Gambit

Overview

Gambit is a superscalar processor with a 52-bit native operating mode. Native mode makes use of a 32-entry register file. Native mode instructions vary in length up to 52-bits. The processor manages branching using a compare instruction and compare result registers.

Programming Model

General Registers

Reg			Usage
R0	z	This register is always zero	
R1	acc	Accumulator	First parameter / return value / loop count
R2	x	'x' index register	Second parameter
R3	y	'y' index register	Third parameter
R4			
R5			
R6			
R7 to 29			
R30	fp	frame pointer	
R31	sp	stack pointer	

Compare Result Registers

Compare result registers are two-bit registers that can hold a value from -1 to 1.

-1 means the first operand was less than the second one. 0 means both operands are equal, +1 means the first operand was greater than the second.

Reg	Usage Convention
C0	Integer Compares / set
C1	Float compares / set
C2 to C7	

Status Register

51 7	6	5	4	3	2	1	0
~	N	V	U	C	~	I	Z

Bit		Meaning
0	Z	result was zero
1	I	interrupt mask
2	~	
3	C	carry out from addition .// subtraction
4	U	user defined use
5	V	overflow
6	N	result was negative
7 to 51		reserved

Memory Addressing

The cpu is word oriented with byte addressable memory. The smallest addressable unit of data is a 13-bit byte. Up to 2^{52} Bytes of data are supported and 2^{52} bytes of code.

Exceptions

All processor exceptions vector using the same vector stored in memory location $FFFFFFFFFFFC. The exceptions are differentiated by the value in the exception cause register.

Cause Code	Which Exception	
0	Unimplemented instruction encountered	
1	Interrupt occurred	
2	Non-maskable interrupt occurred	
3	Processor was reset	
4	the BRK instruction was executed	

Instruction Set Summary

The instruction set includes basic arithmetic, logic, and shift instructions including: add, sub, and, or, eor, asl, rol, lsr, and ror. There are several additional instructions which are alternate mnemonics for other instructions. These include bit, and cmp.

The cycle counts are assuming no wait states are required for either instructions or data and both instructions and data can be found in the cache.

ALU Operations

ADD – Addition

Description:

Add two operand values and place the result in the target register. The first operand must be in a register specified by the Ra_5 field of the instruction. The second operand may be either a register specified by the Rb_5 field of the instruction, or an immediate value.

Formats Supported: RR, RI8, RI22, RI35

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

ADDIS – Add Immediate Shifted

Description:

Add an immediate value to bits 22 to 51 of register Ra_5 and place the result in target register Rt_5. Use this instruction when performing an addition with a full 52-bit constant or when building a value in a register.

Formats Supported: RIS

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

AND – Bitwise ‘And’

Description:

Bitwise ‘And’ two operand values and place the result in the target register. The first operand must be in a register specified by the Ra_5 field of the instruction. The second operand may be either a register specified by the Rb_5 field of the instruction, or an immediate value.

Formats Supported: RR, RI8, RI22, RI35

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

ASL – Arithmetic Shift Left

Description:

Left shift one operand value by a second operand value and place the result in the target register. Zeros are shifted into the least significant bits. The first operand must be in a register specified by the Ra_5 field of the instruction. The second operand may be either a register specified by the Rb_5 field of the instruction, or an immediate value.

Formats Supported: RR, RI8

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

ASR – Arithmetic Shift Right

Description:

Right shift one operand value by a second operand value while preserving the sign bit and place the result in the target register. The sign bit is preserved as the shift takes place. The first operand must be in a register specified by the Ra_5 field of the instruction. The second operand may be either a register specified by the Rb_5 field of the instruction, or an immediate value.

Formats Supported: RR, RI8

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

CMP – Comparison

Description:

Compare two operand values and store the relationship in the target compare result register. The operand values are treated as signed integers. The first operand must be in a register specified by the Ra_5 field of the instruction. The second operand may be either a register specified by the Rb_5 field of the instruction, or an immediate value.

