle Number	1	2	3	4	5
F1					
FD	**fmadds**	**stfs**			
FPM		**fmadds**	**stfs**		
FPA			**fmadds**	**stfs**	
FW				**fmadds**	**stfs**
Resources required nonexclusively	**fr**A1, **fr**B1, **fr**C_1				
Resources required exclusively				**fr**B2—note that **fr**B2 is also sent to memory subsystem.	

Table I-93 shows the instruction timing for the previous scenario when an intervening instruction is scheduled, in this instance an **fsub** instruction. Note that the **stfs** instruction cannot enter the FPM stage until the second cycle after the **fsub** instruction completes the FW stage.

Table I-93. Store Instruction with RAW Data Dependency on Previous Single-Precision Accumulate Instruction with an Intervening Operation (fmadds, fsub, stfs)

Cycle Number	1	2	3	4	5	6	7	8
F1								
FD	**fmadds**	**fsub**	**stfs**	**stfs**	**stfs**			
FPM		**fmadds**	**fsub**			**stfs**		
FPA			**fmadds**	**fsub**			**stfs**	
FW				**fmadds**	**fsub**			**stfs**
Resources required nonexclusively	**fr**A1, **fr**B1, **fr**C1	**fr**A2, **fr**B2			**fr**B3			
Resources required exclusively				**fr**B3	**fr**D2			

I.3.2 Timing Examples Involving Floating-Point Code Sequences

LINPACK is a standard floating-point benchmark which has an inner loop where a large portion of the execution time is spent. The follow tables show both single- and double-precision nonunrolled LINPACK loops.

The examples of nonoptimized loops show the importance of understanding the entire pipeline and how different stages interact in order to understand the more subtle aspects of the 601's instruction timing. Even though the 601 generally performs single-precision floating-point arithmetic faster than double-precision floating-point arithmetic, the nonoptimized, single-precision loop takes six cycles while the nonoptimized double-precision loop takes five. This is due to an interaction between the different units—the FPU finishes its instructions sooner, which allows the IU to execute its instructions sooner (the load instructions), which in turn causes the load instructions to interfere with the fetch of the next loop iteration, which actually causes stalls a few cycles later (due to a lack of instructions to execute). Simply by adding add the bnoop instruction (a conditional branch that is never taken), at the start of the loop in Case 3, performance of the single-precision loop is improved. Instead of a six-cycle-per-loop stable state (Case 2), there is a four-cycle-per-loop stable state (Case 3).

Note that this instruction never affects instruction timing after the loop execution is in a stable state (it is always folded out without stalls and causes no other instructions to stall). The only time this instruction has any effect is in the very first iteration (cycle 3). At that time, it blocks the loads from getting to the cache before the fetch of the second iteration (cycle 4). The only way to improve the performance of the loop beyond four-cycles-per-iteration is to use loop unrolling techniques (to reduce the fetch demands on the cache).

Table I-94 shows the timing for a nonoptimized, double-precision LINPACK loop (Case 1), using the following code example:

```
Start:  lfd f1=mem(r5+x'0080')
        lfd f2=mem(r5+x'07cf')
        fmadd f3=(f1*f2)+f3
        stfd mem(r5+x'07d0')=f3
        bcdnz Start
```

Table I-94. Double-Precision LINPACK Loop (Not Unrolled)—Case 1

Pipeline Stage		Cycle Number				
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**
Memory	FA	Start:0			Start:1	Start:1
	CARB	**lfd**1, **lfd**2, **fmadd**,**stfd**, **bc**			**lfd**1	**lfd**2
	CACC		**lfd**1, **lfd**2, **fmadd**,**stfd**, **bc**			**lfd**1
Dispatch	IQ5					
	IQ4			**bc**		
	IQ3			**stfd**		
	IQ2			**fmadd**	**bc**	
	IQ1			**lfd**2	**stfd**	
	IQ0			**lfd**1	**lfd**2, Tag**fmadd**	**stfd**, Tag**stfd**, Tag**bc**
IU	ID			**lfd**1	**lfd**2, Tag**fmadd**	**stfd**, Tag**stfd**, Tag**bc**
	IE				**lfd**1	**lfd**2, Tag**fmadd**
	IC					**lfd**1
	IWA					
	IWL					
	FPSB					
	ISB					
BPU	BE				**bc**	
	MR					
	BW					**bc**
FPU	F1					**stfd**
	FD				**fmadd**	**fmadd**
	FPM					
	FPA					
	FWA					
	FWL					
Notes					**fmadd** stalls in FD waiting for load data	Second iteration stalls in FA because of loads in the cache

Table I-94. Double-Precision LINPACK Loop (Not Unrolled)—Case 1 (Continued)

Pipeline Stage		Cycle Number			
Unit	**Stage**	**6**	**7**	**8**	**9**
Memory	FA	Start:1			Start:2
	CARB	**lfd**1, **lfd**2, **fmadd**,**stfd**,**bc**			**lfd**1, **lfd**2, **fmadd**,**stfd**,**bc**
	CACC	**lfd**2	**lfd**1, **lfd**2, **fmadd**,**stfd**,**bc**		
Dispatch	IQ5				
	IQ4			**bc**	
	IQ3			**stfd**	**bc**
	IQ2			**fmadd**	**stfd**
	IQ1			**lfd**2	**fmadd**
	IQ0			**lfd**1	**lfd**2
IU	ID			**lfd**1	**lfd**2
	IE	**stfd**, Tag**stfd**, Tag**bc**			**lfd**1
	IC	**lfd**2, Tag**fmadd**	**stfd**, Tag**stfd**, Tag**bc**		
	IWA				
	IWL				
	FPSB		**stfd**	**stfd**	**stfd**
	ISB				
BPU	BE				**bc**
	MR				
	BW	**bc**	**bc**		
FPU	F1	**stfd**	**stfd**	**stfd**	
	FD	**fmadd**	**fmadd**	**fmadd**	**stfd**
	FPM			**fmadd**	**fmadd**
	FPA				**fmadd**
	FWA				
	FWL	**lfd**1	**lfd**2		
Notes			**bc** writes-back to CTR		**lfd** stalls in IE because of a WAR hazard with the **fmadd** in FD

Table I-94. Double-Precision LINPACK Loop (Not Unrolled)—Case 1 (Continued)

Pipeline Stage		Cycle Number				
Unit	**Stage**	**10**	**11**	**12**	**13**	**14**
Memory	FA					Start:3
	CARB	**lfd**1	**lfd**2	**stfd**		**lfd**1, **lfd**2, **fmadd**,**stfd**,**bc**
	CACC	**lfd**1, **lfd**2, **fmadd**,**stfd**,**bc**	**lfd**1	**lfd**2	**stfd**	
Dispatch	IQ5		**bc**			
	IQ4		**stfd**	**bc**	**bc**	
	IQ3		**fmadd**	**stfd**	**stfd**	**bc**
	IQ2		**lfd**2	**fmadd**	**fmadd**	**stfd**
	IQ1	**stfd**, Tag**bc**	**lfd**1	**lfd**2	**lfd**2	**fmadd**
	IQ0	**lfd**2, Tag**fmadd**	**stfd**, Tag**stfd**, Tag**bc**	**lfd**1	**lfd**1	**lfd**2
IU	ID	**lfd**2, Tag**fmadd**	**stfd**, Tag**stfd**, Tag**bc**	**lfd**1	**lfd**1	**lfd**2
	IE	**lfd**1	**lfd**2, Tag**fmadd**	**stfd**, Tag**stfd**, Tag**bc**	**stfd**, Tag**stfd**, Tag**bc**	**lfd**1
	IC		**lfd**1	**lfd**2, Tag**fmadd**		**stfd**, Tag**stfd**, Tag**bc**
	IWA					
	IWL					
	FPSB	**stfd**	**stfd**	**stfd**		**stfd**
	ISB					
BPU	BE					**bc**
	MR					
	BW	**bc**	**bc**	**bc**	**bc**	**bc**
FPU	F1		**stfd**	**stfd**	**stfd**	**stfd**
	FD	**fmadd**	**fmadd**	**fmadd**	**fmadd**	**fmadd**
	FPM	**stfd**				**fmadd**
	FPA	**fmadd**	**stfd**			
	FWA		**fmadd**	**stfd**		
	FWL			**lfd**1	**lfd**2	
Notes			**fmadd** stalls in FD waiting for load data	data for **stfd** is sent to memory subsystem.		**lfd** stalls in IE due to WAR hazard with the **fmadd** in FD

Table I-94. Double-Precision LINPACK Loop (Not Unrolled)—Case 1 (Continued)

Pipeline Stage		Cycle Number			
Unit	**Stage**	**15**	**16**	**17**	**18**
Memory	FA				Start:5
	CARB	**lfd**1	**lfd**2		**stfd**
	CACC	**lfd**1, **lfd**2, **fmadd**,**stfd**,**bc**	**lfd**1	**lfd**2	
Dispatch	IQ5		**bc**		
	IQ4		**stfd**	**bc**	
	IQ3		**fmadd**	**stfd**	**bc**
	IQ2		**lfd**2	**fmadd**	**stfd**
	IQ1	**stfd**, Tag**bc**	**lfd**1	**lfd**2	**fmadd**
	IQ0	**fmadd**	**stfd**, Tag**bc**	**lfd**1	**lfd**2
IU	ID	**lfd**2	**stfd**, Tag**bc**	**lfd**1	**lfd**1
	IE	**lfd**1	**lfd**2, Tag**fmadd**	**stfd**, Tag**stfd**, Tag**bc**	**stfd**, Tag**stfd**, Tag**bc**
	IC		**lfd**1	**lfd**2, Tag**fmadd**	
	IWA				
	IWL				
	FPSB	**stfd**	**stfd**	**stfd**	**stfd**
	ISB				
BPU	BE				
	MR				
	BW	**bc**	**bc**	**bc**	**bc**
FPU	F1			**stfd**	**stfd**
	FD	**stfd**	**fmadd**	**fmadd**	**fmadd**
	FPM	**fmadd**	**stfd**		
	FPA	**fmadd**	**fmadd**	**stfd**	
	FWA			**fmadd**	**stfd**
	FWL			**lfd**1	**lfd**2
Notes					data for **stfd** is sent to memory subsystem.

