Reference Materials

1. RTN Description of SRC

Unified RTN for SRC

Below is the entire RTN description for SRC with reset and exceprion handling integrated into it.

Memory

Processor state

PC 31..0 :	program counter (address of the next instruction)
IR 31..0 :	instruction register
Run:	one bit run/halt indicator
Strt:	start and hard reset signal
Rst:	soft reset signal
R[0..31] 31..0 :	general purpose registers

Processor interrupt mechanism

ireq:	interrupt request signal
iack:	interrupt acknowledge signal
IE:	one bit interrupt enable flag
IPC 31..0 :	storage for PC saved upon interrupt
II 31..0 :	interrupt info.: about source of last interrupt
Isrc_info 15..0 :	information from interrupt source
Isrc_vect 7..0 :	type code from interrupt source

Ivect 31..0 := 20@0#Isrc_vect 7..0 #4@0:

Main memory state

Mem[0..2^{32} - 1] 7..0 :	2^{32} addressable bytes of memory

M[x] 31..0 := Mem[x]#Mem[x+1]#Mem[x+2]#Mem[x+3]:

Formats

Instruction formats

op 4..0 := IR 31..27 :	operation code field
ra 4..0 := IR 26..22 :	target register field
rb 4..0 := IR 21..17 :	operand, address index, or branch target
rc 4..0 := IR 16..12 :	2nd operand, conditional test, or shift count
c1 21..0 := IR 21..0 :	long displacement field
c2 16..0 := IR 16..0 :	short displacement or immediate field
c3 11..0 := IR 11..0 :	count or modifier field

Branch condition format

cond := (c3 2..0 =0 → 0:	never
c3 2..0 =1 → 1:	always
c3 2..0 =2 → R[rc]=0:	if register is zero
c3 2..0 =3 → R[rc]≠0:	if register is nonzero
c3 2..0 =4 → R[rc] 31 =0:	if register is positive or zero
c3 2..0 =5 → R[rc] 31 =1):	if register is negative

Shift count format

n := ((c3 4..0 =0) → R[rc] 4..0 :	shift count is register or
(c3 4..0 ≠0) → c3 4..0):	constant field of instruction

Effective address calculations

```
disp 31..0 := ((rb=0) → c2 16..0 {sign extend}:                              disp.
        (rb≠0) → R[rb] + c2 16..0 {sign extend, 2's complement} ):           addr.
rel 31..0 := PC 31..0 + c1 21..0 {sign extend, 2’s comp.}:                   rel. adr.
```

Instruction interpretation

```
instruction_interpretation :=
    (¬Run  Strt → (Run ← 1: PC, R[0..31] ← 0;              Hard reset
                    instruction_interpretation):
    Run  Rst → ( Rst ← 0: IE ← 0: PC ← 0;                  Soft reset
                    instruction_interpretation):
    Run  ¬Rst  (ireq  IE) → (IPC ← PC 31..0 :              Interrupt
                    II 15..0   ←Isrc_info 15..0  :
                    IE ← 0: PC ← Ivect 31..0 :
                     iack ← 1; iack ← 0;
                    instruction_interpretation):
    Run  ¬Rst  ¬(ireq  IE)→ (IR ← M[PC]:                   Normal fetch
            PC ← PC + 4; instruction_execution):
```

Instruction execution instruction_execution := (

Load and store instructions

ld (:= op= 1) → R[ra] ← M[disp]:	load register
ldr (:= op= 2) → R[ra] ← M[rel]:	load register relative
st (:= op= 3) → M[disp] ← R[ra]:	store register
str (:= op= 4) → M[rel] ← R[ra]:	store register relative
la (:= op= 5) → R[ra] ← disp:	load displacement address
lar (:= op= 6) → R[ra] ← rel:	load relative address

Branch instructions

br (:= op= 8) → (cond → PC ← R[rb]):	cond. branch
brl (:= op= 9) → (R[ra] ← PC: cond → (PC ← R[rb])):	branch & link

