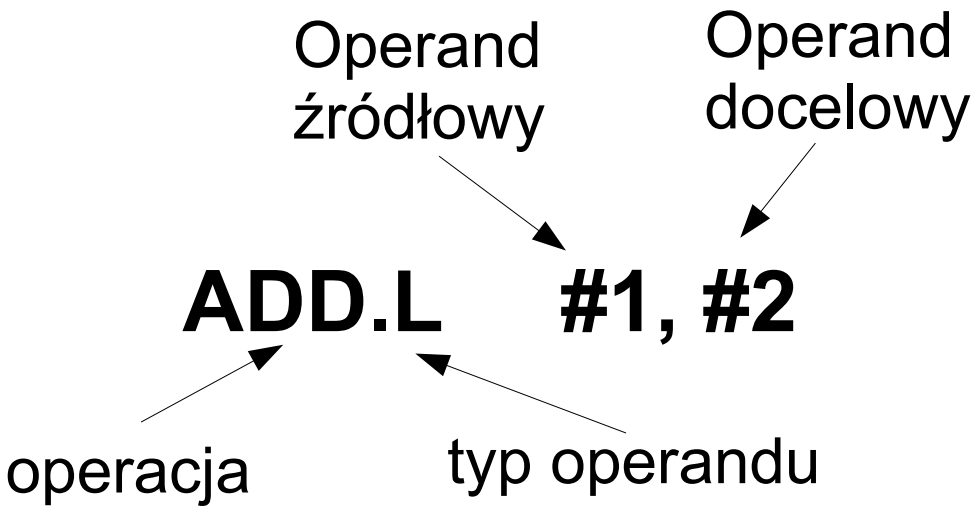
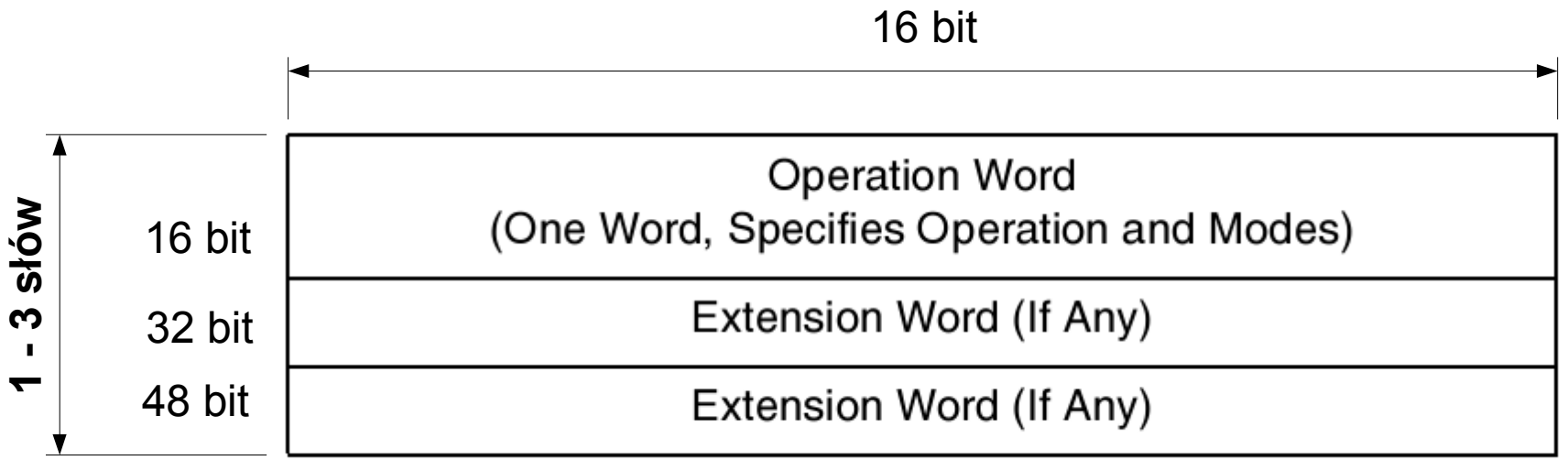
Programowanie mikroprocesora ColdFire - instrukcje asemblera

Format instrukcji procesora Motorola ColdFire (1)





Notacja

Register Specifications	
A*n*	Any address register *n* (example: A3 is address register 3)
Ax, Ay	Destination and source address registers, respectively
D*n*	Any data register *n* (example: D5 is data register 5)
Dx, Dy	Destination and source data registers, respectively
Dw	Data register containing a remainder
Rc	Control register
R*n*	Any address or data register
Rx, Ry	Any destination and source registers, respectively
Xi	Index register, can be any address or data register; all 32-bits are used.
Subfields and Qualifiers	
#<data>	Immediate data following the instruction word(s).
( )	Identifies an indirect address in a register.
d_n	Displacement value, *n* bits wide (example: d_{16} is a 16-bit displacement).
sz	Size of operation: Byte (B), Word (W), Longword (L)
lsb, msb	Least significant bit, most significant bit
LSW, MSW	Least significant word, most significant word
SF	Scale factor for an index register

Format instrukcji procesora Motorola ColdFire (2)

SINGLE EFFECTIVE ADDRESS OPERATION WORD FORMAT

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X	X	X	X	X	X	X	X	EFFECTIVE ADDRESS					
										MODE			REGISTER		

BRIEF EXTENSION WORD FORMAT

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D/A	REGISTER			W/L	SCALE		0	DISPLACEMENT							

FULL EXTENSION WORD FORMAT

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D/A	REGISTER			W/L	SCALE		1	BS	IS	BD SIZE		0	I/IS		
BASE DISPLACEMENT (0, 1, OR 2 WORDS)															
OUTER DISPLACEMENT (0, 1, OR 2 WORDS)															

Field	Definition
Instruction	
Mode	Addressing mode (see Table 2-3)
Register	General register number (see Table 2-3)
Extensions	
D/A	Index register type 0 = D*n* 1 = A*n*
W/L	Word/longword index size 0 = Sign-Extended Word ← **Short** 1 = Long Word
Scale	Scale factor 00 = 1 01 = 2 10 = 4 11 = 8 (supported only if FPU is present)

Przykładowa instrukcja (1)

Operation: Source + Destination → Destination

Assembler Syntax: ADD.L <ea>y,Dx
ADD.L Dy,<ea>x

Attributes: Size = longword

Description: Adds the source operand to the destination operand using binary addition and stores the result in the destination location. The size of the operation may only be specified as a longword. The mode of the instruction indicates which operand is the source and which is the destination as well as the operand size.

The Dx mode is used when the destination is a data register; the destination <ea>x mode is invalid for a data register.

In addition, ADDA is used when the destination is an address register. ADDI and ADDQ are used when the source is immediate data.

Condition Codes:

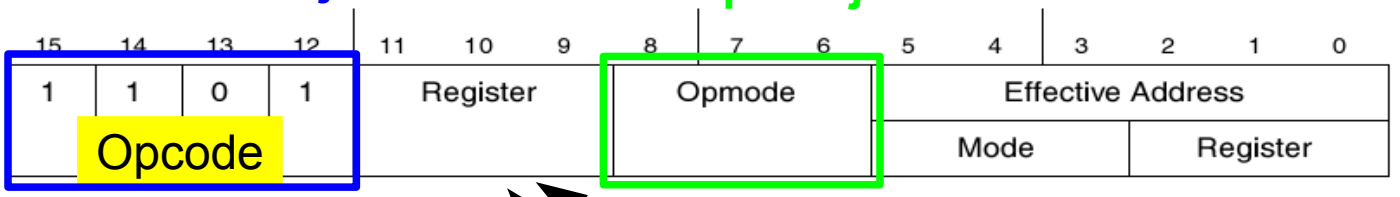
X	N	Z	V	C
*	*	*	*	*

- X Set the same as the carry bit
- N Set if the result is negative; cleared otherwise
- Z Set if the result is zero; cleared otherwise
- V Set if an overflow is generated; cleared otherwise
- C Set if an carry is generated; cleared otherwise

Instruction Format:

kod instrukcji

kod operacji



Instruction Fields:

- Register field—Specifies the data register.
- Opmode field:

Długość instrukcji: 2-3 słów

Byte	Word	Longword	Operation
—	—	010	<ea>y + Dx → Dx
—	—	110	Dy + <ea>x → <ea>x

← kod operacji

Przykładowa instrukcja (2)

Instruction Fields (continued):

- Effective Address field—Determines addressing mode
  - — For the source operand <ea>y, use addressing modes listed in the following table:

Addressing Mode	Mode	Register
Dy	000	reg. number:Dy
Ay	001	reg. number:Ay
(Ay)	010	reg. number:Ay
(Ay) +	011	reg. number:Ay
– (Ay)	100	reg. number:Ay
(d_{16},Ay)	101	reg. number:Ay
(d_8,Ay,Xi)	110	reg. number:Ay

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xi)	111	011

  - — For the destination operand <ea>x, use addressing modes listed in the following table:

Addressing Mode	Mode	Register
Dx	—	—
Ax	—	—
(Ax)	010	reg. number:Ax
(Ax) +	011	reg. number:Ax
– (Ax)	100	reg. number:Ax
(d_{16},Ax)	101	reg. number:Ax
(d_8,Ax,Xi)	110	reg. number:Ax

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xi)	—	—

Kod instrukcji (Opcode)

Bits 15–12	Hex	Operation
0000	0	Bit Manipulation/Immediate
0001	1	Move Byte
0010	2	Move Longword
0011	3	Move Word
0100	4	Miscellaneous
0101	5	ADDQ/SUBQ/Scc/TPF
0110	6	Bcc/BSR/BRA
0111	7	MOVEQ/MVS/MVZ
1000	8	OR/DIV
1001	9	SUB/SUBX
1010	A	MAC/EMAC instructions/MOV3Q
1011	B	CMP/EOR
1100	C	AND/MUL
1101	D	ADD/ADDX
1110	E	Shift
1111	F	Floating-Point/Debug/Cache Instructions

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	1	8-bit displacement							
16-bit displacement if 8-bit displacement = 0x00															
32-bit displacement if 8-bit displacement = 0xFF															

MOVE.sz <ea>y, <ea>x

Operation: Source → Destination

Attributes: Size = byte, word, longword

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

Tryby adresowania argumentu źródłowego <ea>y:

Dy, Ay, (Ay), (Ay)+, -(Ay), (d16,Ay), (d8,Ay,Xi), (xxx).W/L, #<data>, (d16,PC), (d8,PC,Xi)

Tryby adresowania argumentu docelowego <ea>x:

Dx, (Ax), (Ax)+, -(Ax), (d16,Ax), (d8,Ax,Xi), (xxx).W/L, (d16,PC), (d8,PC,Xi)

Ograniczenia:

Source Addressing Mode	Destination Addressing Mode
Dy, Ay, (Ay), (Ay)+,-(Ay)	All possible
(d_{16}, Ay), (d16, PC)	All possible except (d_8, Ax, Xi), (xxx).W, (xxx).L
(d8, Ay, Xi), (d8, PC, Xi), (xxx).W, (xxx).L, #<xxx>	All possible except (d_{16}, Ax), (d_8, Ax, Xi), (xxx).W, (xxx).L