Table I-94. Double-Precision LINPACK Loop (Not Unrolled)—Case 1 (Continued)

Pipeline Stage		Cycle Number		
Unit	**Stage**	**19**	**20**	**21**
Memory	FA	Start:5		
	CARB	**lfd**1, **lfd**2, **fmadd**,**stfd**,**bc**	**lfd**1	**lfd**2
	CACC	**stfd**	**lfd**1, **lfd**2, **fmadd**,**stfd**,**bc**	**lfd**1
Dispatch	IQ5			**bc**
	IQ4			**stfd**
	IQ3	**bc**	**bc**	**fmadd**
	IQ2	**stfd**	**stfd**	**lfd**2
	IQ1	**fmadd**	**fmadd**	**lfd**1
	IQ0	**lfd**2	**lfd**2	**stfd**, Tag**bc**
IU	ID	**lfd**1	**lfd**2	**stfd**, Tag**bc**
	IE	**stfd**, Tag**stfd**, Tag**bc**	**lfd**1	**lfd**2, Tag**fmadd**
	IC		**stfd**, Tag**stfd**, Tag**bc**	**lfd**1
	IWA			
	IWL			
	FPSB		**stfd**	**stfd**
	ISB			
BPU	BE		**bc**	
	MR			
	BW	**bc**	**bc**	**bc**
FPU	F1	**stfd**		
	FD	**fmadd**	**stfd**	**fmadd**
	FPM	**fmadd**	**fmadd**	**stfd**
	FPA		**fmadd**	**fmadd**
	FWA			
	FWL			
Notes				This state is the same as in cycle 16

Table I-95 shows the instruction timing for a nonoptimized, single-precision LINPACK loop (Case 2), using the following code sequence:

```
Start: lfs f1=mem(r5+x'0080')
       lfs f2=mem(r5+x'07cf')
       fmadds f3=(f1*f2)+f3
       stfs mem(r5+x'07d0')=f3
       bcdnz Start
```

Table I-95. Single-Precision LINPACK Loop (Nonoptimized)—Case 2

Pipeline Stage		Cycle Number					
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**
Memory	FA	Start:0			Start:1	Start:1	Start:1
	CARB	**lfs**1,**lfs**2, **fmadds**, **stfs**, **bc**			**lfs**1	**lfs**2	**lfs**1,**lfs**2, **fmadds**,**stfs**, **bc**
	CACC		**lfs**1,**lfs**2, **fmadds**, **stfs**, **bc**			**lfs**1	**lfs**2
Dispatch	IQ4			**bc**			
	IQ3			**stfs**			
	IQ2			**fmadds**	**bc**		
	IQ1			**lfs**2	**stfs**		
	IQ0			**lfs**1	**lfs**2, Tag**fmadds**	**stfs**, Tag**stfs**, Tag**bc**	
IU	ID			**lfs**1	**lfs**2, Tag**fmadds**	**stfs**, Tag**stfs**, Tag**bc**	
	IE				**lfs**1	**lfs**2, Tag**fmadds**	**stfs**, Tag**stfs**, Tag**bc**
	IC					**lfs**1	**lfs**2, Tag**fmadds**
	IWA						
	IWL						
	FPSB						
	ISB						
BPU	BE				**bc**		
	MR						
	BW					**bc**	**bc**
FPU	F1					**stfs**	**stfs**
	FD				**fmadds**	**fmadds**	**fmadds**
	FPM						
	FPA						
	FWA						
	FWL						**lfs**1
Notes					**fmadds** stalls in FD waiting for load data	Second iteration stalls in FA due to loads in the cache	

Table I-95. Single-Precision LINPACK Loop (Nonoptimized)—Case 2 (Continued)

Pipeline Stage		Cycle Number				
Unit	**Stage**	**7**	**8**	**9**	**10**	**11**
Memory	FA			Start:2	Start:2	Start:2
	CARB			**lfs**1	**lfs**2	**stfs**
	CACC	**lfs**1,**lfs**2, **fmadds**,**stfs**, **bc**			**lfs**1	**lfs**2
Dispatch	IQ4		**bc**			
	IQ3		**stfs**			
	IQ2		**fmadds**	**bc**		
	IQ1		**lfs**2	**stfs**		
	IQ0		**lfs**1	**lfs**2, Tag**fmadds**	**stfs**, Tag**stfs**, Tag**bc**	
IU	ID		**lfs**1	**lfs**2, Tag**fmadds**	**stfs**, Tag**stfs**, Tag**bc**	
	IE			**lfs**1	**lfs**2, Tag**fmadds**	**stfs**, Tag**stfs**, Tag**bc**
	IC	**stfd**, Tag**stfs**, Tag**bc**			**lfs**1	**lfs**2, Tag**fmadds**
	IWA					
	IWL					
	FPSB	**stfs**	**stfs**	**stfs**	**stfs**	**stfs**
	ISB					
BPU	BE			**bc**		
	MR					
	BW	**bc**			**bc**	**bc**
FPU	F1	**stfs**			**stfs**	**stfs**
	FD	**fmadds**	**stfs**	**fmadds**	**fmadds**	**fmadds**
	FPM		**fmadds**	**stfs**		
	FPA			**fmadds**	**stfs**	
	FWA				**fmadds**	**stfs**
	FWL	**lfs**2				**lfs**1
Notes		**bc** writes back to CTR		**fmadds** stalls in FD waiting for load data		data for **stfs** is sent to memory subsystem.

Table I-95. Single-Precision LINPACK Loop (Nonoptimized)—Case 2 (Continued)

Pipeline Stage		Cycle Number				
Unit	**Stage**	**12**	**13**	**14**	**15**	**16**
Memory	CARB	Start:2			Start:3	Start:3
	CARB	**lfs**1,**lfs**2, **fmadds**,**stfs**, **bc**			**lfs**1	**lfs**2
	CACC	**stfs**	**lfs**1,**lfs**2, **fmadds**,**stfs**, **bc**			**lfs**1
Dispatch	IQ4			**bc**		
	IQ3			**stfs**		
	IQ2			**fmadds**	**bc**	
	IQ1			**lfs**2	**stfs**	
	IQ0			**lfs**1	**lfs**2, Tag**fmadds**	**stfs**, Tag**stfs**, Tag**bc**
IU	ID			**lfs**1	**lfs**2, Tag**fmadds**	**stfs**, Tag**stfs**, Tag**bc**
	IE	**stfs**, Tag**stfs**, Tag**bc**			**lfs**1	**lfs**2, Tag**fmadds**
	IC		**stfs**, Tag**stfs**, Tag**bc**			**lfs**1
	IWA					
	IWL					
	FPSB		**stfs**	**stfs**	**stfs**	**stfs**
	ISB					
BPU	BE				**bc**	
	MR					
	BW	**bc**	**bc**			**bc**
FPU	F1	**stfs**				**stfs**
	FD	**fmadds**	**stfs**		**fmadds**	**fmadds**
	FPM		**fmadds**	**stfs**		
	FPA			**fmadds**	**stfs**	
	FWA				**fmadds**	**stfs**
	FWL	**lfs**2				
Notes					**fmadds** stalls in FD waiting for load data	data for **stfs** is sent to memory subsystem.

Table I-95. Single-Precision LINPACK Loop (Nonoptimized)—Case 2 (Continued)

Pipeline Stage		Cycle Number				
Unit	**Stage**	**17**	**18**	**19**	**20**	**21**
Memory	CARB	Start:3	Start:3			
	CARB	**stfs**	**lfs**1,**lfs**2, **fmadds**,**stfs**, **bc**			**lfs**1
	CACC	**lfs**2	**stfs**	**lfs**1,**lfs**2, **fmadds**,**stfs**, **bc**		
Dispatch	IQ4				**bc**	
	IQ3				**stfs**	
	IQ2				**fmadds**	**bc**
	IQ1				**lfs**2	**stfs**
	IQ0				**lfs**1	**lfs**2, Tag**fmadd**
IU	ID				**lfs**1	**lfs**2, Tag**fmadd**
	IE	**stfs**, Tag**stfs**, Tag**bc**	**stfs**, Tag**stfs**, Tag**bc**			**lfs**1
	IC	**lfs**2, Tag**fmadds**		**stfs**, Tag**stfs**, Tag**bc**		
	IWA					
	IWL					
	FPSB	**stfs**		**stfs**	**stfs**	**stfs**
	ISB					
BPU	BE					**bc**
	MR					
	BW	**bc**	**bc**	**bc**		
FPU	F1	**stfs**	**stfs**			
	FD	**fmadds**	**fmadds**	**stfs**		**fmadds**
	FPM			**fmadds**	**stfs**	
	FPA				**fmadds**	**stfs**
	FWA					**fmadds**
	FWL	**lfs**1	**lfs**2			
Notes						this state is the same as the state in cycle 15.

Table I-96 shows the instruction timing for a semi-optimized, single-precision LINPACK loop (Case 3), using the following code sequence:

```
Start: bnoop (bc which is never taken)
       lfs f1=mem(r5+x'0080')
       lfs f2=mem(r5+x'07cf')
       fmadds f3=(f1*f2)+f3
       stfs mem(r5+x'07d0')=f3
       bcdnz Start
```

Note that except for the initial bnoop (an untaken **bc** instruction), this code sequence is identical to that of Case 2. The insertion of bnoop improves the performance of the single-precision loop. Instead of a six-cycle-per-loop stable state (Case 2), there is a four-cycle-per-loop stable state (Case 3).