If the first operand is less than the second, a minus one is stored in the target register. If the first operand equals the second a zero is stored to the target register, otherwise a positive one is stored.

Formats Supported: RR, RI8, RI22, RI35

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

CMPU – Unsigned Comparison

Description:

Compare two operand values and store the relationship in the target compare result register. The operand values are treated as unsigned integers. The first operand must be in a register specified by the Ra_5 field of the instruction. The second operand may be either a register specified by the Rb_5 field of the instruction, or an immediate value.

If the first operand is less than the second, a minus one is stored in the target register. If the first operand equals the second a zero is stored to the target register, otherwise a positive one is stored.

Formats Supported: RR, RI8, RI22, RI35

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

CSR – Control and Status Register Operation

Description:

The CSR instruction may be used to read, write or read and write a control or status register. The CSR to access is identified by a nine-bit constant in the instruction. The current value of the CSR will be read into the register specified by the Rt field of the instruction. If Rt is zero the read data will be discarded. The CSR will be updated with the contents of the register specified by the Ra field of the instruction. If Ra is specified as register zero, then no update takes place.

Formats Supported: CSR

Flags Affected: none

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

Examples:

```
CSR r1,r0,#CAUSECD ; read the cause code register into r1
```

EOR – Bitwise Exclusive 'Or'

Description:

Bitwise exclusive 'Or' two operand values and place the result in the target register, updating status flags. The first operand must be in a register specified by the Ra_5 field of the instruction. The second operand may be either a register specified by the Rb_5 field of the instruction, or an immediate value.

Formats Supported: RR, RI8, RI22, RI35

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

LSR – Logical Shift Right

Description:

Right shift one operand value by a second operand value and place the result in the target register. Zeros are shifted into the most significant bits. The first operand must be in a register specified by the Ra_5 field of the instruction. The second operand may be either a register specified by the Rb_5 field of the instruction, or an immediate value.

Formats Supported: RR, RI8

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

MUL – Multiplication

Description:

Multiply two operand values and place the result in the target register. The first operand must be in a register specified by the Ra_5 field of the instruction. The second operand may be either a register specified by the Rb_5 field of the instruction, or an immediate value. Both operands are treated as signed values.

Formats Supported: RR, RI8, RI22, RI35

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

OR – Bitwise 'Or'

Description:

Bitwise 'Or' two operand values and place the result in the target register, updating status flags. The first operand must be in a register specified by the Ra_5 field of the instruction. The second operand may be either a register specified by the Rb_5 field of the instruction, or an immediate value.

Formats Supported: RR, RI8, RI22, RI35

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

PERM – Permute Bytes

Description:

This instruction allows any combination of bytes in a source register to be copied to a target register. There are four two-bit fields which specify which source bytes to copy to the target. The source fields may be either a constant specified in the instruction, or the low order eight bits of register Rb. Field S0 indicates the source byte for target byte position 0. S1 indicates the source byte for target byte position 1. S2 and S3 work similarly for the last two target bytes. There are many interesting possibilities with this instruction. A single source byte could be copied to all target byte positions for instance. Or the order of bytes in a word could be reversed.

Formats Supported: RR, RI8

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

ROL – Rotate Left

Description:

Rotate left one operand value by a second operand value and place the result in the target register, updating status flags. The most significant bits are placed in the least significant bits. The first operand must be in a register specified by the Ra_5 field of the instruction. The second operand may be either a register specified by the Rb_5 field of the instruction, or an immediate value.

Formats Supported: RR, RI8

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

ROR – Rotate Right

Description:

Rotate right one operand value by a second operand value and place the result in the target register, updating status flags. The least significant bits are placed in the most significant bits. The first operand must be in a register specified by the Ra_5 field of the instruction. The second operand may be either a register specified by the Rb_5 field of the instruction, or an immediate value.