Instrukcje przesyłania danych

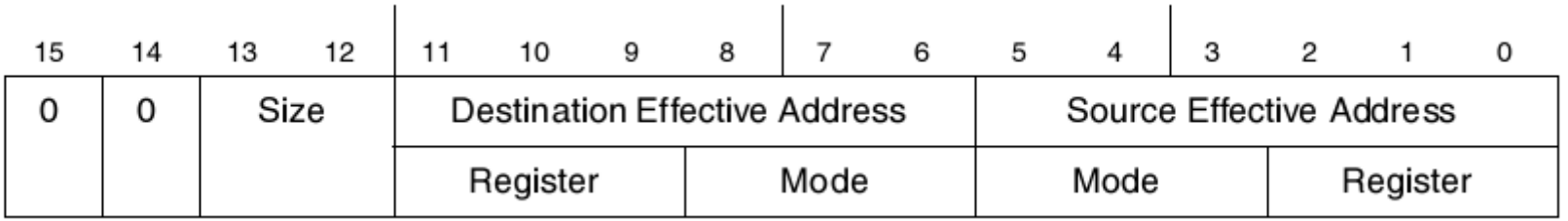
Table 3-2. Data Movement Operation Format

Instruction	Operand Syntax	Operand Size	Operation	First Appeared: ISA, (E)MAC, or FPU
FDMOVE	FPy,FPx	D	Source → Destination; round destination to double	FPU
FMOVE	<ea>y,FPx FPy,<ea>x FPy,FPx FPcr,<ea>x <ea>y,FPcr	B,W,L,S,D B,W,L,S,D D L L	Source → Destination FPcr can be any floating-point control register: FPCR, FPIAR, FPSR	FPU
FMOVEM	#list,<ea>x <ea>y,#list	D	Listed registers → Destination Source → Listed registers	FPU
FSMOVE	<ea>y,FPx	B,W,L,S,D	Source → Destination; round destination to single	FPU
LEA	<ea>y,Ax	L	<ea>y → Ax	ISA_A
LINK	Ay,#<displacement>	W	SP – 4 → SP; Ay → (SP); SP → Ay, SP + d_n → SP	ISA_A
MOV3Q	#<data>,<ea>x	L	Immediate Data → Destination	ISA_B
MOVCLR	ACCy,Rx	L	Accumulator → Destination, 0 → Accumulator	EMAC
MOVE MOVE from CCR MOVE to CCR	<ea>y,<ea>x MACcr,Dx <ea>y,MACcr CCR,Dx <ea>y,CCR	B,W,L L L W W	Source → Destination where MACcr can be any MAC control register: ACCx, ACCext01, ACCext23, MACSR, MASK	ISA_A[1] MAC[2] MAC[2] ISA_A ISA_A
MOVEA	<ea>y,Ax	W,L → L	Source → Destination	ISA_A
MOVEM	#list,<ea>x <ea>y,#list	L	Listed Registers → Destination Source → Listed Registers	ISA_A
MOVEQ	#<data>,Dx	B → L	Immediate Data → Destination	ISA_A
MVS	<ea>y,Dx	B,W	Source with sign extension → Destination	ISA_B
MVZ	<ea>y,Dx	B,W	Source with zero fill → Destination	ISA_B

Adresowanie natychmiastowe

Instruction Format:

MOVE



Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100

Addressing Mode	Mode	Register
Dy	000	reg. number:Dy
Ay	001	reg. number:Ay

68K GAS gp.asm page 1

```
1          .equ   IPSBAR, 0x40000000
2          .equ   PTCPAR, 0x0010005A
3          .equ   PTDPAR, 0x0010005B
4          .equ   DDRTC,  0x00100023
5          .equ   DDRTD,  0x00100024
6          .equ   PORTTC, 0x0010000F
7          .equ   PORTTD, 0x00100010
8
9                 _start:

10 0000    103C 0011          MOVE.B   #0x11,       %d0
11 0004    323C 2211          MOVE.W   #0x2211,     %d1
12 0008    247C 4433  2211    MOVE.L   #0x44332211, %a2
12 000e
13
```

Size field

01 byte operation

11 word operation

10 longword operation

Przykładowa instrukcja (2)

Instruction Fields (continued):

- Effective Address field—Determines addressing mode
  - — For the source operand <ea>y, use addressing modes listed in the following table:

Addressing Mode	Mode	Register
Dy	000	reg. number:Dy
Ay	001	reg. number:Ay
(Ay)	010	reg. number:Ay
(Ay) +	011	reg. number:Ay
– (Ay)	100	reg. number:Ay
(d_{16},Ay)	101	reg. number:Ay
(d_8,Ay,Xi)	110	reg. number:Ay

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xi)	111	011

  - — For the destination operand <ea>x, use addressing modes listed in the following table:

Addressing Mode	Mode	Register
Dx	—	—
Ax	—	—
(Ax)	010	reg. number:Ax
(Ax) +	011	reg. number:Ax
– (Ax)	100	reg. number:Ax
(d_{16},Ax)	101	reg. number:Ax
(d_8,Ax,Xi)	110	reg. number:Ax

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xi)	—	—

Rodzaje instrukcji procesora ColdFire (1)

- Arytmetyczne
- Transferu danych
- Sterujące przepływem programu
- Logiczne oraz arytmetyczne przesunięcia
- Operacje na bitach
- Specjalne
- Diagnostyczne
- Sterujące pamięcią podręczną

Rodzaje instrukcji procesora ColdFire (2)

DATA

CLR	MOVE
EXT	MOVEQ
EXTB	SWAP

SHIFT

ASL	LSR
ASR	LSL

CONTROL

BRA	*RTE
BSR	RTS
JSR	TRAP
JMP	TRAPF
NOP	

ARITHMETIC

ADD	MUL
CMP	NEG
TST	SUB

MULTIPLE PRECISION

ADDX	SUBX
NEGX	

CONDITIONAL

Bcc	Scc

LOGICAL

AND	NOT
EOR	OR

BIT MANIPULATION

BCHG	BSET
BCLR	BTST

SPECIAL INSTRUCTIONS

ILLEGAL	LINK	MOVEM
LEA	UNLK	*MOVEC
PEA	*STOP	MAC

DEBUG

*PULSE
WDDATA
*WDEBUG
! HALT

Rozszerzona lista instrukcji ColdFire V2

Mnemonic	Description	ISA_A	ISA_A+	ISA_B	ISA_C	FPU	MAC	EMAC	EMAC_B
ADD	Add	X	X	X	X				
ADDA	Add Address	X	X	X	X				
ADDI	Add Immediate	X	X	X	X				
ADDQ	Add Quick	X	X	X	X				
ADDX	Add with Extend	X	X	X	X				
AND	Logical AND	X	X	X	X				
ANDI	Logical AND Immediate	X	X	X	X				
ASL, ASR	Arithmetic Shift Left and Right	X	X	X	X				
Bcc.{B,W}	Branch Conditionally, Byte and Word	X	X	X	X				
BITREV	Bit Reverse		X		X				
BRA.{B,W}	Branch Always, Byte and Word	X	X	X	X				
BRA.L	Branch Always, Longword		X	X					
FF1	Find First One		X		X				
MOVE ACC to ACC	Copy Accumulator							X	X
MOVE from ACC	Move from Accumulator						X	X	X
MOVE from ACCext01	Move from Accumulator 0 and 1 Extensions							X	X
MOVE ACCext23	Move from Accumulator 2 and 3 Extensions							X	X

Asemblacja programu (1)

/opt/labtools/cold-asm test.asm

/opt/gnutools/bin/**m68k-elf-as** -Wa -as -m 528x gp.asm -g -o test.o > test.lst

-a[sub-option...] turn on listings, s - include symbols
-mcpu=<cpu> set cpu [default 68020]

Architecture variants are: 68000 ... cpu32 ... 5280 | 5281 | 5282 | **528x** ... 532x ... 5407 ... 548x
-m[no-]float enable/disable architecture extension
-m[no-]div enable/disable ColdFire architecture extension
-m[no-]usp enable/disable ColdFire architecture extension
-m[no-]mac enable/disable ColdFire architecture extension
-m[no-]emac enable/disable ColdFire architecture extension
-g --gen-debug generate debugging information

/opt/gnutools/bin/**m68k-elf-ld** -T /opt/labtools/COBRA5282-ram.ld test.o -o test.elf

-o FILE, --output FILE Set output file name
-T FILE, --script FILE Read linker script

test.asm => test.o, test.lst => test.elf

Asemblacja programu (2)

lab@dmakow:/home/dmakow/motorola/!/asm/link# **file test.elf**
test.elf: ELF 32-bit MSB executable, Motorola version 1 (SYSV), statically linked, not stripped

lab@dmakow:/home/dmakow/motorola/!/asm/link# **file test.lst**
test.lst: ISO-8859 assembler program text

lab@dmakow:/home/dmakow/motorola/!/asm/link# **file test.o**
test.o: ELF 32-bit MSB relocatable, Motorola version 1 (SYSV), not stripped

lab@dmakow:/home/dmakow/motorola/!/asm/link# **file test.asm**
test.asm: ISO-8859 assembler program text

```
68K GAS  test.asm                       page 1
  1          .equ          IPSBAR, 0x40000000
  2          _start:
  3 0000 6000 0002     bra.w MAIN
  4
  5          MAIN:
  6
  7 0004 203C FFFF     MOVE.L  #-1000,%D0
  8 0016 C1C1          MULS.W  %D1, %D0
```

Plik test.lst

Rejestry procesora a GDB

(gdb) info r

d0	0x0	0
d1	0xc	12
d2	0x20	32
d3	0x28ebc06c	686538860
d4	0x0	0
d5	0x85820a45	-2055075259
d6	0xd6d6d6d6	-690563370
d7	0x605b3e56	1616592470
a0	0x40000000	0x40000000
a1	0x5c44	0x5c44
a2	0x6de0c5d3	0x6de0c5d3
a3	0x10000	0x10000
a4	0x167a	0x167a
a5	0x80014cc1	0x80014cc1
fp	0x5b94	0x5b94
sp	0x5b94	0x5b94

ps	0x2704	9988
pc	0xffc2cda4	0xffc2cda4
vbr	0x100	256
cacr	0x0	0
acr0	0x600b0000	1611333632
acr1	0x5d094084	0x5d094084
rambar	0xf0000121	-268435167
mbar	0x1ffff	131071
csr	0x1000000	16777216
aatr	0x5	5
tdr	0x40000000	1073741824
pbr	0xffc04ee0	-4174112
pbmr	0x0	0
abhr	0x0	0
ablr	0x0	0
dbr	0x0	0
dbmr	0x0	0

Instrukcje arytmetyczne (1)