Table I-96. Single-Precision LINPACK Loop (Semi-Optimized)

Pipeline Stage		Cycle Number				
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**
Memory	FA	Start:0			Start:1	
	CARB	bnoop,**lfs**1, **lfs**2,**fmadds**, **stfs**,**bc**			bnoop,**lfs**1, **lfs**2,**fmadds**, **stfs**,**bc**	**lfs**1
	CACC		bnoop,**lfs**1, **lfs**2,**fmadds**, **stfs**,**bc**			bnoop,**lfs**1, **lfs**2,**fmadds**, **stfs**,**bc**
Dispatch	IQ6					
	IQ5			**bc**		
	IQ4			**stfs**		
	IQ3			**fmadds**	**bc**	
	IQ2			**lfs**2	**stfs**	
	IQ1			**lfs**1	**lfs**2, Tag**fmadds**	**stfs**, Tag**stfs**, Tag**bc**
	IQ0			bnoop	**lfs**1	**lfs**2, Tag**fmadds**
IU	ID				**lfs**1	**lfs**2, Tag**fmadds**
	IE					**lfs**1
	IC					
	IWA					
	IWL					
	FPSB					
	ISB					
BPU	BE			bnoop	**bc**	
	MR			bnoop		
	BW					**bc**
FPU	F1					**stfs**
	FD				**fmadds**	**fmadds**
	FPM					
	FPA					
	FWA					
	FWL					
Notes					**fmadds** stalls in FD waiting for load data	

Table I-96. Single-Precision LINPACK Loop (Semi-Optimized) (Continued)

Pipeline Stage		Cycle Number			
Unit	**Stage**	**6**	**7**	**8**	**9**
Memory	FA			Start:2	
	CARB	**lfs**2		bnoop,**lfs**1, **lfs**2,**fmadds**, **stfs**,**bc**	**lfs**1
	CACC	**lfs**1	**lfs**2		bnoop,**lfs**1, **lfs**2,**fmadds**, **stfs**,**bc**
Dispatch	IQ6	**bc**			
	IQ5	**stfs**			
	IQ4	**fmadds**	**bc**		
	IQ3	**lfs**2	**stfs**	**bc**	
	IQ2	**lfs**1	**fmadds**	**stfs**	
	IQ1	bnoop	**lfs**2	**fmadds**	**stfs**, Tag**bc**
	IQ0	**stfs**, Tag**stfs**, Tag**bc**	**lfs**1	**lfs**2	**fmadds**
IU	ID	**stfs**, Tag**stfs**, Tag**bc**	**lfs**1	**lfs**2	**lfs**2
	IE	**lfs**2, Tag**fmadds**	**stfs**, Tag**stfs**, Tag**bc**	**lfs**1	**lfs**1
	IC	**lfs**1	**lfs**2, Tag**fmadds**	**stfs**, Tag**stfs**, Tag**bc**	
	IWA				
	IWL				
	FPSB			**stfs**	**stfs**
	ISB				
BPU	BE	bnoop		**bc**	
	MR	bnoop			
	BW	**bc**	**bc**		**bc**
FPU	F1	**stfs**	**stfs**	**stfs**	
	FD	**fmadds**	**fmadds**	**fmadds**	**stfs**
	FPM				**fmadds**
	FPA				
	FWA				
	FWL		**lfs**1	**lfs**2	
Notes					**fmadds** stalls in FD waiting for load data

Table I-96. Single-Precision LINPACK Loop (Semi-Optimized) (Continued)

Pipeline Stage		Cycle Number				
Unit	**Stage**	**10**	**11**	**12**	**13**	**14**
Memory	FA			Start:3		
	CARB	**lfs**2	**stfs**	bnoop,**lfs**1, **lfs**2,**fmadds**, **stfs**,**bc**	**lfs**1	**lfs**2
	CACC	**lfs**1	**lfs**2	**stfs**	bnoop,**lfs**1, **lfs**2,**fmadds**, **stfs**,**bc**	**lfs**1
Dispatch	IQ6	**bc**				**bc**
	IQ5	**stfs**				**stfs**
	IQ4	**fmadds**	**bc**			**fmadds**
	IQ3	**lfs**2	**stfs**	**bc**		**lfs**2
	IQ2	**lfs**1	**fmadds**	**stfs**	**stfs**, Tag**bc**	**lfs**1
	IQ1	bnoop	**lfs**2	**fmadds**	**fmadds**	bnoop
	IQ0	**stfs**, Tag**bc**	**lfs**1	**lfs**2	**lfs**2	**stfs**, Tag**bc**
IU	ID	**stfs**, Tag**bc**	**lfs**1	**lfs**1	**lfs**2	**stfs**, Tag**bc**
	IE	**lfs**2, Tag**fmadds**	**stfs**, Tag**stfs**, Tag**bc**	**stfs**, Tag**stfs**, Tag**bc**	**lfs**1	**lfs**2, Tag**fmadds**
	IC	**lfs**1	**lfs**2		**stfs**, Tag**stfs**, Tag**bc**	**lfs**1
	IWA					
	IWL					
	FPSB	**stfs**	**stfs**		**stfs**	**stfs**
	ISB					
BPU	BE	bnoop		**bc**		bnoop
	MR	bnoop				bnoop
	BW	**bc**	**bc**	**bc**	**bc**	**bc**
FPU	F1		**stfs**	**stfs**		
	FD	**fmadds**	**fmadds**	**fmadds**	**stfs**	**fmadds**
	FPM	**stfs**			**fmadds**	**stfs**
	FPA	**fmadds**	**stfs**			**fmadds**
	FWA		**fmadds**	**stfs**		
	FWL		**lfs**1	**lfs**2		
Notes			data for **stfs** is sent to memory subsystem.			State matches cycle 10.

The next code sequences shows the interactions between load and store operations. Table I-97 shows the instruction timing for a load feeding a compare inside a loop, using the following code sequence:

```
start:  lw r0=mem(r3,4)
        ori r5=r3
        cmp cr0=r5,r0
        addi r0=0
        bc- out, cr0
        stw mem(r5,8)=r0
        lw r5=mem(r5,0)
        lw r4=mem(r3,4)
        cmp cr1=r5,r4
lab1:   bc- out, cr1
        stw mem(r5,8)=r0
        lw r5=mem(r5,0)
        lw r4=mem(r3,4)
        cmp cr1=r5,r4
        b lab1
out:
```

Table I-97. Load->Compare Operations in a Loop

Pipeline Stage		Cycle Number				
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**
Memory	FA	start:	start+8(*4)	start+9(*4)	start+10(*4)	start+11(*4)
	CARB	**lw**1,**ori**,**cmp**1, **addi**,**bc**1,**stw**1, **lw**2,**lw**3	**cmp**2,**bc**2, **stw**2,**lw**4,**lw**5, **cmp**3, **b**	**bc**2, **stw**2,**lw**4,**lw**5, **cmp**3, **b**	**lw**1	**lw**4,**lw**5, **cmp**3, **b**
	CACC		**lw**1,**ori**,**cmp**1, **addi**,**bc**1,**stw**1, **lw**2,**lw**3	**cmp**2,**bc**2, **stw**2,**lw**4,**lw**5, **cmp**3, **b**	**bc**2, **stw**2,**lw**4,**lw**5, **cmp**3, **b**	**lw**1
Dispatch	IQ7			**lw**3	**cmp**2	**stw**2
	IQ6			**lw**2	**lw**3	**bc**2
	IQ5			**stw**1	**lw**2	**cmp**2
	IQ4			**bc**1	**stw**1	**lw**3
	IQ3			**addi**	**bc**1	**lw**2
	IQ2			**cmp**1	**addi**	**stw**1
	IQ1			**ori**	**cmp**1	**addi**, Tagpbr
	IQ0			**lw**1	**ori**	**cmp**1
IU	ID			**lw**1	**ori**	**cmp**1
	IE				**lw**1	**ori**
	IC					**lw**1
	IWA					
	IWL					
	FPSB					
	ISB					
BPU	BE				**bc**1	
	MR				**bc**1	**bc**1
	BW					
Notes					**bc**1 is predicted not-taken	

Table I-97. Load->Compare Operations in a Loop (Continued)

Pipeline Stage			Cycle Number				
Unit	**Stage**	**6**	**7**	**8**	**9**	**10**	**11**
Memory	FA	start+13(*4)	start+14(*4)	start+16(*4)	start+16(*4)	start+16(*4)	start+16(*4)
	CARB	**cmp**3, **b**	**b**	**stw**1	**lw**2	**lw**3	start+16(*4)
	CACC	**lw**4,**lw**5, **cmp**3, **b**	**cmp**3, **b**	**b**	**stw**1	**lw**2	**lw**3
Dispatch	IQ7		**lw**5	**cmp**3			
	IQ6	**stw**2	**lw**4	**lw**5	**b**		
	IQ5	**bc**2	**stw**2	**lw**4	**cmp**3	**b**	
	IQ4	**cmp**2	**bc**2	**stw**2	**lw**5	**cmp**3	**b**
	IQ3	**lw**3	**cmp**2	**bc**2	**lw**4	**lw**5	**cmp**3
	IQ2	**lw**2	**lw**3	**cmp**2	**stw**2	**lw**4	**lw**5
	IQ1	**stw**1	**lw**2	**lw**3	**cmp**2, Tagpbr	**stw**2	**lw**4
	IQ0	**addi**, Tagpbr	**stw**1	**lw**2	**lw**3	**cmp**2, Tagpbr	**stw**2
IU	ID	**addi**, Tagpbr	**stw**1	**lw**2	**lw**3	**cmp**2, Tagpbr	**stw**2
	IE	**cmp**1	**addi**	**stw**1	**lw**2	**lw**3	**cmp**2, Tagpbr
	IC	**ori**	**cmp**1	**addi**	**stw**1	**lw**2	**lw**3
	IWA	**ori**	**cmp**1	**addi**			
	IWL	**lw**1					**lw**2
	FPSB						
	ISB						
BPU	BE			**bc**2			
	MR	**bc**1		**bc**2	**bc**2	**bc**2	**bc**2
	BW						
Notes		**cmp**1 result is forwarded to the MR stage; **bc**1 is resolved: predicted correctly		**bc**2 is predicted not-taken			**cmp**2 stalls in IE because of a data dependency on **lw**3.