Formats Supported: RR, RI8

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

SUB – Subtraction

Description:

Subtract two operand values and place the result in the target register, updating status flags. The first operand must be in a register specified by the Ra_5 field of the instruction. The second operand may be either a register specified by the Rb_5 field of the instruction, or an immediate value. The subtraction instruction has an alternate mnemonic CMP which updates the flags differently when the target register is R0.

Formats Supported: RR, RI8, RI22, RI35

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

Memory Operations

LD – Load Data (52 bits)

Description:

Data is loaded from the memory address which is either the sum of Ra and an immediate value or the sum of Ra and Rb. Both register indirect with displacement and indexed addressing are supported.

Formats Supported: RR, RI9, RI23, RI36

Flags Affected: n z

Operation:

Rt = $Memory_{52}$[d+Ra]
or
Rt = $Memory_{52}$[Ra+Rb]

Execution Units: Mem

Clock Cycles: 4 if data is in the cache.

Exceptions: none

LDB – Load Data Byte (13 bits)

Description:

Data is loaded from the memory address which is the sum of Ra and an immediate value. Only register indirect with displacement addressing is supported.

Formats Supported: RI36

Flags Affected: n z

Operation:

$Rt = Memory_{13}[d+Ra]$

Execution Units: Mem

Clock Cycles: 4 if data is in the cache.

Exceptions: none

PHP – Push Processor Status to Stack (52 bits)

Description:

The stack pointer is decremented then the status register is stored to the memory address which is contained in the stack pointer register (R31).

Formats Supported: PHP

Flags Affected: none

Operation:

SP = SP - 4
$Memory_{13}[SP] = SR$

Execution Units: Mem

Clock Cycles: 4 if data is in the cache.

Exceptions: none

PLP – Pull Processor Flags from Stack (52 bits)

Description:

The status register is loaded from the memory address which is contained in the stack pointer register (R31) into a general-purpose register. Then the stack pointer is incremented.

Formats Supported: PLP

Flags Affected: c v n z i u

Operation:

Rt = $Memory_{13}$[SP]
SP = SP + 4

Execution Units: Mem

Clock Cycles: 4 if data is in the cache.

Exceptions: none

POP – Pop Data from Stack (52 bits)

Description:

Data is loaded from the memory address which is contained in the stack pointer register (R31) into a general purpose register. Then the stack pointer is incremented.

Formats Supported: POP

Flags Affected: n z

Operation:

Rt = $Memory_{13}$[SP]
SP = SP + 4

Execution Units: Mem

Clock Cycles: 4 if data is in the cache.

Exceptions: none

PSH – Push Data to Stack (52 bits)

Description:

The stack pointer is decremented then data from a general-purpose register is stored to the memory address which is contained in the stack pointer register (R31).

Formats Supported: PSH

Flags Affected: none

Operation:

SP = SP - 4
$Memory_{13}$[SP] = Rs

Execution Units: Mem

Clock Cycles: 4 if data is in the cache.

Exceptions: none

ST – Store Data (52 bits)

Description:

Data is stored to the memory address which is either the sum of Ra and an immediate value or the sum of Ra and Rb. Both register indirect with displacement and indexed addressing are supported.

Formats Supported: RR, RI9, RI23, RI36

Flags Affected: none

Operation:

$Memory_{52}[d+Ra] = Rs$
or
$Memory_{52}[Ra+Rb] = Rs$

Execution Units: Mem

Clock Cycles: 4 if data is in the cache.

Exceptions: none

STB – Store Data Byte (13 bits)

Description:

Data is stored to the memory address which is the sum of Ra and an immediate value. Only register indirect with displacement addressing is supported.

Formats Supported: RI36

Flags Affected: none

Operation:

$Memory_{13}[d+Ra] = Rs$
or
$Memory_{13}[Ra+Rb] = Rs$

Execution Units: Mem

Clock Cycles: 4 if data is in the cache.