Instruction	Operand Syntax	Operand Size	Operation	ISA, (E)MAC
ADD ADDA	Dy,<ea>x <ea>y,Dx <ea>y,Ax	L L L	Source + Destination → Destination	ISA_A
ADDI ADDQ	#<data>,Dx #<data>,<ea>x	L L	Immediate Data + Destination → Destination	ISA_A
ADDX	Dy,Dx	L	Source + Destination + CCR[X] → Destination	ISA_A
CLR	<ea>x	B, W, L	0 → Destination	ISA_A
CMP CMPA	<ea>y,Dx <ea>y,Ax	B, W, L W, L	Destination – Source → CCR	ISA_A[1]
CMPI	#<data>,Dx	B, W, L	Destination – Immediate Data → CCR	ISA_A[1]
DIVS/DIVU	<ea>y,Dx	W, L	Destination / Source → Destination (Signed or Unsigned)	ISA_A
EXT EXTB	Dx Dx Dx	B → W W → L B → L	Sign-Extended Destination → Destination	ISA_A
MAC	Ry,RxSF,ACCx Ry,RxSF,<ea>y,Rw,ACCx	W, L W, L	ACCx + (Ry * Rx){<<|>>}SF → ACCx ACCx + (Ry * Rx){<<|>>}SF → ACCx; (<ea>y(&MASK)) → Rw	ISA_A
MSAC	Ry,RxSF,ACCx Ry,RxSF,<ea>y,Rw,ACCx	W, L W, L	ACCx - (Ry * Rx){<<|>>}SF → ACCx ACCx - (Ry * Rx){<<|>>}SF → ACCx; (<ea>y(&MASK)) → Rw	ISA_A
MULS/MULU	<ea>y,Dx	W * W → L L * L → L	Source * Destination → Destination (Signed or Unsigned)	ISA_A
NEG	Dx	L	0 – Destination → Destination	ISA_A
NEGX	Dx	L	0 – Destination – CCR[X] → Destination	ISA_A
REMS/REMU	<ea>y,Dw:Dx	L	Destination / Source → Remainder (Signed or Unsigned)	ISA_A
SUB SUBA	<ea>y,Dx Dy,<ea>x <ea>y,Ax	L L L	Destination - Source → Destination	ISA_A
SUBI SUBQ	#<data>,Dx #<data>,<ea>x	L L	Destination – Immediate Data → Destination	ISA_A
SUBX	Dy,Dx	L	Destination – Source – CCR[X] → Destination	ISA_A

ADD

Operation: Source + Destination → Destination

Assembler Syntax: ADD.L <ea>y,Dx
ADD.L Dy,<ea>x

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

Instruction Format:

15	14	13	12	11 10 9	8 7 6	5 4	3 2 1 0
1	1	0	1	Register	Opmode	Effective Address	
						Mode	Register

Tryby adresowania argumentu źródłowego <ea>y:

Dy, Ay, (Ay), (Ay)+, -(Ay), (d16,Ay), (d8,Ay,Xi), (xxx).W/L, #<data>, (d16,PC), (d8,PC,Xi)

Tryby adresowania argumentu docelowego <ea>x:

(Ax), (Ax)+, -(Ax), (d16,Ax), (d8,Ax,Xi), (xxx).W/L

ADDA.L <ea>y, Ax

Operation: Source + Destination → Destination

Assembler Syntax: ADDA.L <ea>y,Ax

Condition Codes: Not affected

Tryby adresowania argumentu źródłowego <ea>y:

Dy, Ay, (Ay), (Ay)+, -(Ay), (d16,Ay), (d8,Ay,Xi), (xxx).W/L, #<data>, (d16,PC), (d8,PC,Xi)

```
ADD.L   %D0, %D1
ADD.L   %D0, %D0
ADD.L   %D0, %A0

ADD.L  %A0, %D0

ADDA.L  %D0, %A0
ADDA.L  %A0, %A0

ADDA.L  %A0, %D0    Error: operands mismatch -- statement `adda.l %A0,%D0' ignored
```

```
1 001a      D3F0 1CEC           ADDA.L      -20(%A0, %D1*4), %A1
2 001e      D3FC 0000  0003     ADDA.L      #0x03, %A1
```

ADDI.L #<data>, Dx

Operation: Immediate Data + Destination → Destination

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	0	0	Register, Dx		
Upper Word of Immediate Data															
Lower Word of Immediate Data															

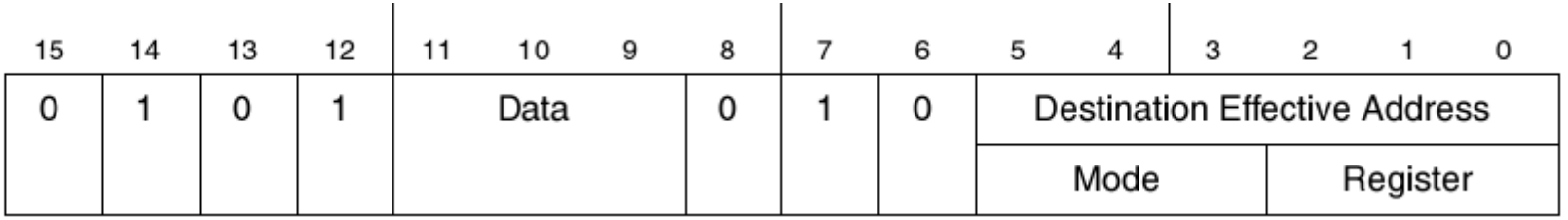
1 001a	0681 0000 0009	ADDI.L #0x09, %D1
2 0020	0681 0000 0003	ADDI.L #0x03, %D1
3	????	ADDI.L #1, %A0

Error: operands mismatch -- statement `addi.l #1,%A0' ignored

ADDQ.L #<data>, <ea>x

Operation: Immediate Data + Destination → Destination

Instruction Format:



1 - 8

Tryby adresowania argumentu docelowego <ea>x:

Dx, Ax, (Ax), (Ax)+, -(Ax), (d16,Ax), (d8,Ax,Xi), (xxx).W/L

```
1 001a      0681 0000 0001      ADDI.L  #0x01, %D1
2 0020      0681 FFFF FFFF      ADDI.L  #0xFFFFFFFF, %D1
3 0026      5682                ADDQ.L #0x3, %D2

4 001a      5681                ADD.L    #0x03, %D1  =>  ADDQ
5 001a      0681 0000 0009      ADD.L    #0x09, %D1  =>  ADDI
6 001c      D2B8 0003           ADD.L     0x03, %D1  =>  ADD
```

```
ADDQ.L    #0x9, %D3    gp.asm:30: Error: operands mismatch --
                       statement `addq.l #0x9,%D3' ignored

ADDQ.L    #0x0, %D2    gp.asm:31: Error: operands mismatch --
                       statement `addq.l #0x0,%D2' ignored
```

ADDX.L Dy, Dx

Operation: Source + Destination + CCR[X] → Destination

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

Instruction Format:

15	14	13	12	11 10 9	8	7	6	5	4	3	2 1 0
1	1	0	1	Register, Dx	1	1	0	0	0	0	Register, Dy

```
1 0026  D983      ADDX.L    %D3,  %D4
2 0028  D983      ADDX.L    %D3,  %D4
```

Dodawanie 64-bitowych liczb całkowitych

```
ADD.L     %D0, %D2      |    [D2] <= [D2] + [D0]
ADDX.L    %D1, %D3      |    [D3] <= [D3] + [D1] + [X]
```

SUB

Operation: Destination – Source → Destination

Assembler Syntax: SUB.L <ea>y,Dx
SUB.L Dy,<ea>x

Assembler Syntax: SUBA.L <ea>y,Ax

Assembler Syntax: SUBI.L #<data>,Dx

Assembler Syntax: SUBQ.L #<data>,<ea>x

Assembler Syntax: SUBX.L Dy,Dx

```
1 001a   93F0 1CEC        SUB.L   -20(%A0, %D1*4), %A1
2 001e   95D0             SUBA.L  (%A0), %A2
3 0020   0481 0000 0003   SUBI.L  #0x03, %D1
4 0026   5782             SUBQ.L  #0x3, %D2
5 0028   9780             SUBX.L  %D0, %D3
```

NEG / NEGX

Operation: 0 – Destination → Destination

Assembler Syntax: NEG.L Dx

Attributes: Size = longword

Operation: 0 – Destination – CCR[X] → Destination

Assembler Syntax: NEGX.L Dx

Condition Codes:	X	N	Z	V	C
	*	*	*	*	*

Generuje liczbę ujemną w kodzie U2

```
1 0008   203C 0000 03E8      MOVE.L  #1000,%D0       | D0 = 1000
2 000e   4480                NEG.L   %D0             | D0 = -1000
```

EXT / EXTB

Operation: Destination Sign-Extended → Destination

Assembler Syntax:

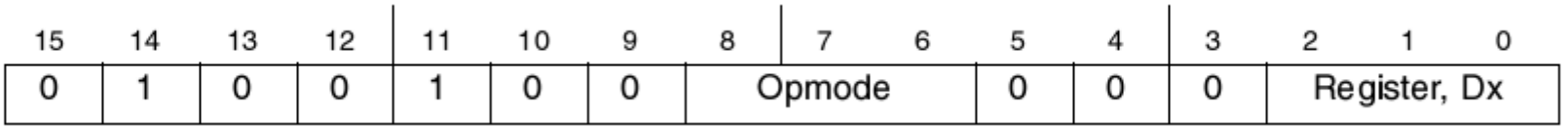
EXT.W Dx	extend byte to word
EXT.L Dx	extend word to longword
EXTB.L Dx	extend byte to longword

Attributes: Size = word, longword

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

Instruction Format:



MOVE.B	# - 0x55,%D0	0xEEEE.EE**AB**	- 85
EXTB.L	%D0	0x**FFFF.FFAB**	0xA = 1010 b
MOVE.B	# - 0x55,%D0	0xEEEE.EE**AB**	
EXT.W	%D0	0xEEEE.**FFAB**	
MOVE.W	# 0xF667,%D0	0xEEEE.**F667**	- 2457
EXT.L	%D0	0x**FFFF.F667**	

Instrukcje arytmetyczne (2)

MULU – operacja mnożenia

		Operacja	Wynik Dx
MULU.W	<ea>y. Dx	16 x 16	32-bit
MULU.L	<ea>y. Dx	32 x 32	32-bit

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

N Set if result is negative; cleared otherwise
Z Set if the result is zero; cleared otherwise

Tryby adresowania argumentu źródłowego (Word) <ea>y:

Dy, (Ay), (Ay)+, -(Ay), (d16,Ay), (d8,Ay,Xi), (xxx).W/L, #<data>, (d16,PC), (d8,PC,Xi)

Tryby adresowania argumentu źródłowego (Longword) <ea>y:

Dy, (Ay), (Ay)+, -(Ay), (d16,Ay)

Bit V rejestru CCR jest zawsze zerowany – nie ma informacji o przepełnieniu

```
MOVE.L  #1000,%D0
MOVE.L  #300,%D1

MULU.W  %D1, %D0
```

```
(gdb) p $d1
$3 = 300
(gdb) p $d0
$4 = 300000
```

MULS – operacja mnożenia (ze znakiem)

		Operacja	Wynik Dx
MULS.W	<ea>y. Dx	16 x 16	32-bit
MULS.L	<ea>y. Dx	32 x 32	32-bit

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

N Set if result is negative; cleared otherwise
Z Set if the result is zero; cleared otherwise

Tryby adresowania argumentu źródłowego (Word) <ea>y:

Dy, (Ay), (Ay)+, -(Ay), (d16,Ay), (d8,Ay,Xi), (xxx).W/L, #<data>, (d16,PC), (d8,PC,Xi)