Arithmetic instructions (assumed to be 2's complement arithmetic)

add (:= op= 12) → R[ra] ← R[rb] + R[rc]:
addi (:= op= 13) → R[ra] ← R[rb] + $c2_{16..0}$ {2's comp. sign ext.}:
sub (:= op= 14) → R[ra] ← R[rb] - R[rc]:
neg (:= op= 15) → R[ra] ← -R[rc]:
and (:= op= 20) → R[ra] ← R[rb]   R[rc]:
andi (:= op= 21) → R[ra] ← R[rb] $c2_{16..0}$ {sign extend}:
or (:= op= 22) → R[ra] ← R[rb]   R[rc]:
ori (:= op= 23) → R[ra] ← R[rb] $c2_{16..0}$ {sign extend}:
not (:= op= 24) → R[ra] ← ¬R[rc]:

Shift instructions

shr (:= op= 26) → $R[ra]_{31..0}$ ← (n @ 0) # $R[rb]_{31..n}$:	right
shra (:= op= 27) → $R[ra]_{31..0}$ ← (n @ $R[rb]_{31}$) # $R[rb]_{31..n}$:	arith.
shl (:= op= 28) → $R[ra]_{31..0}$ ← $R[rb]_{31-n..0}$ #(n @ 0):	left
shc (:= op= 29) → $R[ra]_{31..0}$ ← $R[rb]_{31-n..0}$ #$R[rb]_{31..32-n}$:	circ.

Interrupt instructions

een (:= op = 10) → (IE ← 1):	exception enable
edi (:= op = 11) → (IE ← 0):	exception disable
rfi (:= op = 30) → (PC ← IPC: IE ← 1):	return from interrupt
svi (:= op = 16) → ($R[ra]_{15..0}$ ←$II_{15..0}$: R[rb] ← $IPC_{31..0}$):	save interrupt state
ri (:= op = 17) → ($II_{15..0}$ ← $R[ra]_{15..0}$: $IPC_{31..0}$ ← R[rb]):	restore interrupt state

Miscellaneous instructions

nop (:= op= 0) → :	No operation
stop (:= op= 31) → Run ← 0	Stop instruction
);	End of instruction_execution

instruction_interpretation.

A.19 RegisterTransfer Notation - RTN

←	Register transfer: register on LHS stores value from RHS.
[ ]	Word index: selects word or range from a named memory.
	Bit index: selects bit or bit range from named register.
n..m	Index range: from left index n to right index m; can be decreasing.
→	If-then: true condition on left yields value and/or action on right.
:=	Definition: text substitution with dummy variables.
#	Concatenation: bits on right appended to bits on left.
:	Parallel separator: actions or evaluations carried out simultaneously.
;	Sequential separator: RHS evaluated and/or performed after LHS.
@	Replication: LHS repetitions of RHS are concatenated.
{ }	Operation modifier: describes type of preceding operation.
( )	Nested grouping of operations or values: operators or separators.
= ≠ < ≤ > ≥	Comparisons: produce 0 or 1 (true or false) logical value.
+ − × ÷	Arithmetic operators: also é ùë ûand mod.
¬ ≡	Logical operators: and, or, not, exclusive or, equivalence.

Notes:

Expressions can be values and/or actions. Actions can be considered side effects if a value is present.

A list of conditional expressions need not have disjoint conditions. Right hand sides of conditionals are evaluated for all conditions which are true. No sequencing is implied unless there are sequential separators between conditional expressions. There is no *else* equivalent.

Pseudo-operations.

.org	Value	Load the program starting at address Value.
.equ	Value	Define the Label symbol to be the constant Value.
.dc	Value [,Value]	Allocate memory words and set to the 32 bit Values.
.dcb	Value [,Value]	Allocate bytes and load them with the 8 bit Values.
.dch	Value [,Value]	Allocate halfwords and load with the 16 bit Values.
.db	Count	Allocate storage for Count bytes.
.dh	Count	Allocate storage fot Count 16 bit halfwords.
.dw	Count	Allocate storage for Count 32 bit words.

Table A.1 SRC assembly language pseudo operations

Values are assumed to be decimal unless terminated by B (binary) or H (hexadecimal).