Exceptions: none

Flow Control (Branch Unit) Operations

JMP – Jump

Description:

Transfer execution of instructions to the address specified by the instruction. The target address may be either a 46-bit absolute address or an address contained in a register. For absolute address mode only the low order 46 bits of the program counter are affected. The upper six bits of the program counter remain the same.

Formats Supported: ABS46, R

Flags Affected: none

Operation:

PC = $Address_{46}$
or
PC = Ra

Execution Units: Mem

Clock Cycles: 1

Exceptions: none

JSR – Jump to Subroutine

Description:

Push the address of the next instruction on the stack, then transfer execution of instructions to the address specified by the instruction. The target address may be either a 46-bit absolute address or an address contained in a register. For absolute address mode only the low order 46 bits of the program counter are affected. The upper six bits of the program counter remain the same.

Formats Supported: ABS46, R

Flags Affected: none

Operation:

SP = SP – 4
$Memory_{52}$[SP] = Next PC
PC = $Address_{46}$
or
PC = Ra

Execution Units: Mem

Clock Cycles: 4 if data is in the cache.

Exceptions: none

Notes:

The next PC is either the current PC plus four when absolute addressing is used, or the current PC plus one if register indirect addressing is used.

PFI – Poll for Interrupt

Description:

The PFI instruction tests for the presence of an interrupt and performs the interrupt routine if an interrupt is present. If no interrupt is present a NOP operation is performed and the program continues with the next instruction. PFI does not check for a non-maskable (NMI) interrupt or a reset (RST).

Formats Supported: PFI

Flags Affected: none

Operation:

```
If (IRQ)
        Cause code = 1
        SP = SP – 4
        Memory52[SP] = PC + 1;
        SP = SP – 4
        Memory52[SP] = SR
        PC = Memory52[$FFFFFFFFFFFFC]
Else
        NOP
```

Execution Units: Fetch stage

Clock Cycles:

Exceptions: none

Notes:

RST – Reset Processor

Description:

This instruction initiates the processor reset routine. At power-on a series of RST instructions are forced into the instruction queue. The cause code register is set to the value three. The program counter is then fetched from the last word of memory at address $FFFFFFFFFFFC.

Formats Supported: RST

Operation:

Execution Units: Branch

Clock Cycles:

Exceptions: none

Notes:

RTI – Return from Interrupt Subroutine

Description:

Pop the status register from the stack. Next pop the address to return to from the stack, then transfer execution of instructions to that address.

Formats Supported: RTI

Flags Affected: none

Operation:

SR = $Memory_{52}$[SP]
SP = SP + 4
PC = $Memory_{52}$[SP]
SP = SP + 4

Execution Units: Mem

Clock Cycles: 8 if data is in the cache.

Exceptions: none

Notes:

RTS – Return from Subroutine

Description:

Pop the address of the next instruction from the stack, then transfer execution of instructions to that address.

Formats Supported: RTS

Flags Affected: none

Operation:

PC = $Memory_{52}$[SP]
SP = SP + 4

Execution Units: Mem

Clock Cycles: 4 if data is in the cache.

Exceptions: none

Notes:

WAI – Wait for Interrupt

Description:

The WAI instruction waits for an interrupt to occur by holding the program counter steady. This instruction is similar to the PFI instruction except that it stops and waits for an interrupt whereas PFI doesn't wait. WAI does not check for a non-maskable (NMI) interrupt or a reset (RST).