Tryby adresowania argumentu źródłowego (Longword) <ea>y:

Dy, (Ay), (Ay)+, -(Ay), (d16,Ay)

Bit V rejestru CCR jest zawsze zerowany – nie ma informacji o przepełnieniu

```
MOVE.L  #  -1000,%D0
MOVE.L  #300,%D1

MULS.W  %D1, %D0
```

```
(gdb) p$d0
$1 = -300000
(gdb) p$d1
$2 = 300
```

DIVU (dzielenie liczb dodatnich)

		Operacja	Wynik Dx
DIVU.W	<ea>y. Dx	32-bit Dx/16bit <ea>y	16reszta:16wynik
DIVU.L	<ea>y. Dx	32-bit Dx/32bit <ea>y	32-bit wynik

Condition Codes:

X	N	Z	V	C
—	*	*	*	0

- N Cleared if overflow is detected; otherwise set if the quotient is negative, cleared if positive
- Z Cleared if overflow is detected; otherwise set if the quotient is zero, cleared if nonzero
- V Set if an overflow occurs; cleared otherwise

Tryby adresowania argumentu źródłowego (Word) <ea>y:

Dy, (Ay), (Ay)+, -(Ay), (d16,Ay), (d8,Ay,Xi), (xxx).W/L, #<data>, (d16,PC), (d8,PC,Xi)

Tryby adresowania argumentu źródłowego (Longword) <ea>y:

Dy, (Ay), (Ay)+, -(Ay), (d16,Ay)

```
        MOVE.L  #1000,%D0
        MOVE.L  #20,%D1
        DIVU.W  %D1, %D0
(gdb) p $d1
$3 = 50
(gdb) p/x $ps
$4 = 0x2700
```

```
        MOVE.L  #1000,%D0
        MOVE.L  #300,%D1
        DIVU.W  %D1, %D0
(gdb) p $d1
$3 = 0x64.0003
(gdb) p/x $ps
$4 = 0x2700
```

DIVS (dzielenie liczb ze znakiem)

		Operacja	Wynik Dx
DIVS.W	<ea>y. Dx	32-bit Dx/16bit <ea>y	16reszta:16wynik
DIVS.L	<ea>y. Dx	32-bit Dx/32bit <ea>y	32-bit wynik

Condition Codes:

X	N	Z	V	C
—	*	*	*	0

- N Cleared if overflow is detected; otherwise set if the quotient is negative, cleared if positive
- Z Cleared if overflow is detected; otherwise set if the quotient is zero, cleared if nonzero
- V Set if an overflow occurs; cleared otherwise

Tryby adresowania argumentu źródłowego (Word) <ea>y:

Dy, (Ay), (Ay)+, -(Ay), (d16,Ay), (d8,Ay,Xi), (xxx).W/L, #<data>, (d16,PC), (d8,PC,Xi)

Tryby adresowania argumentu źródłowego (Longword) <ea>y:

Dy, (Ay), (Ay)+, -(Ay), (d16,Ay)

```
        MOVE.L  #  - 1000,%D0
        MOVE.L  #20,%D1
        DIVU.W  %D1, %D0
(gdb) p $d1
$3 = 0xffce          (0xFFCE= - 50)
(gdb) p/x $ps
$4 = 0x2708
```

```
        MOVE.L  #1000,%D0
        MOVE.L  # - 300,%D1
        DIVS.W  %D1, %D0
(gdb) p $d1
$1 = 0x64.fffd          (0xFFFD= - 3)
(gdb) p/x $ps
$4 = 0x2708
```

REMS / REMU (reszta z dzielenia)

		Operacja	Wynik
REMS.L	<ea>y. Dw:Dx	32-bit Dx/32bit <ea>y	32-bit reszta w Dw
REMU.L	<ea>y. Dw:Dx	32-bit Dx/32bit <ea>y	32-bit reszta w Dw

Condition Codes:

X	N	Z	V	C
—	*	*	*	0

N Cleared if overflow is detected; otherwise set if the quotient is negative, cleared if positive
Z Cleared if overflow is detected; otherwise set if the quotient is zero, cleared if nonzero
V Set if an overflow occurs; cleared otherwise

Tryby adresowania argumentu źródłowego (Longword) <ea>y:

Dy, (Ay), (Ay)+, -(Ay), (d16,Ay)

```
MOVE.L  # 1000,%D0
MOVE.L  # 300,%D1
MOVE.L  # 0xD2D2D2D2,%D2

REMU.L  %D1, %D2:%D0
```

```
(gdb) p $d2
      $7 = 100
(gdb) p $d1
      $8 = 300
(gdb) p $d0
      $9 = 1000
(gdb)
```

Instrukcje sterujące przepływem programu (1)

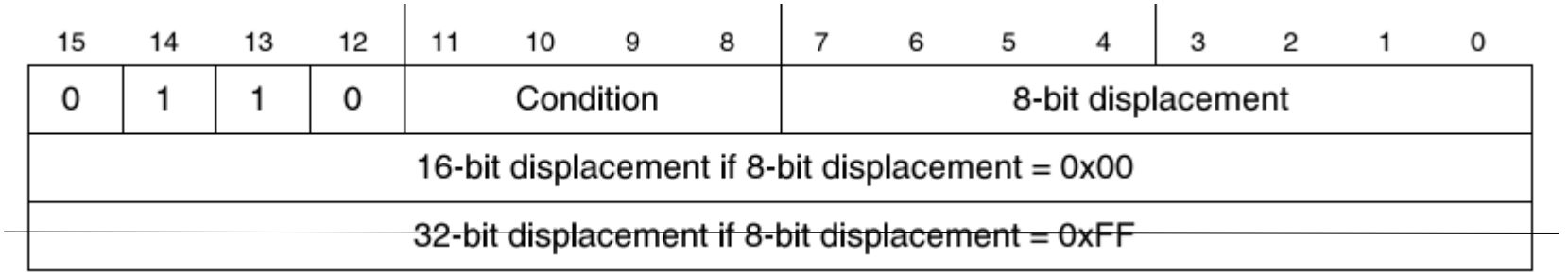
Instruction	Operand Syntax	Operand Size	Operation	ISA, (E)MAC, or FPU
Conditional				
Bcc	<label>	B, W, L	If Condition True, Then PC + d_n → PC	ISA_A[1]
Scc	Dx	B	If Condition True, Then 1s → Destination; Else 0s → Destination	ISA_A
Unconditional				
BRA	<label>	B, W, L	PC + d_n → PC	ISA_A[1]
BSR	<label>	B, W, L	SP – 4 → SP; nextPC → (SP); PC + d_n → PC	ISA_A[1]
JMP	<ea>y	none	Source Address → PC	ISA_A
JSR	<ea>y	none	SP – 4 → SP; nextPC → (SP); Source → PC	ISA_A
NOP	none	none	PC + 2 → PC (Integer Pipeline Synchronized)	ISA_A
TPF	none #<data> #<data>	none W L	IPC + 2→ PC PC + 4 → PC PC + 6→ PC	ISA_A
Returns				ISA_A
RTS	none	none	(SP) → PC; SP + 4 → SP	
Test Operand				
TST	<ea>y	B, W, L	Source Operand Tested → CCR	ISA_A

Bcc.B <label>, Bcc.W <label>

Operation: If Condition True Then PC + $d_n \rightarrow$ PC

Code	Condition	Encod-ing	Test	Code	Condition	Encod-ing	Test
CC(HS)	Carry clear	0100	$\overline{C}$	LS	Lower or same	0011	C | Z
CS(LO)	Carry set	0101	C	LT	Less than	1101	N & $\overline{V}$ | $\overline{N}$ & V
EQ	Equal	0111	Z	MI	Minus	1011	N
GE	Greater or equal	1100	N & V | $\overline{N}$ & $\overline{V}$	NE	Not equal	0110	$\overline{Z}$
GT	Greater than	1110	N & V & $\overline{Z}$ | $\overline{N}$ & $\overline{V}$ & $\overline{Z}$	PL	Plus	1010	$\overline{N}$
HI	High	0010	$\overline{C}$ & $\overline{Z}$	VC	Overflow clear	1000	$\overline{V}$
LE	Less or equal	1111	Z | N & $\overline{V}$ | $\overline{N}$ & V	VS	Overflow set	1001	V

Instruction Format:



BCC.L read_UART Error: Invalid instr. for this architecture; needs 5407,... statement ignored

BCC.W read_UART

Struktura warunkowa typu SWITCH (1)

```
short      TEST;
        ...
        ...
        switch (TEST){

        case       0:
                .....
        break;

        case       1:
                .....
        break;

        case       2:
                .....
        break;

        default:
                .....
        break;
}
```

```
TEST:           .byte  0x0001
.align 2
```

Struktura warunkowa typu SWITCH (2)

```
CASE:
        MOVE.B          TEST, %D0       | sprawdź zmienną TEST
        EXTB.L          %D0
        BEQ.S       ACT1
        SUBQ.L          #1, %D0
        BEQ.S       ACT2
        SUBQ.L          #1, %D0
        BEQ.S       ACT3
OTHERS:
        CLR.B           %D1
        BRA.S       EXIT

ACT1:
        CLR.B           %D2     | case if TEST=0
        BRA.S       EXIT

ACT2:
        CLR.B           %D3     | case if TEST=1
        BRA.S       EXIT

ACT3:
       CLR.B            %D4     | case if TEST=2
       BRA.W        EXIT

EXIT:
```

Konwersja HEX na ASCII (1)

HEX	->	ASCII
0	->	?
1	->	?
...		...
A	->	?
...		...
F	->	?

D1 - licznik 0-15
D0 - wynik ASCII

ASCII: .byte 0x00, 0x00, 0x00...