Table I-97. Load->Compare Operations in a Loop (Continued)

Pipeline Stage		Cycle Number					
Unit	**Stage**	**12**	**13**	**14**	**15**	**16**	**17**
Memory	FA	lab1:1	lab1+7(*4)	lab1+7(*4)	lab1+7(*4)	lab1+7(*4)	lab1:2
	CARB	**bc**2, **stw**2,**lw**4,**lw**5, **cmp**3, **b**	**stw**2	**lw**4	**lw**5	lab1+7(*4)	**bc**2, **stw**2,**lw**4,**lw**5, **cmp**3, **b**
	CACC	start+16(*4)	**bc**2, **stw**2,**lw**4,**lw**5, **cmp**3, **b**	**stw**2	**lw**4	**lw**5	lab1+7(*4)
Dispatch	IQ7			**b**			
	IQ6			**cmp**3			
	IQ5			**lw**5	**b**		
	IQ4			**lw**4	**cmp**3	**b**	
	IQ3	**b**		**stw**2	**lw**5	**cmp**3	**b**
	IQ2	**cmp**3	**cmp**3	**bc**2	**lw**4	**lw**5	**cmp**3
	IQ1	**lw**5	**lw**5	**cmp**3	**stw**2	**lw**4	**lw**5
	IQ0	**lw**4	**lw**4	**lw**5	**cmp**3, Tagpbr	**stw**2	**lw**4
IU	ID	**stw**2	**lw**4	**lw**5	**cmp**3, Tagpbr	**stw**2	**stw**2
	IE	**cmp**2, Tagpbr	**stw**2	**lw**4	**lw**5	**cmp**3, Tagpbr	**cmp**3, Tagpbr
	IC		**cmp**2	**stw**2	**lw**4	**lw**5	
	IWA		**cmp**2				
	IWL	**lw**3				**lw**4	**lw**5
	FPSB						
	ISB						
BPU	BE	**b**		**bc**2			**b**
	MR	**bc**2		**bc**2	**bc**2	**bc**2	**bc**2
	BW						
Notes		**cmp**2 result is forwarded to the MR stage; **bc**2 is resolved: predicted correctly		**bc**2 is predicted not-taken		**cmp**3 stalls in IE because of a data dependency on **lw**5	**cmp**2 result is forwarded to the MR stage; **bc**2 is resolved: predicted correctly

Table I-97. Load->Compare Operations in a Loop (Continued)

Pipeline Stage		Cycle Number						
Unit	**Stage**	**18**	**19**	**20**	**21**	**22**	**23**	**24**
Memory	FA	lab1+7(*4)	lab1+7(*4)	lab1+7(*4)	lab1+7(*4)	out:		
	CARB	**stw**2	**lw**4	**lw**5	lab1+7(*4)	out:		
	CACC	**bc**2, **stw**2,**lw**4, **lw**5, **cmp**3, **b**	**stw**2	**lw**4	**lw**5	lab1+7(*4)	out:	
Dispatch	IQ7		**b**					
	IQ6		**cmp**3					
	IQ5		**lw**5	**b**				
	IQ4		**lw**4	**cmp**3	**b**			
	IQ3		**stw**2	**lw**5	**cmp**3	**b**		
	IQ2	**cmp**3	**bc**2	**lw**4	**lw**5	**cmp**3		
	IQ1	**lw**5	**cmp**3	**stw**2	**lw**4	**lw**5		
	IQ0	**lw**4	**lw**5	**cmp**3, Tagpbr	**stw**2	**lw**4		out:
IU	ID	**lw**4	**lw**5	**cmp**3, Tagpbr	**stw**2	**stw**2		out:
	IE	**stw**2	**lw**4	**lw**5	**cmp**3, Tagpbr	**cmp**3, Tagpbr		
	IC	**cmp**3	**stw**2	**lw**4	**lw**5		**cmp**3	
	IWA	**cmp**3					**cmp**3	
	IWL				**lw**4	**lw**5		
	FPSB							
	ISB							
BPU	BE		**bc**2			**b**		
	MR		**bc**2	**bc**2	**bc**2	**bc**2		
	BW							
Notes			**bc**2 is predicted not-taken; The state in this cycle is the same as the state in cycle 14.		**cmp**3 stalls in IE because of a data dependency on **lw**5	**cmp**2 result is forwarded to the MR stage; **bc**2 is resolved: predicted incorrectly		Code at exit target (out:) is in IQ0/ID.

I.4 Additional Code Sequence Timings

This section shows instruction timings for code sequences that illustrate certain features and code scheduling restrictions in the 601. This is primarily of interest to those writing code generators (such as compilers) and assembly language routines to optimize performance

Interlocks are implemented in the 601 to handle register dependencies that affect how instructions flow through the pipeline. One set of interlocks exist between floating-point loads (which go through the IU pipeline) and floating-point ALU instructions (that flow through the FPU pipeline). These interlocks are illustrated by the following code sequence:

Example 1:

```
lfdu   fr5,0(r3)
fnmadd fr4, fr5, fr6, fr2
lfdu   fr2, x'(r9)'          Load has WAR dependency previous op.
fsub   fr10, fr11, fr2
```

The timing for this code sequence is shown in Table I-98.

Table I-98. Interlocks between Floating-Point ALU and Subsequent Floating-Point Load Instruction (WAR Dependency)

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Memory	FA							
	CARB							
	CACC							
Dispatch	IQ3	**fsub**						
	IQ2	**lfdu** 2						
	IQ1	**fnmadd**	**fsub**					
	IQ0	**lfdu** 1	**lfdu** 2					
IU	ID	**lfdu** 1	**lfdu** 2					
	IE		**lfdu** 1	**lfdu** 2	**lfdu** 2	**lfdu** 2	**lfdu** 2	
	IC			**lfdu** 1				**lfdu** 2
	IWA			**lfdu** 1				**lfdu** 2
	IWL							
	FPSB							
	ISB							
FPU	F1			**fsub**	**fsub**	**fsub**		
	FD		**fnmadd**	**fnmadd**	**fnmadd**	**fnmadd**	**fsub**	**fsub**
	FPM					**fnmadd**	**fnmadd**	
	FPA						**fnmadd**	**fnmadd**
	FWA							
	FWL				**lfdu** 1			
Notes			**lfdu** 1 stalls waiting for target data from **lfdu**1.	The float load in IE is stalled waiting for the **fnmadd** to leave FD.		**fnmadd** proceeds now that all register data is available.	**lfdu** 2 can proceed since no instruction is in FD uses the load target register. **fsub** stalls for **lfdu**2.	**fsub** is stalled for **lfdu** 2 target data.

Table I-98. Interlocks between Floating-Point ALU and Subsequent Floating-Point Load Instruction (WAR Dependency) (Continued)

Pipeline Stage		Cycle Number			
Unit	**Stage**	**8**	**9**	**10**	**11**
Memory	FA				
	CARB				
	CACC				
Dispatch	IQ3				
	IQ2				
	IQ1				
	IQ0				
IU	ID				
	IE				
	IC				
	IWA				
	IWL				
	FPSB				
	ISB				
FPU	F1				
	FD	**fsub**			
	FPM		**fsub**		
	FPA			**fsub**	
	FWA	**fnmadd**			**fsub**
	FWL	**lfdu** 2			
Notes		The **fsub** is still waiting on the **lfdu** 2 data.	All data available for the **fsub** to start decode.		

There are three register dependencies in this code sequence—**fr**5 between the first and second instructions, **fr**2 between the second and third instructions, and **fr**2 between the third and fourth instructions. These register dependencies cause the following pipeline stalls.

- The **fnmadd** must stall in the FD stage until **fr**5 has been updated with the load target data.
- The second **lfdu** stalls in the IE stage until the **fnmadd** has left the FD stage. This interlock prevents the second **lfdu** from writing **fr**2 before it is read by the **fnmadd**. (This could happen if the first **lfdu** were to miss in the cache and the second **lfdu** were to hit in the cache; however, the second **lfdu** stalls for a couple of cycles in IE even if the first **lfdu** hit in the cache.)
- The register dependency between the third and fourth instruction causes the **fsub** to stall in the FD stage until the target data is available from the second **lfdu**.

The WAR (write after read) dependency between the second third instruction can be avoided by choosing a different load target for the third instruction—one that is not used by the preceding **fnmadd**. This is illustrated by slightly adjusting the previous code sequence, as shown in example 2.

Example 2:

```
lfdu   fr5,0(r3)
fnmadd fr4, fr5, fr6, fr2
lfdu   fr7, x'(r9)'  Load target is not in previous float ALU op.
fsub   fr10, fr11, fr7
```

The timing for this code sequence is shown in Table I-99.

Table I-99. Recoding of Sequence to Prevent lfdu 2 from Stalling in the IE Stage

Pipeline Stage		Cycle Number								
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Memory	FA									
	CARB		**lfdu** 1	**lfdu** 2						
	CACC			**lfdu** 1	**lfdu** 2					
Dispatch	IQ3	**fsub**								
	IQ2	**lfdu** 2								
	IQ1	**fnmadd**	**fsub**							
	IQ0	**lfdu** 1	**lfdu**2							
IU	ID	**lfdu** 1	**lfdu** 2							
	IE		**lfdu** 1	**lfdu** 2						
	IC			**lfdu** 1	**lfdu** 2					
	IWA			**lfdu** 1	**lfdu** 2					
	IWL									
	FPSB									
	ISB									
FPU	F1			**fsub**	**fsub**	**fsub**				
	FD		**fnmadd**	**fnmadd**	**fnmadd**	**fnmadd**	**fsub**			
	FPM					**fnmadd**	**fnmadd**	**fsub**		
	FPA						**fnmadd**	**fnmadd**	**fsub**	
	FWA								**fnmadd**	**fsub**
	FWL				**lfdu** 1	**lfdu** 2				
Notes			**fnmadd** stalls waiting for target data from **lfdu** 1.	**lfdu** 2 does not wait for **fnmadd** to leave FD because it has been recoded to use a distinct FPR	**fsub** stalls waiting on **fnmadd**	**fnmadd** proceeds now that all register data is available; **lfdu** 2 completes before **fsub** is in FD.				

The second **lfdu** can be sent to the cache without waiting on the **fnmadd** if it uses a target FPR that is not in the **fnmadd**. Note that it takes only 9 cycles to complete the recoded sequence, while it takes 11 cycles to complete the original sequence. Stall cycles are added to both the IU and the FPU pipelines.

Note that the RAW dependencies between the first and second instructions and the third and fourth instructions still exist in the recoded sequence because these are true data dependencies. Nondependent floating-point pipeline instructions can be moved in between the load and the ALU instruction that uses the load target to reduce the effects of the load latency.

The following code sequence has a nondependent instruction inserted between the load and the dependent operation. The nondependent instruction is executed during the cycles the FPU had been stalled in the previous example. This does not decrease the load latency, but rather, reduces the number of cycles that the FPU stalls. The nondependent floating-point ALU instruction was executed without increasing the total number of cycles (nine cycles total).

Example 3:

```
lfdu   fr5,0(r3)
fmul   f20, f21, f22
fnmadd fr4, fr5, fr6, fr2
lfdu   fr7, x'(r9)'          Loads moved further from dependent op.
fsub   fr10, fr11, fr7
```

The timing for this example is shown in Table I-100.