Formats Supported: WAI

Flags Affected: none

Operation:

```
If (IRQ)
        Cause Code  = 1
        SP = SP – 4
        Memory52[SP] = PC + 1;
        SP = SP – 4
        Memory52[SP] = SR
        PC = Memory52[$FFFFFFFFFFFFC]
Else
        PC = PC
```

Execution Units: Fetch stage

Clock Cycles:

Exceptions: none

Notes:

Instruction Formats

Arithmetic / Logical

ADD							Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11 7	6 0		
		0	~3	Rb5	Ra5	Rt5	04h	ADD Rt,Ra,Rb	2
		1	Imm8		Ra5	Rt5	04h	ADD Rt,Ra,#imm8	2
	Imm22				Ra5	Rt5	14h	ADD Rt,Ra,#imm22	3
Imm35					Ra5	Rt5	24h	ADD Rt,Ra,#imm35	4

SUB							Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11 7	6 0		
		0	~3	Rb5	Ra5	Rt5	05h	Rt,Ra,Rb	2
		1	Imm8		Ra5	Rt5	05h	Rt,Ra,#imm8	2
	Imm22				Ra5	Rt5	15h	Rt,Ra,#imm22	3
Imm35					Ra5	Rt5	25h	Rt,Ra,#imm35	4

CMP								Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11	7	6 0		
		0	~3	Rb5	Ra5	~	Ct3	06h	Rt,Ra,Rb	2
		1	Imm8		Ra5	~	Ct3	06h	Rt,Ra,#imm8	2
	Imm22				Ra5	~	Ct3	16h	Rt,Ra,#imm22	3
Imm35					Ra5	~	Ct3	26h	Rt,Ra,#imm35	4

CMPU								Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11	7	6 0		
		0	~3	Rb5	Ra5	~	Ct3	07h	Rt,Ra,Rb	2
		1	Imm8		Ra5	~	Ct3	07h	Rt,Ra,#imm8	2
	Imm22				Ra5	~	Ct3	17h	Rt,Ra,#imm22	3
Imm35					Ra5	~	Ct3	27h	Rt,Ra,#imm35	4

MUL							Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11 7	6 0		
		0	~$_3$	Rb_5	Ra_5	Rt_5	0Eh	ADD Rt,Ra,Rb	2
		1	Imm_8		Ra_5	Rt_5	0Eh	ADD Rt,Ra,#imm_8	2
	Imm_{22}				Ra_5	Rt_5	1Eh	ADD Rt,Ra,#imm_{22}	3
Imm_{35}					Ra_5	Rt_5	2Eh	ADD Rt,Ra,#imm_{35}	4

AND							Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11 7	6 0		
		0	~$_4$	Rb_5	Ra_5	Rt_5	30o	ADD Rt,Ra,Rb	2
		1	Imm_8		Ra_5	Rt_5	30o	ADD Rt,Ra,#imm_6	2
	Imm_{22}				Ra_5	Rt_5	31o	ADD Rt,Ra,#imm_{14}	3
Imm_{35}					Ra_5	Rt_5	32o	ADD Rt,Ra,#imm_{30}	4

OR							Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11 7	6 0		
		0	~$_4$	Rb_5	Ra_5	Rt_5	40o	ADD Rt,Ra,Rb	2
		1	Imm_8		Ra_5	Rt_5	40o	ADD Rt,Ra,#imm_6	2
	Imm_{22}				Ra_5	Rt_5	41o	ADD Rt,Ra,#imm_{14}	3
Imm_{35}					Ra_5	Rt_5	42o	ADD Rt,Ra,#imm_{30}	4

EOR							Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11 7	6 0		
		0	~$_4$	Rb_5	Ra_5	Rt_5	50o	ADD Rt,Ra,Rb	2
		1	Imm_8		Ra_5	Rt_5	50o	ADD Rt,Ra,#imm_6	2
	Imm_{22}				Ra_5	Rt_5	51o	ADD Rt,Ra,#imm_{14}	3
Imm_{35}					Ra_5	Rt_5	52o	ADD Rt,Ra,#imm_{30}	4

Shifted Immediate

ADDIS					Opcode		Bytes
51 39	38 26	25 22	21 17	16 12	11 7	6 0		
~$_5$	Imm$_{30}$			Ra$_5$	Rt$_5$	23h	Rt,Ra,#imm$_{35}$	4