	0	1	2	3	4	5	6	7
0	NUL	DLE	space	0	@	P	`	p
1	SOH	DC1 XON	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3 XOFF	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	del

Rozliczenie godzinowe zajęć

Zajęcia zaległe:

24.10	3 h
31.10	2 h

Zajęcia odrobione:

8.10	2 h
15.10	2 h

	-1 h

Zajęcia, które nie odbędą się

12.11	-
14.11	3 h
5.12	3 h

Konwersja HEX na ASCII (2)

```
0                           LEA         ASCII, %A3
...                              ...
21 0000 4281                CLR.L       %D1
22
23                    NEXT:
24 0002 2001                MOVE.L      %D1,%D0
25 0004 0680 0000           ADDI.L      #0x30, %D0
                0030
26 000a 0C80 0000           CMPI.L      #0x39, %D0
                0039
27 0010 6302                BLS.B       EXIT
28 0012 5E80                ADDQ.L      #0x7, %D0
29 0014 2600          EXIT: MOVE.L      %D0, (%A3)+
30
31 0016 5281                ADDQ.L      #1,%D1
32 0018                     CMPI.L      #16, %D1
34                          BNE         NEXT
```

skok -32768...+32767

```
27 0010 6300 0004       BLS.W       EXIT
28 0014 5E80                ADDQ.L      #0x7, %D0
29 0016 2600          EXIT: MOVE.L      %D0, %D3
```

	0	1	2	3	4	5	6	7
0	NUL	DLE	space	0	@	P	`	p
1	SOH	DC1 XON	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3 XOFF	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	del

Bcc – tablica warunków

Code	Condition	Encod-ing	Test	Code	Condition	Encod-ing	Test
CC(HS)	Carry clear	0100	$\overline{C}$	LS	Lower or same	0011	C | Z
CS(LO)	Carry set	0101	C	LT	Less than	1101	N & $\overline{V}$ | $\overline{N}$ & V
EQ	Equal	0111	Z	MI	Minus	1011	N
F	False	0001	0	NE	Not equal	0110	$\overline{Z}$
GE	Greater or equal	1100	N & V | $\overline{N}$ & $\overline{V}$	PL	Plus	1010	$\overline{N}$
GT	Greater than	1110	N & V & $\overline{Z}$ | $\overline{N}$ & $\overline{V}$ & $\overline{Z}$	T	True	0000	1
HI	High	0010	$\overline{C}$ & $\overline{Z}$	VC	Overflow clear	1000	$\overline{V}$
LE	Less or equal	1111	Z | N & $\overline{V}$ | $\overline{N}$ & V	VS	Overflow set	1001	V

Scc.B Dx

If Condition True Then 1s → Destination Else 0s → Destination

```
   BSR        GET_DATA
| GET_DATA zwraca dana lub -1 w przypadku
| niepowodzenia

   CLR.B      FLAG
   TST.L          %D0

   BMI.S      NEXT
   MOVE.B         #0xFF, %D1
   MOVE.B         %D1, FLAG

NEXT:

   HALT
```

Code	Condition	Encoding	Test
PL	Plus	1010	$\overline{N}$
MI	Minus	1011	N

```
SPL.B          %D1
MOVE.B         %D1,FLAG
```

N Set if the operand Is negative; cleared otherwise

Instruction Format:

15	14	13	12	11 10 9 8	7	6	5	4	3	2 1 0
0	1	0	1	Condition	1	1	0	0	0	Register, Dx

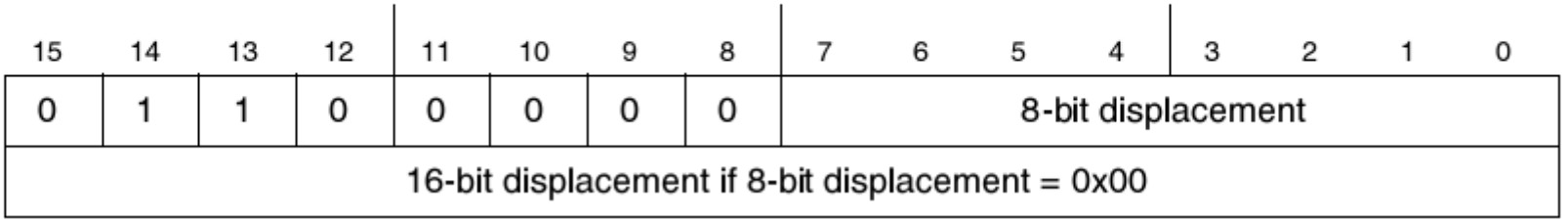
BRA.sz <label>

Operation: Destination Address → PC

BRA - skok bezwarunkowy względny (-128...+127 lub -32768...+32767)

Condition codes: Not affected

Instruction Format:



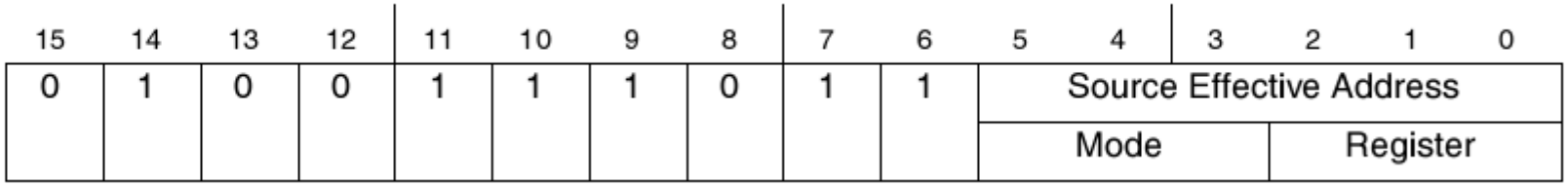

```
20 0002   6000 0076        BRA.W    LCD_subroutine
22 0006   6072             BRA.B    LCD_subroutine
23 0008   60FF 0000 0070   BRA.L    LCD_subroutine   Error: invalid instruction for
this architecture; needs 68020, ...  statement `bra.l exit' ignored
```

JMP <ea>y

Operation: Destination Address → PC

Skok bezwarunkowy do adresu absolutnego podanego jako argument. Instrukcja umożliwia wykonanie skoku w zakresie dostępnej przestrzeni adresowej (4 GB, w kodzie instrukcji umieszczony zostaje 32-bitowy adres).

Instruction Format:



Tryby adresowania argumentu <ea>y:

(Ay), (d16,Ay), (d8,Ay,Xi), **(xxx).W/L**, (d16,PC), (d8,PC,Xi)

```
25 000a    4ED0              JMP    (%A0)
26 000c    4EF0 7CF6         JMP    -10(%A0, %D7*4)
27 0010    4EF8 FF00         JMP     0xFFFFFF00
28 0018    4EF9 EFFF FF00    JMP     0xEFFFFF00
```

JSR <ea>y <=> BSR.sz <label>

JSR <ea>y **Operation:** $SP - 4 \rightarrow SP;\ nextPC \rightarrow (SP);\ Destination\ Address \rightarrow PC$

Skok bezwarunkowy do po podprogramu (adres absolutny podany jako argument).
Instrukcja umożliwia wykonanie skoku w zakresie dostępnej przestrzeni adresowej (4 GB, w kodzie instrukcji umieszczony zostaje 32-bitowy adres).
Programy wykonujące skok do etykiet są nierelokowalne. Przykład: obsługa wyjątków.

Tryby adresowania argumentu <ea>y:

(Ay), (d16,Ay), (d8,Ay,Xi), **(xxx).W/L**, (d16,PC), (d8,PC,Xi)

BSR.sz <label> **Operation:** $SP - 4 \rightarrow SP;\ nextPC \rightarrow (SP);\ PC + d_n \rightarrow PC$

Skok bezwarunkowy do po podprogramu (podobnie jak JSR) jednak adres podawany jest w postaci względnej.
Programy wykonujące skok przy pomocy BSR są relokowalne.

```
        JSR     EXCEPTION_1
        .
        .
        .
EXCEPTION_1:
        RTS
```

```
        BSR.W   LED_display
        .
        .
        .
LED_display:
        RTS
```

NOP

Instrukcja nie wykonująca żadnej operacji. Zwiększa licznik programu. Instrukcja synchronizuje potok. Wykorzystywania do generowania niewielkich opóźnień.

```
                                                MOVE.L  #0, %D0
                                                MOVE.L  #1000, %D1

  .                                             NEXT:
  .
  .                                             NOP
275     01b4 4E71           NOP                 NOP
276     01b6 4E71           NOP                 NOP
277     01b8 4E71           NOP                 NOP
  .                                             NOP
  .                                             NOP
  .
                                                ADDQ.L  #1, %D0
                                                CMP.L   %D1, %D0
                                                BNE NEXT

                                                EXIT:
```

TFP.sz #<data>

Instrukcja nie wykonująca żadnej operacji. Zwiększa licznik programu. Instrukcja nie synchronizuje potoku. Wykorzystywania do wyrównywania programu.

TPF	PC + 2 → PC
TPF.W #<data>	PC + 4 → PC
TPF.L #<data>	PC + 6 → PC

```
if (a == b)     z = 1;
else            z = 2;
```

```
        CMP.L       %D0, %D1
        BEQ.B       label0
        MOVEQ.L     #2, %d2
TPF.W   #2  ---->   BRA.B       label1

label0:
        MOVE.L      #1, %d2
label1:
```

Condition Codes: Not affected

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	0	1	1	1	1	1	1	Opmode		
Optional Immediate Word															
Optional Immediate Word															

TST.sz <ea>y

Instrukcja porównuje podany operand z wartością równą zero, ustawia bity rejestru statusowego SR.

Attributes: Size = byte, word, longword

```
while (Data == 0)
{
        Data--;
}
```

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

```
NEXT:   TST.L     Data
        BEQ.B     EXIT
        SUBQ.L    #1, Data
        BRA.B     NEXT
EXIT:
          ...
Data:   .long     5
```

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	Size		Destination Effective Address					
										Mode			Register		

Tryby adresowania argumentu <ea>y:

Dy, Ay, (Ay), (Ay)+, -(Ay), (d16,Ay), (d8,Ay,Xi), **(xxx).W/L**, #<data>, (d16,PC), (d8,PC,Xi)

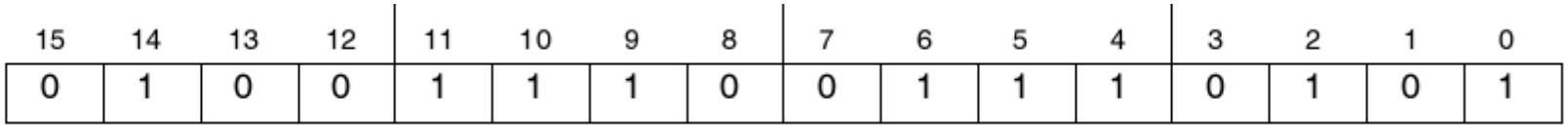
RTS

Operation: (SP) → PC; SP + 4 → SP

Assembler Syntax: RTS

Attributes: Unsized

Instruction Format:



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	0	1

Instrukcje transferu danych

Instruction	Operand Syntax	Operand Size	Operation	ISA, (E)MAC, or FPU
LEA	<ea>y,Ax	L	<ea>y → Ax	ISA_A
LINK	Ay,#<displacement>	W	SP – 4 → SP; Ay → (SP); SP → Ay, SP + d_n → SP	ISA_A
MOVCLR	ACCy,Rx	L	Accumulator → Destination, 0 → Accumulator	EMAC
MOVE MOVE from CCR MOVE to CCR	<ea>y,<ea>x MACcr,Dx <ea>y,MACcr CCR,Dx <ea>y,CCR	B,W,L L L W W	Source → Destination where MACcr can be any MAC control register: ACCx, ACCext01, ACCext23, MACSR, MASK	ISA_A[1] MAC[2] MAC[2] ISA_A ISA_A
MOVEA	<ea>y,Ax	W,L → L	Source → Destination	ISA_A
MOVEM	#list,<ea>x <ea>y,#list	L	Listed Registers → Destination Source → Listed Registers	ISA_A
MOVEQ	#<data>,Dx	B → L	Immediate Data → Destination	ISA_A
PEA	<ea>y	L	SP – 4 → SP; <ea>y → (SP)	ISA_A
UNLK	Ax	none	Ax → SP; (SP) → Ax; SP + 4 → SP	ISA_A

Instrukcje przenoszenia danych operują na 8, 16 lub 32 bitach. Umożliwiają przesłanie danych z pamięci do pamięci, z pamięci do rejestru lub odwrotnie. Procesory wyposażone w jednostką do obliczeń zmiennoprzecinkowych posiadają rozbudowaną liczbę instrukcji operujących na liczbach zmiennoprzecinkowych pojedynczej lub podwójnej precyzji.