Table I-100. Nondependent Floating-Point ALU Instruction Inserted in the Stall Cycle between a Floating-Point Load and a Dependent Instruction

Pipeline Stage		Cycle Number				
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**
Memory	FA					
	CARB		**lfdu** 1		**lfdu** 2	
	CACC			**lfdu** 1		**lfdu** 2
Dispatch	IQ4	**fsub**				
	IQ3	**lfdu** 2				
	IQ2	**fnmadd**	**fsub**			
	IQ1	**fmul**	**lfdu** 2	**fsub**		
	IQ0	**lfdu** 1	**fnmadd**	**lfdu** 2		
IU	ID	**lfdu** 1	**fnmadd**	**lfdu** 2		
	IE		**lfdu** 1, TAG**fnmadd**	TAG**fsub**	**lfdu** 2, Tag**fsub**	
	IC			**lfdu** 1, TAG**fnmadd**	Tag**fnmadd**	**lfdu** 2, Tag**fsub**
	IWA			**lfdu** 1		**lfdu** 2
	IWL					
	FPSB					
	ISB					

Table I-100. Nondependent Floating-Point ALU Instruction Inserted in the Stall Cycle between a Floating-Point Load and a Dependent Instruction (Continued)

Pipeline Stage		Cycle Number				
FPU	F1			**fnmadd**	**fsub**	**fsub**
	FD		**fmul**	**fmul**	**fnmadd**	**fnmadd**
	FPM			**fmul**	**fmul**	**fnmadd**
	FPA				**fmul**	**fmul**
	FWA					
	FWL				**lfdu** 1	
Notes			**fmul** is allowed to execute because it is not dependent on the load data.		**fsub** stalls because the **fnmadd** is a double-precision operation and takes two cycles in FD. **fnmadd** can start because load data is available.	

Table I-100. Nondependent Floating-Point ALU Instruction Inserted in the Stall Cycle between a Floating-Point Load and a Dependent Instruction (Contd.)

Pipeline Stage		Cycle Number			
Unit	**Stage**	**6**	**7**	**8**	**9**
Memory	FA				
	CARB				
	CACC				
Dispatch	IQ4				
	IQ3				
	IQ2				
	IQ1				
	IQ0				
IU	ID				
	IE				
	IC				
	IWA				
	IWL				
	FPSB				
	ISB				
FPU	F1				
	FD	**fsub**			
	FPM	**fnmadd**	**fsub**		
	FPA	**fnmadd**	**fnmadd**	**fsub**	
	FWA	**fmul**		**fnmadd**	**fsub**
	FWL	**lfdu** 2			
Notes		**fsub** can start this cycle because the load data from **lfdu** 2 is available.			

Note that moving the second **lfdu** ahead of the **fnmadd** does not improve the latency for this four-instruction sequence. The first **lfdu** and the **fnmadd** are dispatched on the same cycle as in example 2 and example 4A; therefore, the number of stall cycles is the same.

Example 4A:

```
lfdu   fr5,0(r3)
lfdu   fr7, x'(r9)'  No improvement in latency.
fnmadd fr4, fr5, fr6, fr2
fsub   fr10, fr11, fr7
```

The instruction timing for this example is the same as that shown in Table I-98. Note that moving the load instructions ahead of the floating-point arithmetic instruction improves latency on longer instruction sequences when the floating-point pipeline is doing double-precision operations as shown in Example 4B and 4C. Example 4C benefits from having the load instructions executed ahead of the **fmadd** instructions. The first two **fmadd** instructions can be dispatched to the FPU, but then floating-point dispatch stalls for a few cycles. In the code sequence shown in Example 4B, the dispatch of the last **lfdu** instruction must wait on the dispatch of the third **fmadd** instruction. Timing for this example is shown in Table I-101.

Example 4B:

```
lfdu   fr1, 0(r5)
fmadd  fr10, fr20, fr1, fr10
lfdu   fr2, 0(r5)
fmadd  fr11, fr20, fr2, fr11
lfdu   fr3, 0(r5)
fmadd  fr12, fr20, fr2, fr12
lfdu   fr4, 0(r5)
fmadd  fr13, fr20, fr2, fr13
```

Table I-101. Mixture of Floating-Point Loads and Floating-Point Accumulate Instructions

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Memory	FA							
	CARB		**lfdu** 1	**lfdu** 2			**lfdu** 3	
	CACC			**lfdu** 1	**lfdu** 2			**lfdu** 3
Dispatch	IQ7	**fmadd** 4						
	IQ6	**lfdu** 4						
	IQ5	**fmadd** 3	**fmadd** 4					
	IQ4	**lfdu** 3	**lfdu** 4					
	IQ3	**fmadd** 2	**fmadd** 3	**fmadd** 4				
	IQ2	**lfdu** 2	**lfdu** 3	**lfdu** 4	**fmadd** 4	**fmadd** 4	**fmadd** 4	
	IQ1	**fmadd** 1	**fmadd** 2	**fmadd** 3	**lfdu** 4	**lfdu** 4	**lfdu** 4	**fmadd** 4
	IQ0	**lfdu** 1	**lfdu** 2	**lfdu** 3	**fmadd** 3	**fmadd** 3	**fmadd** 3	**lfdu** 4
IU	ID	**lfdu** 1	**lfdu** 2	**lfdu** 3	**fmadd** 3	**fmadd** 3	**fmadd** 3	**lfdu** 4
	IE		**lfdu** 1, Tag**fmadd** 1	**lfdu** 2, Tag**fmadd** 2	**lfdu** 3	**lfdu** 3	**lfdu** 3	Tag**fmadd** 3
	IC			**lfdu** 1, Tag**fmadd** 1	**lfdu** 2, Tag**fmadd** 2			**lfdu** 3
	IWA			**lfdu** 1	**lfdu** 2			**lfdu** 3
	IWL							
	FPSB							
	ISB							
FPU	F1			**fmadd** 2	**fmadd** 2	**fmadd** 2		**fmadd** 3
	FD		**fmadd** 1	**fmadd** 1	**fmadd** 1	**fmadd** 1	**fmadd** 2	**fmadd** 2
	FPM					**fmadd** 1	**fmadd** 1	**fmadd** 2
	FPA						**fmadd** 1	**fmadd** 1
	FWA							
	FWL				**lfdu** 1	**lfdu** 2		
Notes			**fmadd** 1 stalled waiting for **lfdu** 1 data.	**fmadd** 2 stalled waiting for **fmadd** 1 to leave FD.	**lfdu** 3 stalls due to floating op in F1 that is ahead of the **lfdu** 3 in program.		FD is open so **lfdu** 3 can execute and **fmadd** 3 can dispatch.	

Table I-101. Mixture of Floating-Point Loads and Floating-Point Accumulate Instructions (Continued)

Pipeline Stage		Cycle Number						
Unit	**Stage**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
Memory	FA							
	CARB	**lfdu** 4						
	CACC		**lfdu** 4					
Dispatch	IQ7							
	IQ6							
	IQ5							
	IQ4							
	IQ3							
	IQ2							
	IQ1							
	IQ0	**fmadd** 4						
IU	ID	**fmadd** 4						
	IE	**lfdu** 4	Tag**fmadd** 4					
	IC	Tag**fmadd** 3	**lfdu** 4	Tag**fmadd** 4				
	IWA		**lfdu** 4					
	IWL							
	FPSB							
	ISB							
FPU	F1		**fmadd** 4					
	FD	**fmadd** 3	**fmadd** 3	**fmadd** 4	**fmadd** 4			
	FPM	**fmadd** 2	**fmadd** 3	**fmadd** 3	**fmadd** 4	**fmadd** 4		
	FPA	**fmadd** 2	**fmadd** 2	**fmadd** 3	**fmadd** 3	**fmadd** 4	**fmadd** 4	
	FWA	**fmadd** 1		**fmadd** 2		**fmadd** 3		**fmadd** 4
	FWL	**lfdu** 3		**lfdu** 4				
Notes								

In example 4C, the **lfdu** instructions are rearranged so that they do not have to wait on dispatch of floating-point instructions.

Example 4C:

```
fdu     fr1, 0(r5)
lfdu    fr2, 0(r5)
lfdu    fr3, 0(r5)
```

```
fmadd   fr10, fr20, fr1, fr10
lfdu    fr4, 0(r5)
fmadd   fr11, fr20, fr2, fr11
fmadd   fr12, fr20, fr3, fr12
fmadd   fr13, fr20, fr4, fr13
```

Timing for this code sequence is shown in Table I-101. Note that this did not improve the flow of the instructions through the FPU pipeline (the limiting factor is that it takes two clock cycles to process each double-precision **fmadd**). The advantage in moving the load instructions (or other IU instructions) ahead of the floating-point instructions is that avoids stalls in the IU pipeline caused by the dependency checking logic.

Table I-102. Mixture of Floating-Point Loads and Floating-Point Accumulate Instructions Rearranged to Move the Loads Ahead of the Accumulates in Program Order

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Memory	FA							
	CARB		**lfdu** 1	**lfdu** 2	**lfdu** 3	**lfdu** 4		
	CACC			**lfdu** 1	**lfdu** 2	**lfdu** 3	**lfdu** 4	
Dispatch	IQ7	**fmadd** 4						
	IQ6	**fmadd** 3						
	IQ5	**fmadd** 2	**fmadd** 4					
	IQ4	**lfdu** 4	**fmadd** 3					
	IQ3	**fmadd** 1	**fmadd** 2	**fmadd** 4				
	IQ2	**lfdu** 3	**lfdu** 4	**fmadd** 3	**fmadd** 4			
	IQ1	**lfdu** 2	**lfdu** 3	**lfdu** 4	**fmadd** 3	**fmadd** 4	**fmadd** 4	
	IQ0	**lfdu** 1	**lfdu** 2	**lfdu** 3	**lfdu** 4	**fmadd** 3	**fmadd** 3	**fmadd** 4
IU	ID	**lfdu** 1	**lfdu** 2	**lfdu** 3	**lfdu** 4	**fmadd** 3	**fmadd** 3	**fmadd** 4
	IE		**lfdu** 1, Tag**fmadd** 1	**lfdu** 2, Tag**fmadd** 2	**lfdu** 3	**lfdu** 4		Tag**fmadd** 3
	IC			**lfdu** 1, Tag**fmadd** 1	**lfdu** 2, Tag**fmadd** 2	**lfdu** 3	**lfdu** 4	
	IWA			**lfdu** 1	**lfdu** 2	**lfdu** 3	**lfdu** 4	
	IWL							
	FPSB							
	ISB							
FPU	F1			**fmadd** 2	**fmadd** 2	**fmadd** 2		**fmadd** 3
	FD		**fmadd** 1	**fmadd** 1	**fmadd** 1	**fmadd** 1	**fmadd** 2	**fmadd** 2
	FPM					**fmadd** 1	**fmadd** 1	**fmadd** 2
	FPA						**fmadd** 1	**fmadd** 1
	FWA							
	FWL				**lfdu** 1	**lfdu** 2	**lfdu** 3	**lfdu** 4
Notes			**fmadd** 1 stalls waiting for **lfdu** 1 data.	**fmadd** 2 stalls waiting for **fmadd** 1 to leave FD.			Since FD is empty **fmadd** 3 can dispatch.	