ANDIS					Opcode		Bytes
51 39	38 26	25 22	21 17	16 12	11 7	6 0		
~$_5$	Imm$_{30}$			Ra$_5$	Rt$_5$	22h	Rt,Ra,#imm$_{35}$	4

ORIS					Opcode		Bytes
51 39	38 26	25 22	21 17	16 12	11 7	6 0		
~$_5$	Imm$_{30}$			Ra$_5$	Rt$_5$	2Ch	Rt,Ra,#imm$_{35}$	4

Shift Operations

ASL							Opcode		Bytes
51 39	38 26	25 22	21 17	16 12	11 7	6 0			
		0	~$_3$	Rb_5	Ra_5	Rt_5	0Ch	Rt,Ra,Rb	2
		1	~$_2$	Imm_6	Ra_5	Rt_5	0Ch	Rt,Ra,#imm_6	2

ROL							Opcode		Bytes
51 39	38 26	25 22	21 17	16 12	11 7	6 0			
		0	~$_3$	Rb_5	Ra_5	Rt_5	1Ch	Rt,Ra,Rb	2
		1	~$_2$	Imm_6	Ra_5	Rt_5	1Ch	Rt,Ra,#imm_6	2

LSR							Opcode		Bytes
51 39	38 26	25 22	21 17	16 12	11 7	6 0			
		0	~$_3$	Rb_5	Ra_5	Rt_5	0Dh	Rt,Ra,Rb	2
		1	~$_2$	Imm_6	Ra_5	Rt_5	0Dh	Rt,Ra,#imm_6	2

ROR							Opcode		Bytes
51 39	38 26	25 22	21 17	16 12	11 7	6 0			
		0	~$_3$	Rb_5	Ra_5	Rt_5	1Dh	Rt,Ra,Rb	2
		1	~$_2$	Imm_6	Ra_5	Rt_5	1Dh	Rt,Ra,#imm_6	2

ASR							Opcode		Bytes
51 39	38 26	25 22	21 17	16 12	11 7	6 0			
		0	~$_3$	Rb_5	Ra_5	Rt_5	2Dh	Rt,Ra,Rb	2
		1	~$_2$	Imm_6	Ra_5	Rt_5	2Dh	Rt,Ra,#imm_6	2

Load and Store Instructions

LD						Opcode		Bytes
51 39	38 26	25 17		16 12	11 7	6 0		
	0	~3	Rb5	Ra5	Rt5	50h	Rt.[Ra+Rb]	2
	1	Disp8		Ra5	Rt5	50h	Rt,d8[Ra]	2
	Disp22			Ra5	Rt5	60h	Rt,d22[Ra]	3
Addr35				Ra5	Rt5	70h	Rt,d35[Ra]	4

LDB						Opcode		Bytes
51 39	38 26	25 17		16 12	11 7	6 0		
	0	~3	Rb5	Ra5	Rt5	51h	Rt.[Ra+Rb]	2
	1	Disp8		Ra5	Rt5	51h	Rt,d8[Ra]	2
	Disp22			Ra5	Rt5	61h	Rt,d22[Ra]	3
Addr35				Ra5	Rt5	71h	Rt,d35[Ra]	4

ST						Opcode		Bytes
51 39	38 26	25 17		16 12	11 7	6 0		
	0	~3	Rb5	Ra5	Rs5	58h	Rs.[Ra+Rb]	2
	1	Disp8		Ra5	Rs5	58h	Rs,d8[Ra]	2
	Disp22			Ra5	Rs5	68h	Rs,d22[Ra]	3
Addr35				Ra5	Rs5	78h	Rs,d35[Ra]	4

STB						Opcode		Bytes
51 39	38 26	25 17		16 12	11 7	6 0		
	0	~3	Rb5	Ra5	Rs5	59h	Rs.[Ra+Rb]	2
	1	Disp8		Ra5	Rs5	59h	Rs,d8[Ra]	2
	Disp22			Ra5	Rs5	69h	Rs,d22[Ra]	3
Addr35				Ra5	Rs5	79h	Rs,d35[Ra]	4