LEA.L <ea>y, Ax

Operation: <ea>y → Ax

Ładuje obliczony adres efektywny do rejestru Ax. Nie modyfikuje CCR.

```
MOVE.L      #0x100, %D4

LEA         0x10000000, %A5
LEA         0x5(%A0, %D4.L*2), %A6
```

```
(gdb) p $fp
$2 = (void *) 0x10000205
```

Instruction Format:

15	14	13	12	11 10 9	8	7	6	5 4	3 2 1 0
0	1	0	0	Register, Ax	1	1	1	Source Effective Address	
								Mode	Register

Tryby adresowania argumentu <ea>y:

(Ay), (d16,Ay), (d8,Ay,Xi), **(xxx).W/L**, (d16,PC), (d8,PC,Xi)

Program relokowalny

Odczyt danej z tablicy:

```
MOVE.B      TABLE(%PC), %D0
```

Zapis danej do tablicy:

```
MOVE.B      %D0, TABLE(%PC)
```

```
Error: operands mismatch --
statement `move.b %D0,TABLE(%PC)' ignored
```

```
LEA         TABLE(%PC), %A0
MOVE.B      %D0, (%A0)
```

```
TABLE:   .byte   0x1, 0x2, 0x3, 0x44, 0x55, 0x66
```

PEA.L <ea>y

Operation: SP – 4 → SP; <ea>y → (SP)

```
MOVEA.L    #0xF0000000, %A0
MOVE.L     #0x100, %D0

PEA        -4(%A0, %D0*2)
```

```
(gdb) bt 8
#0  _stext () at gp.asm:27
#1  0xf00001fc     in ?? ()
#2  0x00000000     in ?? ()
#3  0x00000000     in ?? ()
#4  0xaffbcaff     in ?? ()
#5  0xafdfecff     in ?? ()
#6  0xeffffeff     in ?? ()
#7  0xdffeebff     in ?? ()
```

Condition Codes: Not affected

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	Source Effective Address					
										Mode			Register		

Tryby adresowania argumentu <ea>y:

(Ay), (d16,Ay), (d8,Ay,Xi), **(xxx).W/L**, (d16,PC), (d8,PC,Xi)

MOVE.sz <ea>y, <ea>x

Operation: Source → Destination

Attributes: Size = byte, word, longword

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

Tryby adresowania argumentu źródłowego <ea>y:

Dy, Ay, (Ay), (Ay)+, -(Ay), (d16,Ay), (d8,Ay,Xi), (xxx).W/L, #<data>, (d16,PC), (d8,PC,Xi)

Tryby adresowania argumentu docelowego <ea>x:

Dx, (Ax), (Ax)+, -(Ax), (d16,Ax), (d8,Ax,Xi), (xxx).W/L, (d16,PC), (d8,PC,Xi)

Ograniczenia:

Source Addressing Mode	Destination Addressing Mode
Dy, Ay, (Ay), (Ay)+,-(Ay)	All possible
(d_{16}, Ay), (d16, PC)	All possible except (d_8, Ax, Xi), (xxx).W, (xxx).L
(d8, Ay, Xi), (d8, PC, Xi), (xxx).W, (xxx).L, #<xxx>	All possible except (d_{16}, Ax), (d_8, Ax, Xi), (xxx).W, (xxx).L

MOVE.L 0x100, -0x10(%A5)

Error: Operands mismatch – statement 'move.l 0x100, -0x10(%A5)' ignored

MOVEA.sz <ea>y, Ax

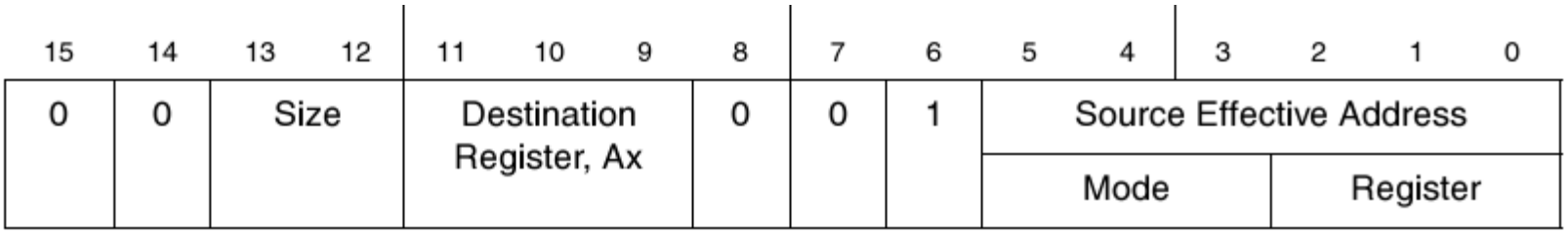
Operation: Source → Destination

Assembler Syntax: MOVEA.sz <ea>y,Ax

Attributes: Size = word, longword

Condition Codes: Not affected

Instruction Format:



Tryby adresowania argumentu źródłowego <ea>y:

Dy, Ay, (Ay), (Ay)+, -(Ay), (d16,Ay), (d8,Ay,Xi), (xxx).W/L, #<data>, (d16,PC), (d8,PC,Xi)

MOVEM.L #list, <ea>x (1)

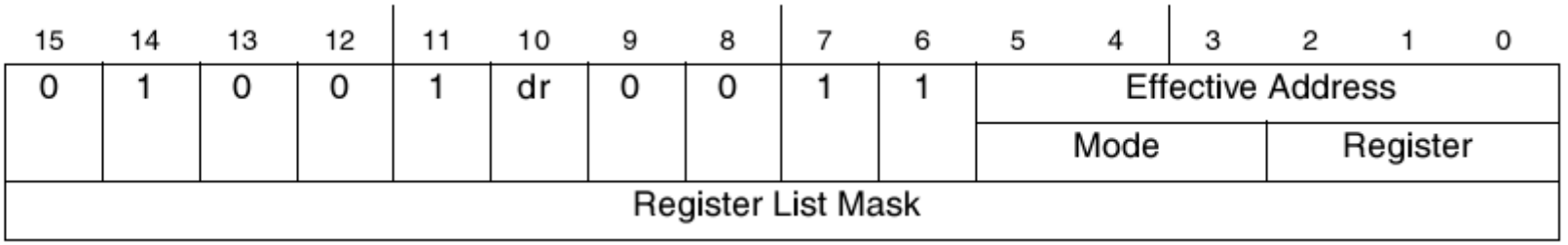
Operation: Registers → Destination;
Source → Registers

Assembler Syntax: MOVEM.L #list,<ea>x
MOVEM.L <ea>y,#list

Attributes: Size = longword

Tryby adresowania argumentu docelowego <ea>x i źródłowego <ea>y:
(Ax), (d16,Ax), (Ay), (d16,Ay)

Instruction Format:



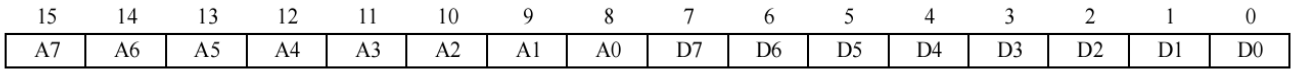
gp.asm:21: Error: operands mismatch -- statement

MOVEM.L %D0-%D2/%A0-%A2, -(%A7) `movem.l %D0-%D2/%A0-%A2,-(%A7)' ignored

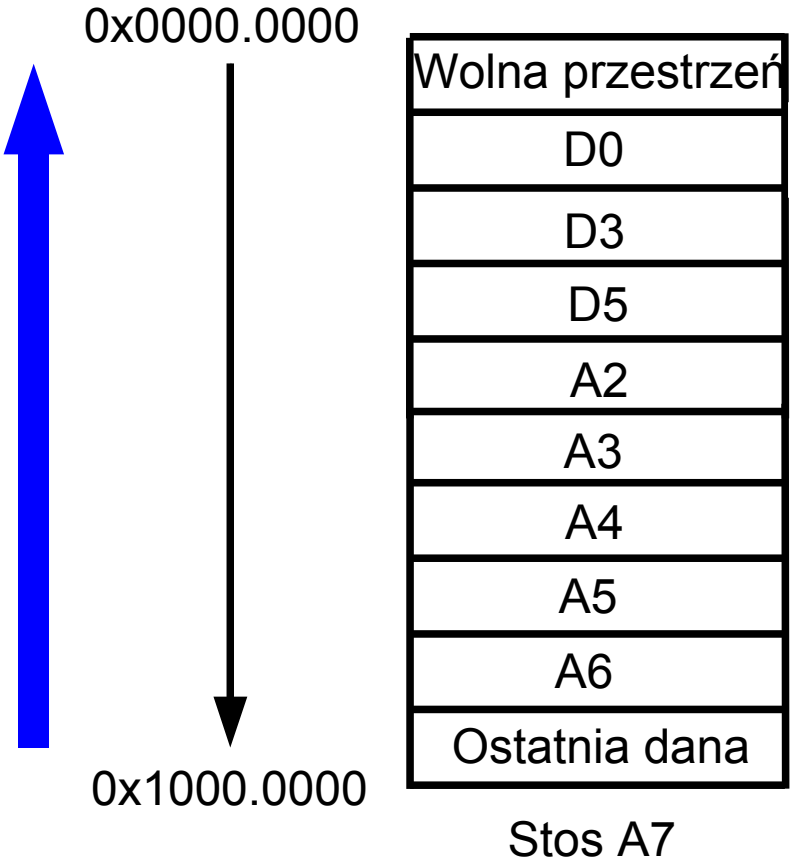
MOVEM.L (%A7)+, %D0-%D2/%A0-%A2 `movem.l (%A7)+,%D0-%D2/%A0-%A2' ignored

MOVEM.L %D0-%D2/%A0-%A2, 0x100 `movem.l %D0-%D2/%A0-%A2,0x100' ignored

MOVEM.L #list, <ea>x (2)




```
move.l      #0x010000d0,%d0
move.l      #0x020000d3,%d3
move.l      #0x030000d5,%d5
move.l      #0x040000a2,%a2
move.l      #0x050000a3,%a3
move.l      #0x060000a4,%a4
move.l      #0x070000a5,%a5
move.l      #0x080000a6,%a6

LEA         -32(%SP), %SP

MOVEM.L     %D0/%A2-%A6/%D3/%D5,
                        (%A7)
  ...
MOVEM.L     (%A7),%D0/%A2-
                        %A6/%D3/%D5

LEA         32(%SP), %SP
```




```
(gdb) x/9x 0x5c64
0x5c64: 0x010000d0    0x020000d3    0x030000d5    0x040000a2
0x5c74: 0x050000a3    0x060000a4    0x070000a5    0x080000a6
0x5c84: 0x00000000
```