SRC Instruction Mnemonics1. MC68000 Reference Materials

ld	ra, c2	Load from absolute address. rb is register 0.
ld	ra, c2(rb)	Load from displacement address.
ldr	ra, c1	Load from relative address.
st	ra, c2	Store into absolute address. rb is register 0.
st	ra, c2(rb)	Store into displacement address.
str	ra, c1	Store into relative address.
la	ra, c2	Load value of absolute address into ra. rb is reg. 0.
la	ra, c2(rb)	Load value of displacement address into ra.
lar	ra, c1	Load value of relative address into ra.
add	ra, rb, rc	Add rb to rc and put result in ra.
addi	ra, rb, c2	Add rb to immediate constant and put result in ra.
sub	ra, rb, rc	Subtract rc from rb and put result in ra.
neg	ra, rc	Place two’s complement negative of rc into ra.
or	ra, rb, rc	OR rb and rc and put result in ra.
ori	ra, rb, c2	OR rb and immediate constant and put result in ra.
and	ra, rb, rc	AND rb and rc and put result in ra.
andi	ra, rb, c2	AND rb and immediate constant and put result in ra.
not	ra, rc	Place logical NOT of rc into ra.
shr	ra, rb, c3	Shift rb right into ra by constant shift count c3 ≤ 31.
shr	ra, rb, rc	Shift rb right into ra by count in rc. c3 is 0.
shra	ra, rb, c3	Shift rb right with sign extend into ra by constant c3.
shra	ra, rb, rc	Shift rb right with sign extend into ra by count in rc.
shl	ra, rb, c3	Shift rb left into ra by constant c3.
shl	ra, rb, rc	Shift rb left into ra by count in rc. c3 is 0.
shc	ra, rb, c3	Shift rb left circularly into ra by constant c3.
shc	ra, rb, rc	Shift rb left circularly into ra by count in rc. c3 is 0.
br	rb, rc, c3	Branch to target in rb if c3 satisfies condition c3.
brl	ra,rb,rc,c3	Branch to rb if rc satisfies c3 and save PC in ra.
br	rb	Branch unconditionally to rb.
brl	ra, rb	Branch unconditionally to rb and save PC in ra.
brlnv	ra	Do not branch, but save PC in ra.
brzr	rb, rc	Branch to rb if rc is zero.
brlzr	ra, rb, rc	Branch to rb if rc is zero and save PC in ra.
brnz	rb, rc	Branch to rb if rc is non-zero.
brlnz	ra, rb, rc	Branch to rb if rc is non-zero and save PC in ra.
brpl	rb, rc	Branch to rb if rc is positive or zero (sign is plus).
brlpl	ra, rb, rc	Branch to rb if rc is positive and save PC in ra.

Table A.2 SRC instructions—assembly language form

`brmi`	`rb, rc`	Branch to rb if rc is negative (sign is minus).
`brlmi`	`ra, rb, rc`	Branch to rb if rc is negative and save PC in ra.
`nop`		No operation. Used to insert pipeline bubble.
`stop`		Set Run to zero, halting the machine.
`een`		Exception enable. Set overall exception enable bit.
`edi`		Exception disable. Clear overall exception enable.
`rfi`		Return from interrupt. PC ← IPC; enable exceptions.
`svi`	`ra, rb`	Save II and IPC in ra and rb, respectively.
`ri`	`ra, rb`	Restore II and IPC from ra and rb, respectively.