Table I-102. Mixture of Floating-Point Loads and Floating-Point Accumulate Instructions Rearranged to Move the Loads Ahead of the Accumulates in Program Order (Continued)

Pipeline Stage		Cycle Number						
Unit	**Stage**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
Memory	FA							
	CARB							
	CACC							
Dispatch	IQ7							
	IQ6							
	IQ5							
	IQ4							
	IQ3							
	IQ2							
	IQ1							
	IQ0	**fmadd** 4						
IU	ID	**fmadd** 4						
	IE		Tag**fmadd** 4					
	IC	Tag**fmadd** 3		Tag**fmadd** 4				
	IWA							
	IWL							
	FPSB							
	ISB							
FPU	F1		**fmadd** 4					
	FD	**fmadd** 3	**fmadd** 3	**fmadd** 4	**fmadd** 4			
	FPM	**fmadd** 2	**fmadd** 3	**fmadd** 3	**fmadd** 4	**fmadd** 4		
	FPA	**fmadd** 2	**fmadd** 2	**fmadd** 3	**fmadd** 3	**fmadd** 4	**fmadd** 4	
	FWA	**fmadd** 1		**fmadd** 2		**fmadd** 3		**fmadd** 4
	FWL							
Notes		FD is empty so **fmadd** 4 can dispatch.						

Floating-point instructions that update the CR are interlocked with the IU pipeline to guarantee that the CR is written back in program order. The FPU updates the CR from the FWA stage. The IU updates the CR from the IE stage (except for **stwcx**. which updates the CR from IWA).

Example 5 shows that the FPU may have to wait on the IU and vice versa in order to keep the update of the CR in program order when the floating-point instruction has the RC bit set.

Example 5:

```
mullw. r3, r4, r5
fmadd. fr1, fr2, fr3, fr4
sraw.  r10, r11, r12
```

Timing for this example is shown in Table I-103.

Table I-103. Stalls Caused by CR Update Synchronization on a Floating-Point Instruction with the RC Bit Set

Pipeline Stage		Cycle Number					
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**
Memory	FA						
	CARB						
	CACC						
Dispatch	IQ2	**sraw.**					
	IQ1	**fmadd.**	**sraw.**	**sraw.**	**sraw.**	**sraw.**	**sraw.**
	IQ0	**mullw.**	Tag**fmadd**	Tag**fmadd**	Tag**fmadd**	Tag**fmadd**	Tag**fmadd**
IU	ID	**mullw.**	Tag**fmadd**	Tag**fmadd**	Tag**fmadd**	Tag**fmadd**	Tag**fmadd**
	IE		**mullw.**	**mullw.**	**mullw.**	**mullw.**	**mullw.**
	IC						
	IWA						
	IWL						
	FPSB						
	ISB						
FPU	F1						
	FD		**fmadd.**	**fmadd.**			
	FPM			**fmadd.**	**fmadd.**		
	FPA				**fmadd.**	**fmadd.**	
	FWA						**fmadd.**
	FWL						
Notes							**fmadd.** stalls until the **mullw.** has written the CR, and the tag has moved into IC.

Table I-103. Stalls Caused by CR Update Synchronization on a Floating-Point Instruction with the RC Bit Set (Continued)

Pipeline Stage		Cycle Number				
Unit	**Stage**	**7**	**8**	**9**	**10**	**11**
Memory	FA					
	CARB					
	CACC					
Dispatch	IQ2					
	IQ1					
	IQ0	**sraw.**				
IU	ID	**sraw.**				
	IE	Tag**fmadd**	**sraw.**	**sraw.**	**sraw.**	
	IC	**mullw.**	Tag**fmadd.**			**sraw.**
	IWA	**mullw.**				**sraw.**
	IWL					
	FPSB					
	ISB					
FPU	F1					
	FD					
	FPM					
	FPA					
	FWA	**fmadd.**	**fmadd.**			
	FWL					
Notes		**fmadd.** is stalled until the **mullw.** has written the CR, and the tag has moved into IC.	**fmadd.** is updating the CR this cycle. The **sraw.** is stalled because the FPU is writing the CR.	The **sraw.** is held for one more cycle. It takes an extra cycle to start IE after the FPU has updated the CR.	**sraw.** is writing the CR on this cycle.	

Another stall mechanism is contained completely within the FPU. A floating-point instruction may stall in the FD stage if it needs the result of a preceding floating-point instruction that is in the FPM, the FPA, or the FWA stages. For this reason it is desirable to separate dependent operations as much as possible, by up to three instructions.

Example 6 shows one FPU instruction stalled in the FD stage:

Example 6:

```
fmr     fr3, fr4
```

```
fadd    fr5, fr6, fr3
```

Timing for this example is shown in Table I-103.

Table I-103. Stall between Dependent Floating-Point Arithmetic Instructions

Pipeline Stage		Cycle Number								
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Memory	FA									
	CARB									
	CACC									
Dispatch	IQ1	**fadd**								
	IQ0	**fmr**	**fadd**							
IU	ID									
	IE		Tag**fmr**	Tag**fadd**						
	IC			Tag**fmr**	Tag**fadd**					
	IWA									
	IWL									
	FPSB									
	ISB									
FPU	F1									
	FD		**fmr**	**fadd**	**fadd**	**fadd**	**fadd**			
	FPM			**fmr**				**fadd**		
	FPA				**fmr**				**fadd**	
	FWA					**fmr**				**fadd**
	FWL									
Notes				**fadd** stalled waiting for source register from **fmr**.						

Example 7 shows that nondependent instructions can be moved in between the dependent instructions to keep the FPU busy during the stall cycles in example 6. Example 7 shows insertion of two nondependent instructions (it could have been one, two, or three instructions).

Example 7:

```
fmr     fr3, fr4
fadd    fr10, fr11, fr12
fadd    fr20, fr21, fr22
fadd    fr5, fr6, fr3
```

Timing for this example is shown in Table I-104.

Table I-104. Inserting Nondependent Instructions in between Dependent Floating-Point Arithmetic Instructions

Pipeline Stage		Cycle Number								
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Memory	FA									
	CARB									
	CACC									
Dispatch	IQ3	**fadd** 3								
	IQ2	**fadd** 2	**fadd** 3							
	IQ1	**fadd** 1	**fadd** 2	**fadd** 3						
	IQ0	**fmr**	**fadd** 1	**fadd** 2	**fadd** 3					
IU	ID									
	IE		Tag**fmr**	Tag**fadd** 1	Tag**fadd** 2	Tag**fadd** 3				
	IC			Tag**fmr**	Tag**fadd** 1	Tag**fadd** 2	Tag**fadd** 3			
	IWA									
	IWL									
	FPSB									
	ISB									
FPU	F1									
	FD		**fmr**	**fadd** 1	**fadd** 2	**fadd** 3	**fadd** 3			
	FPM			**fmr**	**fadd** 1	**fadd** 2		**fadd** 3		
	FPA				**fmr**	**fadd** 1	**fadd** 2		**fadd**	
	FWA					**fmr**	**fadd** 1	**fadd** 2		**fadd**
	FWL									
Notes				**fadd** 1 and **fadd** 2 do not stall because it does not depend on the result of the **fmr**.		**fadd** 3 stalled waiting for source register from **fmr**.				

The FPU can process stores in a special way to avoid having to have the store operation wait for a preceding ALU instruction to generate data to be stored. If the floating-point store is in FD stage and the ALU instruction that will generate that instruction is in FPM stage, then the store is folded onto the FPM. The store data is sent to the FPSB when the preceding ALU instruction is in FWA.

Example 8 shows a code sequence for which the FPU can fold out the store.

Example 8:

```
fadd    fr3, fr4, fr5
stfd    fr3, x'0000'(r5)
```

Timing for this example is shown in Table I-105.

Table I-105. Folding of a Floating-Point Store on to a Floating-Point Arithmetic Instruction

Pipeline Stage		Cycle Number					
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**
Memory	FA						
	CARB					**stfd**	
	CACC						**stfd**
Dispatch	IQ1	**stfd**					
	IQ0	**fadd**	**stfd**				
IU	ID		**stfd**				
	IE		Tag**fadd**	**stfd**			
	IC			Tag**fadd**	**stfd**		
	IWA						
	IWL						
	FPSB				**stfd**	**stfd**	
	ISB						
FPU	F1						
	FD		**fadd**	**stfd**			
	FPM			**fadd**			
	FPA				**fadd**, Tag**stfd**		
	FWA					**fadd**, Tag**stfd**	
	FWL						
Notes							

Example 9 shows a similar sequence where the store instruction must wait in FD for the preceding ALU instruction to complete.

Example 9:

```
fadd    fr3, fr4, fr5
fmr     fr10, fr11, fr12
stfd    fr3, x'0000'(r5)
```

Timing for this example is shown in Table I-106.

Table I-106. Folding of a Floating-Point Store onto a Floating-Point Arithmetic Instruction

Pipeline Stage		Cycle Number									
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**19**
Memory	FA										
	CARB									**stfd**	
	CACC										**stfd**
Dispatch	IQ2	**stfd**									
	IQ1	**fmr**	**stfd**								
	IQ0	**fadd**	**fmr**	**stfd**							
IU	ID			**stfd**							
	IE		Tag**fadd**	Tag**fmr**	**stfd**						
	IC			Tag**fadd**	Tag**fmr**	**stfd**					
	IWA										
	IWL										
	FPSB					**stfd**	**stfd**	**stfd**	**stfd**	**stfd**	
	ISB										
FPU	F1										
	FD		**fadd**	**fmr**	**stfd**	**stfd**	**stfd**				
	FPM			**fadd**	**fmr**			**stfd**			
	FPA				**fadd**	**fmr**			**stfd**		
	FWA					**fadd**	**fmr**			**stfd**	
	FWL										
Notes					The **stfd** is held in FD because it is trying to read the target register of the **fadd.**		The **stfd** is held in the FPSB waiting for the store data from the FPU.				