MOVEQ.L #<data>, Dx

Operation: Immediate Data → Destination

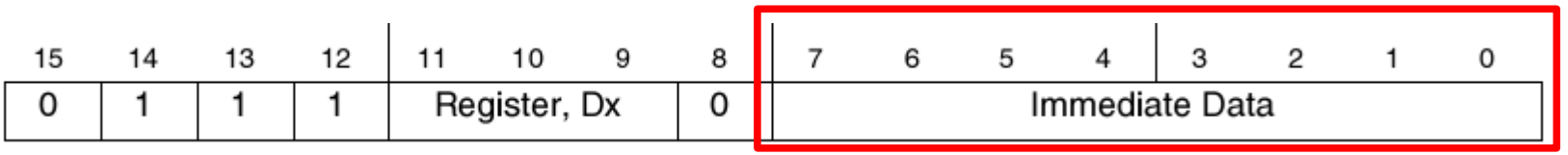
Assembler Syntax: MOVEQ.L #<data>,Dx

Attributes: Size = longword

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

Instruction Format:



MOVEQ.L #0x1FF, %D0

| gp.asm:24: Error: operands mismatch --
| statement `moveq.l #0x1FF,%D0' ignored

MOVE.L #0x1FF, %D0

MOVE.W CCR, Dx

MOVE.W %CCR, %D0

Operation: CCR → Destination

Assembler Syntax: MOVE.W CCR,Dx

Attributes: Size = Word

MOVE.W %D0, %CCR
MOVE.W %CCR, %D0

MOVE.W #<data>, %CCR

Operation: Source → CCR

Assembler Syntax: MOVE.B Dy,CCR
MOVE.B #<data>,CCR

Attributes: Size = Byte

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

(gdb) p/x $ps
$4 = 0x2700
1: x/i $pc 0x10008 <MAIN+4>: **move.w #0xFFFF,%CCR**
(gdb) p/x $ps
$5 = 0x27**1F**

Instrukcje logiczne

Instruction	Operand Syntax	Operand Size	Operation	First appeared: ISA, (E)MAC or FPU
AND	<ea>y,Dx Dy,<ea>x	L L	Source & Destination → Destination	ISA_A
ANDI	#<data>, Dx	L	Immediate Data & Destination → Destination	ISA_A
EOR	Dy,<ea>x	L	Source ^ Destination → Destination	ISA_A
EORI	#<data>,Dx	L	Immediate Data ^ Destination → Destination	ISA_A
NOT	Dx	L	~ Destination → Destination	ISA_A
OR	<ea>y,Dx Dy,<ea>x	L L	Source | Destination → Destination	ISA_A
ORI	#<data>,Dx	L	Immediate Data | Destination → Destination	ISA_A

AND, EOR, NOT, OR - Logiczne operacje arytmetyczne, operand pobierany z rejestru lub pamięci

ANDI, EORI, ORI - Logiczne operacje arytmetyczne, operand w postaci natychmiastowej

AND.L <ea>y, Dx / AND.L Dy, <ea>x

Condition Codes:	X	N	Z	V	C
	—	*	*	0	0

Tryby adresowania argumentu źródłowego <ea>y:

Dy, (Ay), (Ay)+, -(Ay), (d16,Ay), (d8,Ay,Xi), (xxx).W/L, #<data>, (d16,PC), (d8,PC,Xi)

Tryby adresowania argumentu docelowego <ea>x:

(Ax), (Ax)+, -(Ax), (d16,Ax), (d8,Ax,Xi), (xxx).W/L

```
AND.L    %D0, %D0
```

```
gp.asm:29: Error: operands mismatch --
 statement `and.l %A1,%A1' ignored
statement `and.l %A1,%D1' ignored
 statement `and.l %D1,%A1' ignored
```

```
AND.L   %A1,%A1
AND.L   %A1,%D1
AND.L   %D1,%A1
```

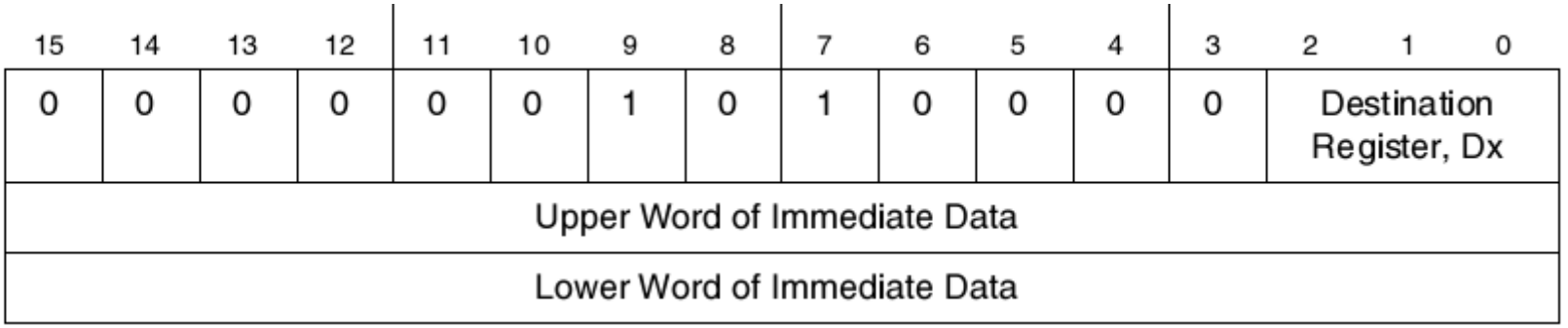
ANDI.L #<data>, Dx

Attributes: Size = longword

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

Instruction Format:




```
MOVE.L      #0xDEADBEEF, %D1
ANDI.L      #0x00FFFF00, %D1
```

```
(gdb) p/x $d1
$2 = 0xdeadbeef

44      ANDI.L      #0x00FFFF00, %D1
1: x/i $pc  0x10010 <MAIN+12>:  andil #0x00ffff00,%d1
(gdb) s
(gdb) p/x $d1
$3 = 0x00ADBE00
```

Przykłady

```
MOVE.L    #0xABCDEF12, %D0
MOVE.L    #0xABCDEF12, %D1
MOVE.L    #0xABCDEF12, %D2
MOVE.L    #0xABCDEF12, %D3

ANDI.L    0xFF00 0000, %D0

ORI.L     0xFF00 0000, %D1

EORI.L    0xFF00 0000, %D2

NOT.L     %D3

MOVE.L    #0xABCDEF12, %D0
EOR.L     %D0, %D0
```

```
(gdb) p/x $d0
$1 = 0xab000000

(gdb) p/x $d1
$2 = 0xffcdef12

(gdb) p/x $d2
$3 = 0x54cdef12

(gdb) p/x $d3
$4 = 0x543210ed

(gdb) p/x $d0
$4 = 0x0
```

Operacje przesunięć

Instruction	Operand Syntax	Operand Size	Operation	First appeared: ISA, (E)MAC or FPU
ASL	Dy,Dx #<data>,Dx	L L	CCR[X,C] ← (Dx << Dy) ← 0 CCR[X,C] ← (Dx << #<data>) ← 0	ISA_A
ASR	Dy,Dx #<data>,Dx	L L	msb → (Dx >> Dy) → CCR[X,C] msb → (Dx >> #<data>) → CCR[X,C]	ISA_A
LSL	Dy,Dx #<data>,Dx	L L	CCR[X,C] ← (Dx << Dy) ← 0 CCR[X,C] ← (Dx << #<data>) ← 0	ISA_A
LSR	Dy,Dx #<data>,Dx	L L	0 → (Dx >> Dy) → CCR[X,C] 0 → (Dx >> #<data>) → CCR[X,C]	ISA_A
SWAP	Dx	W	MSW of Dx ↔ LSW of Dx	ISA_A

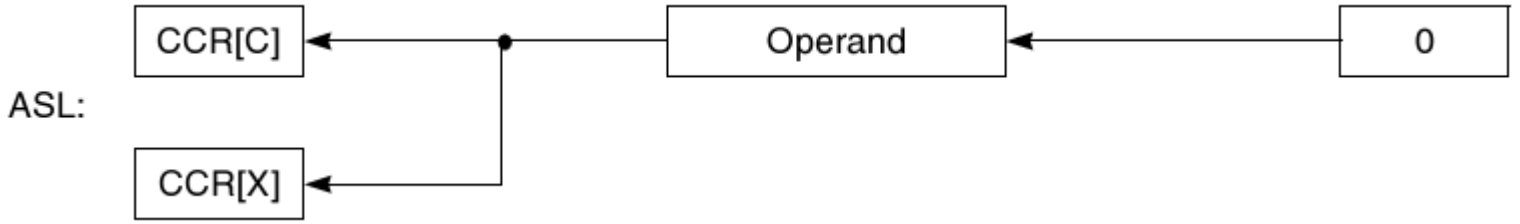
Instrukcje przesunięcia operują wyłącznie na rejestrach. Operacje wykonywane są na podwójnych słowach (DW). Przesunięcie definiowane jest przez podany argument (od 1 do 8) lub określony rejestrem (przesunięcie modulo 64).

Operacje przesunięć arytmetycznych ASL / ASR

Operation: Destination Shifted By Count → Destination

Assembler Syntax: ASd.L Dy,Dx
ASd.L #<data>,Dx
where d is direction, L or R

Attributes: Size = longword



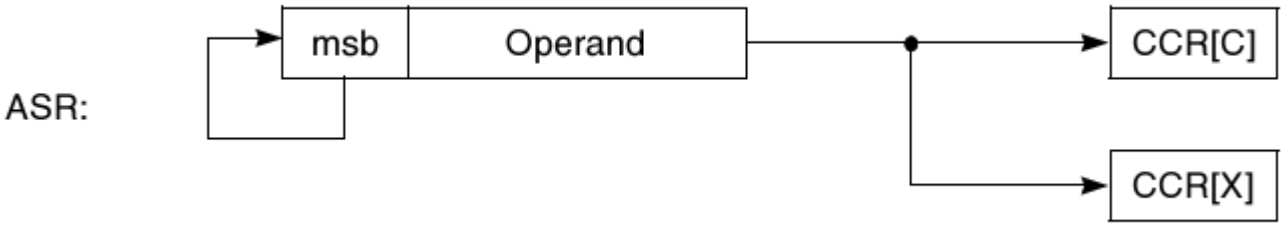

```
MOVE.L    #0x12345678,%D0
ASL.L     #4, %D0
```

D0 = #0x23456780

```
MOVE.L    #0x12345678,%D0
ASL.L     #8, %D0
```

D0 = #0x34567800

ASR.L #<data>, Dx



Operacja przesunięcia arytmetycznego w prawo zachowuje znak liczby zapisanej w kodzie U2.

```
MOVE.L      #0x12345678,%D0
ASR.L       #4, %D0
```
D0 = #0x01234567

```
MOVE.L      #0x82345678,%D0
ASR.L #4,   %D0
```
D0 = #0xf8123456

```
MOVE.L      #0x82345678,%D0
ASR.L       #8, %D0
```
D0 = #0xff812345