Table A.2 SRC instructions—assembly language form

5 4 3 2 1 0 ‡

Mode	Reg

Addressing mode Name	**Mode #**	**Reg #**	**Notation***	**Extra Word**	**Operand location**
Data register direct	0	0-7	`Dn`	0	Dn
Address register direct	1	0-7	`An`	0	An
Address register indirect	2	0-7	`(An)`	0	Mem[An]
Autoincrement	3	0-7	`(An)+`	0	Mem[An]; An ← An+WS
Autodecrement	4	0-7	`-(An)`	0	An ← An-WS; Mem[An]
Based	5	0-7	`disp16 (An)`	1	Mem[An + disp16]
Based indexed short	6	0-7	`disp8(An, XnLo)`	1	Mem[An + XnLo + disp8]
Based indexed long	6	0-7	`disp8(An, Xn)`	1	Mem[An + Xn + disp8]
Absolute short	7	0	`addr16`	1	Mem[addr16]
Absolute long	7	1	`addr32`	2	Mem[addr32]
Relative	7	2	`disp16(PC)`	1	Mem[PC + disp16]
Relative indexed short	7	3	`disp8 (PC, XnLo)`	1	Mem[PC + XnLo + disp8]
Relative indexed long	7	3	`disp8(PC, Xn)`	1	Mem[PC + Xn + disp8]
Immediate	7	4	`#data`	1-2	no location, data is value

Motorola MC68000 addressing modes

‡ When the 6-bit field specifies the `dst` (destination) operand of a `MOVE` instruction, the mode and reg fields are reversed.

* An and Dn denote one of the 8 address or data registers respectively
WS = word size in bytes: 1, 2, or 4.
disp8 and disp16 are 8- and 16-bit displacements.
Xn is one of D0-D7, or A0-A7.
XnLo is the low order 16 bits of register Xn, sign extended to 32 bits.
All values smaller than 32 bits are sign extended to 32 bits before addition.
data is an 8-, 16- or 32-bit value as indicated by .B, .W, or .L in the instruction.

Mnemonic	Operands	Opcode Word‡	XNZVC	Operation	Operand Size
`MOVE.B`	`EAs,EAd`	`0001ddddddssssss`	`-xx00`	dst ← src	byte
`MOVE.W`	`EAs,EAd`	`0011ddddddssssss`	`-xx00`	dst ← src	word
`MOVE.L`	`EAs,EAd`	`0010ddddddssssss`	`-xx00`	dst ← src	long
`MOVEA.W`	`EAs,An`	`0011rrr001ssssss`	`-----`	An ← src	word
`MOVEA.L`	`EAs,An`	`0010rrr001ssssss`	`-----`	An ← src	long
`LEA.L`	`EAc,An`	`0100aaa111ssssss`	`-----`	An ← EAc	Addr.
`EXG`	`Dx, Dy`	`1100rrr1mmmmmyyy`	`-----`	Dx ↔ Dy	long

MC68000 Data movement instructions

‡ Notes:
`EAs`: Source EA–any addressing mode, except cannot move byte to address register
`EAd`: Destination EA–any addressing mode except immediate or relative.
`EAc:` Control EA–all modes except register, auto increment, autodecrement, or immediate
`ssssss`, `dddddd` = src and dst addressing mode specifiers. see Table top.
`rrr`, `yyy` = one of eight registers.
`aaa` = one of the eight address registers
`An`, `Dn`, one of the eight address or data registers, respectively.
`mmmmm` = mode field: 01000-exchange data regs.; 01001-exchange address regs.; 10001-exchange data and addr regs., where `xxx` specifies the data reg., and `yyy` specifies the address reg.
Condition codes: `-` = unchgd. from previous value, `x` = chgd. by the operation. 0, 1 = value.

Mnemonic	Operands	Opcode word	XNZVC	Operation	Oprnd size
`ADD`	`EA,Dn`	`1101rrrmmmaaaaaa`	`xxxxx`	dst ← dst+src	b,w,l
`SUB`	`EA,Dn`	`1001rrrmmmaaaaaa`	`xxxxx`	dst ← dst-src	b,w,l
`CMP`	`EA,Dn`	`1011rrrmmmaaaaaa`	`-xxxx`	dst-src	b,w,l
`CMPI`	`#dat,EA`	`00001100wwaaaaaa`	`-xxxx`	dst-immed.data	b,w,l
`MULS`	`EA,Dn`	`1100rrr111aaaaaa`	`-xx00`	Dn ← Dn*src	l ← w*w
`DIVS`	`EA,Dn`	`1000rrr111aaaaaa`	`-xxx0`	Dn ← Dn/src	l ← l/w
`AND`	`EA,Dn`	`1100rrrmmmaaaaaa`	`-xx00`	dst ← dst src	b,w,l
`OR`	`EA,Dn`	`1000rrrmmmaaaaaa`	`-xx00`	dst ← dst src	b,w,l
`EOR`	`EA,Dn`	`1011rrrmmmaaaaaa`	`-xx00`	dst ← dst src	b,w,l
`CLR`	`EAs`	`01000010wwaaaaaa`	`-0100`	dst ← 0	b,w,l
`NEG`	`EAs`	`01000100wwaaaaaa`	`xxxxx`	dst ← 0-dst	b,w,l
`TST`	`EAs`	`01001010wwaaaaaa`	`-xx00`	dst-0	b,w,l
`NOT`	`EA`	`01000110wwaaaaaa`	`-xx00`	dst ← ¬dst	b,w,l