Flow Control

JMP	Flags:				Bytes
	Address$_{45}$	42h	JMP abs45	4
	~ Ra$_5$	48h	JMP [Ra]	1

JSR	Flags:				Bytes
	Address$_{45}$	43h	JSR abs45	4
	~ Ra$_5$	49h	JSR [Ra]	1

{RTGRP}

RTS		~$_4$	0	44h	RTS	1
RTI		~$_4$	1	44h	RTI	1

{WAIGRP}

PFI		Sigmsk$_4$	0	02h	PFI	1
IRQ		~$_4$	1	02h	IRQ	1
WAI		Sigmsk$_4$	2	02h	WAI	1
IRQ		~$_4$	3	02h	IRQ	1

STP		~$_4$	6	24o	STP	1
NOP		~$_4$	7	24o	NOP	1

BEQ		$Disp_{12}$	Cr_3	P_2	0	40h	BEQ disp	2
BNE		$Disp_{12}$	Cr_3	P_2	1	40h	BNE disp	2
BGT		$Disp_{12}$	Cr_3	P_2	2	40h	BGT disp	2
BLT		$Disp_{12}$	Cr_3	P_2	3	40h	BLT disp	2
BGE		$Disp_{12}$	Cr_3	P_2	0	41h	BGE disp	2
BLE		$Disp_{12}$	Cr_3	P_2	1	41h	BLE disp	2
BRA		$Disp_{12}$	Cr_3	P_2	2	41h	BRA disp	2

{BRKGRP}

RST		$\sim_4$	3	00h	RST	1
NMI		$\sim_4$	2	00h	NMI	1
IRQ		$\sim_4$	1	00h	IRQ	1
BRK		$Const_4$	0	00h	BRK	1

Stack push and pop operations.

PUSH		~	Ra_5	5Ah	PUSH Ra	1
POP		~	Rt_5	6Ah	POP Rt	1

String Operations

MVNB		~4	0	45o	MVN	1
MVPB		~4	1	45o	MVP	1
STSB		~4	2	45o	STS	1
CMPSB		~4	3	45o	CMPS	1
MVN		~4	4	45o	MVN	1
MVP		~4	5	45o	MVP	1
STS		~4	6	45o	STS	1
CMPS		~4	7	45o	CMPS	1

Shuffle

PERM								Opcode		Bytes
51 39	38 26	25 22	21 17	16 12	11 7	6 0				
		0	~3	Rb5	Ra5	Rt5	20h	Rt,Ra,Rb	2
		1	S3	S2	S1	S0	Ra5	Rt5	20h	Rt,Ra,#imm8	2

Control and Status Register Access

CSR					Opcode		Bytes
51 39	38 26	25 17	16 12	11 7	6 0		
	1	Regno8	Ra5	Rt5	01h	CSR	2

Opcode Maps

Root Level

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	BRK	CSR	{WAIGRP}		ADD	SUB	CMP	CMPU	AND	OR	EOR		ASL	LSR	MUL	
1x	{string}	{VP}			ADD	SUB	CMP	CMPU	AND	OR	EOR		ROL	ROR	MUL	
2x	PERM		ANDIS	ADDIS	ADD	SUB	CMP	CMPU	AND	OR	EOR		ORIS	ASR	MUL	
3x																
4x	{Branch}	{Branch}	JMP	JSR	{RTGRP}				JMP [R]	JSR [R]			SEQ	SLT	FSLT	FADD
5x	LD	LDB							ST	STB	PUSH		SNE	SLE	FSLE	FSUB
6x	LD	LDB							ST	STB	POP		FSEQ	SLTU		FMUL
7x	LD	LDB							ST	STB			FSNE	SLEU		FDIV

Appendix

The first nine bits of the instruction are used as a nine-bit index into a table which holds pointers to micro-instructions.