The 601 does not support register renaming for either the IU or the FPU. This has certain consequences on the code generation. For code sequences without true data dependencies, code can be written using different registers for adjacent instructions.

Examples 10 and 11 show two code streams—example 10 uses fewer FPRs, but has poorer performance because there is no hardware renaming.

Example 10:

```
lfd    fr10, x'0'(r1)
fmadd  fr3, fr2, fr10, fr3
lfd    fr10, x'0'(r2)
fmadd  fr3, fr2, fr10, fr3
```

Timing for this example is shown in Table I-107.

Table I-107. No Hardware Register Renaming Mechanism Example—Code Written Assuming Hardware Renaming (Loads Have the Same Target Registers)

Pipeline Stage		Cycle Number					
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**
Memory	FA						
	CARB		**lfd** 1				**lfd** 2
	CACC			**lfd** 1			
Dispatch	IQ3	**fmadd** 2					
	IQ2	**lfd** 2					
	IQ1	**fmadd** 1	**fmadd** 2				
	IQ0	**lfd** 1	**lfd** 2				
IU	ID	**lfd** 1	**lfd** 2				
	IE		**lfd** 1, Tag**fmadd** 1	**lfd** 2, Tag**fmadd** 2	**lfd** 2, Tag**fmadd** 2	**lfd** 2, Tag**fmadd** 2	**lfd** 2, Tag**fmadd** 2
	IC			**lfd** 1, Tag**fmadd** 1			
	IWA						
	IWL						
	FPSB						
	ISB						
FPU	F1			**fmadd** 2	**fmadd** 2	**fmadd** 2	
	FD		**fmadd** 1	**fmadd** 1	**fmadd** 1	**fmadd** 1	**fmadd** 2
	FPM					**fmadd** 1	**fmadd** 1
	FPA						**fmadd** 1
	FWA						
	FWL				**lfd** 1		
Notes			**fmadd** 1 is stalled waiting on data from the **lfd** 1.	**lfd** 2 is stalled waiting on **fmadd** 1 to leave FD. **fmadd** 2 stalls until FD is advancing.	**fmadd** 1 can start because the source data is available from **lfd** 1. The **fmadd** 2 is stalled because the **fmadd** 1 is double-precision.	**fmadd** 1 will not be in FD next cycle, so the instruction in F1 can advance.	**lfd** 2 can arbitrate for the cache because the **fmadd** 1 has left the FD stage. **fmadd** 2 is stalled until data is available from **lfd** 2.

Table I-107. No Hardware Register Renaming Mechanism Example—Code Written Assuming Hardware Renaming (Loads Have the Same Target Registers) (Continued)

Pipeline Stage		Cycle Number					
Unit	**Stage**	**7**	**8**	**9**	**10**	**11**	**12**
Memory	FA						
	CARB						
	CACC	**lfd** 2					
Dispatch	IQ3						
	IQ2						
	IQ1						
	IQ0						
IU	ID						
	IE						
	IC	**lfd** 2, Tag**fmadd** 2					
	IWA						
	IWL						
	FPSB						
	ISB						
FPU	F1						
	FD	**fmadd** 2	**fmadd** 2	**fmadd** 2			
	FPM			**fmadd** 2	**fmadd** 2		
	FPA	**fmadd** 1			**fmadd** 2	**fmadd** 2	
	FWA		**fmadd** 1				**fmadd** 2
	FWL		**lfd** 2				
Notes			Data available from **lfd** 2 so **fmadd** 2 can start in FD. The **fmadd** is a double-precision operation, so it takes 2 cycles in each stage.				

Example 11 uses more FPRs, but has better performance because the second load does not have to wait on the first **fmadd** instruction.

Example 11:

```
lfd    fr10, x'0'(r1)
fmadd  fr3, fr2, fr10, fr3
lfd    fr11, x'0'(r2)
fmadd  fr3, fr2, fr11, fr3
```

Timing for this instruction is shown in Table I-108.

Table I-108. No Hardware Register Renaming Mechanism Example—Code Written Assuming No Hardware Register Renaming (The Loads Have Different Target Registers)

Pipeline Stage		Cycle Number					
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**
Memory	FA						
	CARB		**lfd** 1	**lfd** 2			
	CACC			**lfd** 1	**lfd** 2		
Dispatch	IQ3	**fmadd** 2					
	IQ2	**lfd** 2					
	IQ1	**fmadd** 1	**fmadd** 2				
	IQ0	**lfd** 1	**lfd** 2				
IU	ID	**lfd** 1	**lfd** 2				
	IE		**lfd** 1, Tag**fmadd** 1	**lfd** 2, Tag**fmadd** 2			
	IC			**lfd** 1, Tag**fmadd** 1	**lfd** 2, Tag**fmadd** 2		
	IWA						
	IWL						
	FPSB						
	ISB						
FPU	F1			**fmadd** 2	**fmadd** 2	**fmadd** 2	
	FD		**fmadd** 1	**fmadd** 1	**fmadd** 1	**fmadd** 1	**fmadd** 2
	FPM					**fmadd** 1	**fmadd** 1
	FPA						**fmadd** 1
	FWA						
	FWL				**lfd** 1	**lfd** 2	
Notes			**fmadd** 1 is stalled waiting on data from the **lfd** 1.	**fmadd** 2 is stalled until FD is advancing. The **lfd** 2 uses a different target register than the registers used by **fmadd** 1, so the **lfd** 2 is not held.	**fmadd** 1 can start because the source data is available from **lfd** 1. The **fmadd** 2 is stalled because the **fmadd** 1 is double-precision.	**fmadd** 1 will not be in FD next cycle, so the instruction in F1 can advance. **lfd** 2 data is written to the FPR.	

Table I-108. No Hardware Register Renaming Mechanism Example—Code Written Assuming No Hardware Register Renaming (The Loads Have Different Target Registers) (Continued)

Pipeline Stage		Cycle Number			
Unit	**Stage**	**7**	**8**	**9**	**10**
Memory	FA				
	CARB				
	CACC				
Dispatch	IQ3				
	IQ2				
	IQ1				
	IQ0				
IU	ID				
	IE				
	IC				
	IWA				
	IWL				
	FPSB				
	ISB				
FPU	F1				
	FD	**fmadd** 2			
	FPM	**fmadd** 2	**fmadd** 2		
	FPA	**fmadd** 1	**fmadd** 2	**fmadd** 2	
	FWA		**fmadd** 1		**fmadd** 2
	FWL				
Notes					

Example 12 shows a stream of integer loads and stores feeding into the IU. They will execute one per cycle as long as they hit in the cache, have their translation in the TLB, and the load targets are requested at least two cycles ahead of when they are used.

Example 12:

```
lwu     r1, x'4'(r10)
lwu     r2, x'4'(r10)
lwu     r3, x'4'(r10)
lwu     r4, x'4'(r10)
stwu    r1, x'4'(r12)
stwu    r2, x'4'(r12)
```

```
stwu    r3, x'4'(r12)
stwu    r4, x'4'(r12)
```

Timing for this example is shown in Table I-109. There are no stall cycles in this sequence of code. This sequence might be useful in a loop that is copying data from one location to another. .

Table I-109. Sequence of Integer Load and Store Instructions

Pipeline Stage		Cycle Number									
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Memory	FA										
	CARB		**lwu** 1	**lwu** 2	**lwu** 3	**lwu** 4	**stwu** 1	**stwu** 2	**stwu** 3	**stwu** 4	
	CACC			**lwu** 1	**lwu** 2	**lwu** 3	**lwu** 4	**stwu** 1	**stwu** 2	**stwu** 3	**stwu** 4
Dispatch	IQ7	**stwu** 4									
	IQ6	**stwu** 3	**stwu** 4								
	IQ5	**stwu** 2	**stwu** 3	**stwu** 4							
	IQ4	**stwu** 1	**stwu** 2	**stwu** 3	**stwu** 4						
	IQ3	**lwu** 4	**stwu** 1	**stwu** 2	**stwu** 3	**stwu** 4					
	IQ2	**lwu** 3	**lwu** 4	**stwu** 1	**stwu** 2	**stwu** 3	**stwu** 4				
	IQ1	**lwu** 2	**lwu** 3	**lwu** 4	**stwu** 1	**stwu** 2	**stwu** 3	**stwu** 4			
	IQ0	**lwu** 1	**lwu** 2	**lwu** 3	**lwu** 4	**stwu** 1	**stwu** 2	**stwu** 3	**stwu** 4		
IU	ID	**lwu** 1	**lwu** 2	**lwu** 3	**lwu** 4	**stwu** 1	**stwu** 2	**stwu** 3	**stwu** 4		
	IE		**lwu** 1	**lwu** 2	**lwu** 3	**lwu** 4	**stwu** 1	**stwu** 2	**stwu** 3	**stwu** 4	
	IC			**lwu** 1	**lwu** 2	**lwu** 3	**lwu** 4	**stwu** 1	**stwu** 2	**stwu** 3	**stwu** 4
	IWA			**lwu** 1	**lwu** 2	**lwu** 3	**lwu** 4	**stwu** 1	**stwu** 2	**stwu** 3	**stwu** 4
	IWL				**lwu** 1	**lwu** 2	**lwu** 3	**lwu** 4			
	FPSB										
	ISB			**lwu** 1	**lwu** 2	**lwu** 3	**lwu** 4	**stwu** 1	**stwu** 2	**stwu** 3	**stwu** 4
Notes											

Floating-point load instructions flow through the pipeline in the same way as integer loads with the exception of the write back stage. Floating-point stores, however, cannot be placed consecutively in the instruction stream without incurring some cycles because the FPSB is not handled as efficiently as the ISB stage. Example 13 shows a sequence of floating-point loads and stores. The loads do not stall in the pipeline like the stores do.