MC68000 Integer arithmetic and logic instructions‡

‡ Notes: `rrr` is a D register number.
`mmm` is a 3-bit mode field specifying the dst as EA or Dn, and operands as b, w, or l:

Byte	**Word**	**Long**	**Destination**
000	001	010	Dn
100	101	110	EA

EA is an effective address.
`aaaaaa` is a 6-bit address specifier. Not all modes are available to all instructions. See the manufacturer’s literature for details.
`ww` is a word-size specifier field: 00-byte; 01-word; 10-long.
`CMPI` is followed by one or two words containing the immediate data to compare.

Mnemonic	**Operands**	**Opcode Word**‡	**Operation**
Conditional instructions			
`Bcc`	`disp`	`0110ccccdddddddd` `DDDDDDDDDDDDDDDD`	**`if`** (cond) **`then`** PC ← PC+ disp
`DBcc`	`Dn,disp`	`0101cccc11001rrr` `DDDDDDDDDDDDDDDD`	**`if`** ¬(cond) **`then`** (Dn ← Dn-1 **`if`** Dn ≠ `−1` **`then`** PC ← PC+ disp) **`else`** PC ← PC+ 2
`Scc`	`EA`	`0101cccc11aaaaaa`	**`if`** (cond) **`then`** (EA) ← FFH **`else`** (EA) ← 00H
Unconditional instructions			
`BRA`	`disp`	`01100000dddddddd` `DDDDDDDDDDDDDDDD`	PC ← PC+ disp
`BSR`	`disp`	`01100001dddddddd` `DDDDDDDDDDDDDDDD`	-(SP) ← PC; PC ← PC+ disp
`JMP`	`EA`	`0100111011aaaaaa`	PC ← EA
`JSR`	`EA`	`0100111010aaaaaa`	-(SP) ← PC; PC ← EA
Subroutine return instructions			
`RTR`		`0100111001110111`	CC ← (SP)+; PC ← (SP)+
`RTS`		`0100111001110101`	PC ← (SP)+

MC68000 Program control instructions

‡ Notes:
`rrr` is one of An
If 8-bit displacement `dddddddd` is zero, then displacement is `DDDDDDDDDDDDDDDD`.
EA is an effective address.
`aaaaaa` is a 6-bit effective address specifier. Not all addressing modes are available to all instructions. See the manufacturer's literature for details.
`ww` is a word-size field: 00-byte; 01-word; 10-long.
`cccc` is defined in the Table below:

Name	**Meaning**	**Code**	**Logic**	**Name**	**Meaning**	**Code**	**Logic**
T	true	0000	`1`	F	false	0001	`0`
CC	carry clear	0100	C	LS	low or same	0011	C+Z
CS	carry set	0101	C	LT	less than	1101	N·V+N·V
EQ	equal	0111	Z	MI	minus	1011	N
GE	greater or equal	1100	N·V+N·V	NE	not equal	0110	Z
GT	greater than	1110	N·V·Z+N·V·Z	PL	plus	1010	N
HI	high	0010	C·Z	VC	overflow clear	1000	V
LE	less or equal	1111	N·V+N·V+Z	VS	overflow set	1001	V

MC68000 Conditions