Micro-op Instruction Format

Micro-instructions are 24 bits in size.

FL_2	$Opcode_6$	$Cnst_4$	$Src2_4$	$Src1_4$	Tgt_4	Vm_4	Z

FL_2 are two bits holding the first / last micro-instruction indicators. 00 = middle instruction, 01 = first, 10 = last, 11 = first and last.

Micro-op Fields

Tgt_4	Meaning
0	The value 0
1	Get from instruction bits 6 to 10
2	The accumulator register
3	The X register
4	The Y Register
5	The stack pointer
6	the tmp1 register
7	the tmp2 register
8	the SR register
9	the PC register (source)
10	PC + 1
11	PC + 4

$Src1_4$	Meaning
0	The value 0
1	Get from register spec instruction bits 11 to 15
2	The accumulator register
3	The X register
4	The Y Register

5	The stack pointer
6	the tmp1 register
7	the tmp2 register
8	the SR register
9	Get from register spec instruction bits 12 to 14 (Rc_3)

$Src2_4$	Meaning
0	The value 0
1	Get from register spec instruction bits 16 to 25
2	The accumulator register
3	The X register
4	The Y Register
5	The stack pointer
6	the tmp1 register
7	the tmp2 register
8	the SR register
9	the value 1
11	
15	the value -1

Vm_4	Meaning
0	scalar operation
1	vector operation use vector mask register specified in control register
8 to 15	vector operation use specified mask register from micro-op bits 1 to 3

Z	Meaning
0	merge vector results
1	zero masked vector results

$Cnst_4$	Meaning
0	the value 0
1	the value 1
2	the value 2
3	the value 3
4	the value 4
5	bits 9 to 25
6	bits 9 to 12 (branches)
7	bits 6 to 51 (JMP / JSR)
8	bits 16 to 25 of instruction
9	bits 16 to 38 of instruction
10	bits 16 to 51 of instruction
11	bits 6 to 12 (REP / SEP)
13	the value -3
14	the value -2
15	the value -1

Micro-op Lists for Instructions

BRK

```
SUB   SP,SP,#4
ST    PC+1,[SP]
SUB   SP,SP,#4
ST    FLAGS,[SP]
SEP   #i
LD    TMP,$FFFFFFFFFFFFC
JMP   0[TMP]
```

MVN

```
LD    tmp,[X]
ST    tmp,[Y]
ADD   X,X,4
ADD   Y,Y,4
SUB   AC,AC,#1
```

```
        BNE     PC

MVP
        LD      tmp,[X]
        ST      tmp,[Y]
        SUB     X,X,#4
        SUB     Y,Y,#4
        SUB     AC,AC,#1
        BNE     PC
STS
        ST      X,[Y]
        ADD     Y,Y,#4
        SUB     AC,AC,#1
        BNE     PC
CMPS
        LD      tmp1,[X]
        LD      tmp2,[Y]
        ADD     X,X,#4
        ADD     Y,Y,#4
        SUB     ac,ac,#1
        BEQ     PC+1
        CMP     tmp1,tmp2
        BEQ     PC
```

Micro-Instruction Opcodes

Opc_6		Flags Updated
0	ADD	-
1	ADD.	c v n z
2	SUB	-
3	SUB.	c v n z
4	AND.	n z
5	OR.	n z
6	EOR.	n z
7	LD	-
8	LD.	n z
9	LB	-
10	LB.	n z
11	ST	-
12	STB	-
13	ASL.	c n z
14	ROL.	c n z
15	LSR.	c n z
16	ROR.	c n z
17	BRA	-
18	BEQ	-
19	BNE	-
20	BMI	-
21	BPL	-
22	BCS	-
23	BCC	-
24	BVS	-
25	BVC	-
26	SEP	c v n z i u
27	REP	c v n z i u
28	JMP	
29	STP	
30		
31		