Example 13:

```
lfsu   fr1, x'4'(r10)
lfsu   fr2, x'4'(r10)
lfsu   fr3, x'4'(r10)
lfsu   fr4, x'4'(r10)
stfsu  fr1, x'4'(r12)
stfsu  fr2, x'4'(r12)
stfsu  fr3, x'4'(r12)
stfsu  fr4, x'4'(r12)
```

For a series of floating-point store instructions, the maximum throughput is one floating-point store every three processor cycles. Each store holds in FWA for two cycles and this freezes the entire FPU pipeline. Therefore, one cannot schedule floating-point instructions in between the float stores to improve throughput in the floating-point unit. Because the float stores also take up a position in the integer pipeline, there is no opportunity to schedule in integer instructions and improve throughput in the integer pipeline. The timings for example 13 would be the same if the instructions were double-precision loads and stores (assuming proper data alignment). Assuming that example 13 used double-precision instructions, it would take 12 cycles to transfer 4 words of data using the floating-point instructions. However, notice that it will take only 10 cycles to transfer 4 words of data using the integer loads or stores. The conclusion is that it is more efficient to do even large data move operations with integer loads and stores than with floating-point loads or stores.

Table I-110. Sequence of Floating-Point Load and Store Instructions

Pipeline Stage		Cycle Number						
Unit	**Stage**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Memory	FA							
	CARB		**lfsu** 1	**lfsu** 2	**lfsu** 3	**lfsu** 4		
	CACC			**lfsu** 1	**lfsu** 2	**lfsu** 3	**lfsu** 4	
Dispatch	IQ7	**stfsu** 4						
	IQ6	**stfsu** 3	**stfsu** 4					
	IQ5	**stfsu** 2	**stfsu** 3	**stfsu** 4				
	IQ4	**stfsu** 1	**stfsu** 2	**stfsu** 3	**stfsu** 4			
	IQ3	**lfsu** 4	**stfsu** 1	**stfsu** 2	**stfsu** 3	**stfsu** 4		
	IQ2	**lfsu** 3	**lfsu** 4	**stfsu** 1, Tag**stfsu** 1	**stfsu** 2, Tag**stfsu** 2	**stfsu** 3	**stfsu** 4	
	IQ1	**lfsu** 2	**lfsu** 3	**lfsu** 4	**stfsu** 1, Tag**stfsu** 1	**stfsu** 2, Tag**stfsu** 2	**stfsu** 3	**stfsu** 4
	IQ0	**lfsu** 1	**lfsu** 2	**lfsu** 3	**lfsu** 4	**stfsu** 1, Tag**stfsu** 1	**stfsu** 2, Tag**stfsu** 2	**stfsu** 3, Tag**stfsu** 3
IU	ID	**lfsu** 1	**lfsu** 2	**lfsu** 3	**lfsu** 4	**stfsu** 1, Tag**stfsu** 1	**stfsu** 2, Tag**stfsu** 2	**stfsu** 3, Tag**stfsu** 3
	IE		**lfsu** 1	**lfsu** 2	**lfsu** 3	**lfsu** 4	**stfsu** 1, Tag**stfsu** 1	**stfsu** 2, Tag**stfsu** 2
	IC			**lfsu** 1	**lfsu** 2	**lfsu** 3	**lfsu** 4	**stfsu** 1, Tag**stfsu** 1
	IWA			**lfsu** 1	**lfsu** 2	**lfsu** 3	**lfsu** 4	**stfsu** 1
	IWL							
	FPSB							**stfsu** 1
	ISB			**lfsu** 1	**lfsu** 2	**lfsu** 3	**lfsu** 4	
FPU	F1				**stfsu** 2	**stfsu** 2		
	FD			**stfsu** 1	**stfsu** 1	**stfsu** 1	**stfsu** 2	**stfsu** 3
	FPM						**stfsu** 1	**stfsu** 2
	FPA							**stfsu** 1
	FWA							
	FWL				**lfsu** 1	**lfsu** 2	**lfsu** 3	**lfsu** 4
Notes				Store waiting on data from load.				

Table I-110. Sequence of Floating-Point Load and Store Instructions (Continued)

Pipeline Stage		Cycle Number				
Unit	**Stage**	**8**	**9**	**10**	**11**	**12**
Memory	FA					
	CARB	**stfsu** 1			**stfsu** 2	
	CACC		**stfsu** 1			**stfsu** 2
Dispatch	IQ7					
	IQ6					
	IQ5					
	IQ4					
	IQ3					
	IQ2					
	IQ1					
	IQ0					
IU	ID					
	IE	**stfsu** 2, Tag**stfsu** 2	**stfsu** 2, Tag**stfsu** 2	**stfsu** 3, Tag**stfsu** 3	**stfsu** 3, Tag**stfsu** 3	**stfsu** 3, Tag**stfsu** 3
	IC			**stfsu** 2, Tag**stfsu** 2		
	IWA			**stfsu** 2		
	IWL					
	FPSB	**stfsu** 1	**stfsu** 1	**stfsu** 2	**stfsu** 2	**stfsu** 2
	ISB					
FPU	F1					
	FD	**stfsu** 4				
	FPM	**stfsu** 3	**stfsu** 4	**stfsu** 4	**stfsu** 4	
	FPA	**stfsu** 2	**stfsu** 3	**stfsu** 3	**stfsu** 3	**stfsu** 4
	FWA	**stfsu** 1	**stfsu** 2	**stfsu** 2	**stfsu** 2	**stfsu** 3
	FWL					
Notes		FPSB hold for successful access to cache. IE hold until FPSB not held.	FWA held until tag clears IC. FWA hold causes the rest of FPU to hold.	FPSB hold for successful access to cache. IE hold until FPSB not held. FWA held until tag clears IC. FWA hold causes the rest of FPU to hold.	FPSB hold for successful access to cache.IE hold until FPSB not held.	FWA held until tag clears IC. FWA hold causes the rest of FPU to hold.

Table I-110. Sequence of Floating-Point Load and Store Instructions (Continued)

Pipeline Stage		Cycle Number					
Unit	**Stage**	**13**	**14**	**15**	**16**	**17**	**18**
Memory	FA						
	CARB		**stfsu** 3			**stfsu** 4	
	CACC			**stfsu** 3			**stfsu** 4
Dispatch	IQ7						
	IQ6						
	IQ5						
	IQ4						
	IQ3						
	IQ2						
	IQ1						
	IQ0						
IU	ID						
	IE	**stfsu** 4, Tag**stfsu** 4	**stfsu** 4, Tag**stfsu** 4	**stfsu** 4, Tag**stfsu** 4			
	IC	**stfsu** 3, Tag**stfsu** 3			**stfsu** 4, Tag**stfsu** 4		
	IWA	**stfsu** 3			**stfsu** 4		
	IWL						
	FPSB	**stfsu** 3	**stfsu** 3	**stfsu** 3	**stfsu** 4	**stfsu** 4	**stfsu** 4
	ISB						
FPU	F1						
	FD						
	FPM						
	FPA	**stfsu** 4	**stfsu** 4				
	FWA	**stfsu** 3	**stfsu** 3	**stfsu** 4	**stfsu** 4	**stfsu** 4	
	FWL						
Notes		FPSB hold for successful CACC. IE hold until FPSB not held. FWA held until tag clears IC. FWA hold causes the rest of FPU to hold.	FPSB hold for successful access to cache. IE hold until FPSB not held.	FWA hold for tag to clear.	FWA hold for tag to clear. FPSB hold until successful CACC.		

This document contains information on a new product under development by Motorola and IBM. Motorola and IBM reserve the right to change or discontinue this product without notice. Information in this document is provided solely to enable system and software implementers to use PowerPC microprocessors. There are no express or implied copyright or patent licenses granted hereunder by Motorola or IBM to design, modify the design of, or fabricate circuits based on the information in this document.

The PowerPC 601 microprocessor embodies the intellectual property of Motorola and of IBM. However, neither Motorola nor IBM assumes any responsibility or liability as to any aspects of the performance, operation, or other attributes of the microprocessor as marketed by the other party or by any third party. Neither Motorola nor IBM is to be considered an agent or representative of the other, and neither has assumed, created, or granted hereby any right or authority to the other, or to any third party, to assume or create any express or implied obligations on its behalf. Information such as data sheets, as well as sales terms and conditions such as prices, schedules, and support, for the product may vary as between parties selling the product. Accordingly, customers wishing to learn more information about the products as marketed by a given party should contact that party.

Both Motorola and IBM reserve the right to modify this manual and/or any of the products as described herein without further notice. **NOTHING IN THIS MANUAL, NOR IN ANY OF THE ERRATA SHEETS, DATA SHEETS, AND OTHER SUPPORTING DOCUMENTATION, SHALL BE INTERPRETED AS THE CONVEYANCE BY MOTOROLA OR IBM OF AN EXPRESS WARRANTY OF ANY KIND OR IMPLIED WARRANTY, REPRESENTATION, OR GUARANTEE REGARDING THE MERCHANTABILITY OR FITNESS OF THE PRODUCTS FOR ANY PARTICULAR PURPOSE**. Neither Motorola nor IBM assumes any liability or obligation for damages of any kind arising out of the application or use of these materials. Any warranty or other obligations as to the products described herein shall be undertaken solely by the marketing party to the customer, under a separate sale agreement between the marketing party and the customer. In the absence of such an agreement, no liability is assumed by Motorola, IBM, or the marketing party for any damages, actual or otherwise.

"Typical" parameters can and do vary in different applications. All operating parameters, including "Typicals," must be validated for each customer application by customer's technical experts. Neither Motorola nor IBM convey any license under their respective intellectual property rights nor the rights of others. Neither Motorola nor IBM makes any claim, warranty, or representation, express or implied, that the products described in this manual are designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the product could create a situation where personal injury or death may occur. Should customer purchase or use the products for any such unintended or unauthorized application, customer shall indemnify and hold Motorola and IBM and their respective officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola or IBM was negligent regarding the design or manufacture of the part.

Motorola and  are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

IBM and IBM logo are registered trademarks, and IBM Microelectronics is a trademark of International Business Machines Corp.
The PowerPC name, PowerPC logotype,PowerPC 601, PowerPC 603, PowerPC 603e, PowerPC Architectur, and POWER Architecture are trademarks of International Business Machines Corp. used by Motorola under license from International Business Machines Corp. International Business Machines Corp. is an Equal Opportunity/Affirmative Action Employer.