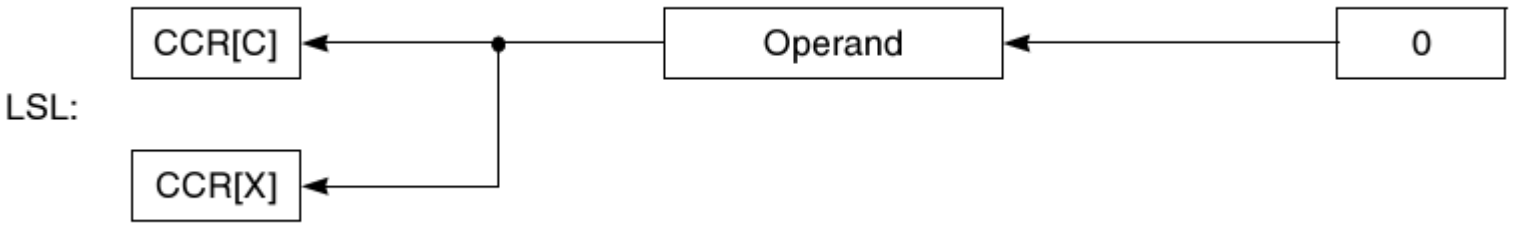
```
MOVE.L      #0x82345678,%D0
ASR.L    #9, %D0
```
gp.asm:31: Error: operands mismatch -- statement `asl.l #12,%D0' ignored

Operacje przesunięć logicznych

Operation: Destination Shifted By Count → Destination

Assembler Syntax: LSd.L Dy,Dx
LSd.L #<data>,Dx
where d is direction, L or R

Attributes: Size = longword




```
MOVE.L  #0x12345678,%D0
LSL.L #4, %D0
```

D0 = 0x23456780

```
MOVE.L  #0x12345678,%D0
LSL.L #8, %D0
```

D0 = 0x34567800

```
MOVE.L  #0x2,%D0
LSL.L #1, %D0
LSL.L #1, %D0
LSL.L #1, %D0
```

D0 = 0x4
D0 = 0x8
D0 = 0x10

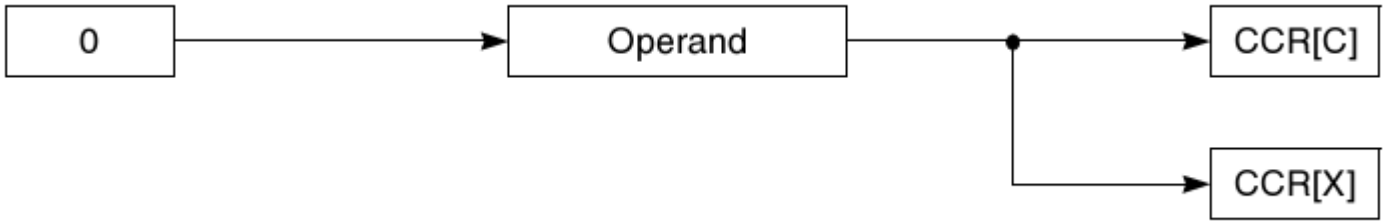
LSR

Operation: Destination Shifted By Count → Destination

Assembler Syntax: LSd.L Dy,Dx
LSd.L #<data>,Dx
where d is direction, L or R

Attributes: Size = longword

LSR:




```
MOVE.L  #0x81234567,%D0
LSR.L #4, %D0
```

D0 = 0x0812.3456

```
MOVE.L  #0x81234567,%D0
LSR.L #8, %D0
```

D0 = 0x0081.2345

```
MOVE.L  #0x10,%D0
LSR.L #1, %D0
LSR.L #1, %D0
LSR.L #1, %D0
```

D0 = 0x8
D0 = 0x4
D0 = 0x2

SWAP.W Dx

Operation: Register[31:16] ↔ Register[15:0]

Attributes: Size = Word

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	0	0	0	Register, Dx		

Użycie SWAP

```
CLR.L   %D7
CLR.L   %D6
MOVE.L  # 1000,%D0
MOVE.L  # 300,%D1

DIVU.W  %D1, %D0

MOVE.W  %D0, %D7
SWAP.W  %D0
MOVE.W  %D0, %D6
HALT
```

```
(gdb) p $d0
$1 = 1000
(gdb) p $d1
$2 = 300

(gdb) p $d0
$3 = 0x640003

29      SWAP.W  %D0
1: x/i $pc  0x1001e <MAIN+26>:  swap %d0

(gdb) p $d0
$4 = 0x30064

(gdb) p/x $d6
$5 = 0x64
(gdb) p/x $d7
$6 = 0x3
```

Instrukcje operujące na bitach

Instruction	Operand Syntax	Operand Size	Operation	First appeared: ISA, (E)MAC or FPU
BCHG	Dy,<ea>x #<data>,<ea>x	B, L B, L	~ (<bit number> of Destination) → CCR[Z] → <bit number> of Destination	ISA_A
BCLR	Dy,<ea>x #<data>,<ea>x	B, L B, L	~ (<bit number> of Destination) → CCR[Z]; 0 →<bit number> of Destination	ISA_A
BITREV	Dx	L	Bit reversed Dx → Dx	ISA_A+, ISA_C
BSET	Dy,<ea>x #<data>,<ea>x	B, L B, L	~ (<bit number> of Destination) → CCR[Z]; 1 → <bit number> of Destination	ISA_A
BYTEREV	Dx	L	Byte reversed Dx → Dx	ISA_A+, ISA_C
BTST	Dy,<ea>x #<data>,<ea>x	B, L B, L	~ (<bit number> of Destination) → CCR[Z]	ISA_A
FF1	Dx	L	Bit offset of First Logical One “1” in Dx → Dx	ISA_A+, ISA_C

Instrukcje operujące na bitach wykorzystują dane zapisane w rejestrach lub pamięci. Operacje wykonywane są na podwójnych słowach (DW).

BITREV.L Dx

Operation: Bit Reversed Dx → Dx

Condition Codes: Not affected

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	1	0	0	0	Register, Dx		

Dx[0] = Dx'[31]
Dx[1] = Dx'[30]
...
...
...
Dx[31] = Dx'[0]

```
MOVE.L    #0x84216666, %D1
BITREV.L  %D1
```

```
(gdb) p/x $d1
$1 = 0x84216666
```

1000.0100.0010.0001.0110.0110.0110.0110

```
(gdb) s
28              BITREV.L %D1
1: x/i $pc  0x1000a <MAIN+6>:   bitrev %d1

(gdb) p/x $d1
$2 = 0x66668421
```

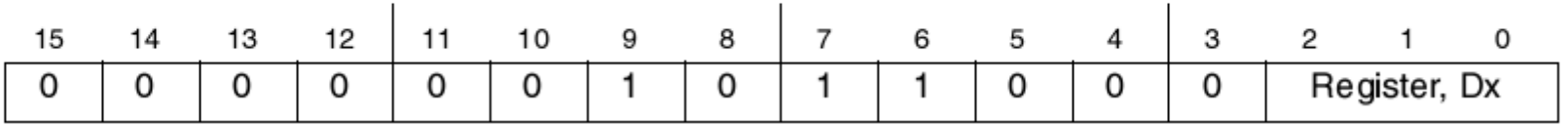
0110.0110.0110.0110.1000.0100.0010.0001

BYTEREV.L Dx

Operation: Byte Reversed Dx → Dx

Condition Codes: Not affected

Instruction Format:



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	1	1	0	0	0	Register, Dx		

new Dx[31:24] = old Dx[7:0]

new Dx[23:16] = old Dx[15:8]

new Dx[15:8] = old Dx[23:16]

new Dx[7:0] = old Dx[31:24]

```
MOVE.L      #0xABCDEF12, %D1
BYTEREV.L   %D1
```

```
(gdb) p/x $d1
$2 = 0xabcdef12

(gdb) s
28              BYTEREV.L %D1
1: x/i $pc  0x1000a <MAIN+6>:   byterev %d1

(gdb) p/x $d1
$3 = 0x12efcdab
```

BTST.sz Dy, <ea>x / BTST.sz #<d>, <ea>x

Operation: ~ (<bit number> of Destination) → CCR[Z]

Assembler Syntax: BTST.sz #<data>,<ea>x

Attributes: Size = byte, longword

Tylko dla Dy

Condition Codes:

X	N	Z	V	C
—	—	*	—	—

Z Set if the bit tested is zero; cleared otherwise

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	Destination Effective Address					
										Mode		Register			
0	0	0	0	0	0	0	0	Bit Number							

Tryby adresowania argumentu docelowego <ea>x:

Dx, (Ax), (Ax)+, -(Ax), (d16,Ax)

Assembler Syntax: BTST.sz Dy,<ea>x

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	Data Register, Dy			1	0	0	Destination Effective Address					
										Mode		Register			

Tryby adresowania argumentu docelowego <ea>x:

Dx, (Ax), (Ax)+, -(Ax), (d16,Ax), (d8,Ax,Xi), (xxx).W/L, (d16,PC), (d8,PC,Xi)

BTST.sz #<data>, <ea>x

Condition Codes:	X	N	Z	V	C
	—	—	*	—	—

Z Set if the bit tested is zero; cleared otherwise

```
MOVE.L #0x00000001, %D2
BTST.L    #32, %d2
```

SR = 0x2700

```
MOVE.L #0x00000001, %D3
BTST.L    #31, %d3
```

SR = 0x2704

```
MOVE.L #0x80000000, %D4
BTST.L    #32, %d4
```

SR = 0x270c 0xc= 1100

Przykład zastosowania BTST

Napisać program zliczający liczbę jedynek w binarnym słowie.

D0 – 32-bit dane wejściowe, wynik 6-bit
D1 – licznik bitów o wartości 1
D2 – wskaźnik badanego bitu 0-31

Przykład zastosowania BTST

```
MOVE.L  #0x11112222,%D0
MOVE.L  #0xD1D1D1D1,%D1
MOVE.L  #0xD2D2D2D2,%D2

BSR     ONES_COUNT
HALT


ONES_COUNT:  LEA      -8(%A7), %A7
             MOVEM.L  D1-D2, %A7
             CLR.L    %D1
             MOVEQ    #31, %D2
NEXT_BIT:    BTST     %D2, %D0
             BEQ.S    LOOP_TEST
             ADDQ.L   #1, %D1
LOOP_TEST:   SUBQ.L   #1, %D2
             BGE      NEXT_BIT
             MOVE.L   %D1, %D0
             MOVEM.L  %A7, %D1-D2%
             LEA      8(%A7), %A7
             RTS
```

```
(gdb) bt 7
#0  NEXT_BIT () at gp.asm:35
#1  0xd1d1d1d1 in ?? ()
#2  0xd2d2d2d2 in ?? ()
#3  0x0001001a in ONES_COUNT
                        () at gp.asm:25

#4  0x00000000 in ?? ()
#5  0x00000000 in ?? ()
#6  0x00000000 in ?? ()
```

```
(gdb) p/x $d0
$4 = 0x8
(gdb) p/x $d1
$5 = 0xd1d1d1d1
(gdb) p/x $d2
$6 = 0xd2d2d2d2
```

BCLR.sz Dy,<ea>x / BCLR.sz #<d>,<ea>x

Operation: ~ (<bit number> of Destination) → CCR[Z];
0 → <bit number> of Destination

Assembler Syntax: BCLR.sz Dy,<ea>x
BCLR.sz #<data>,<ea>x

Attributes: Size = byte, longword

Condition Codes:

X	N	Z	V	C
—	—	*	—	—

Z Set if the bit tested is zero; cleared otherwise

BSET.sz Dy,<ea>x / BSET.sz #<d>,<ea>x

Operation: ~ (<bit number> of Destination) → CCR[Z];
1 → <bit number> of Destination

Assembler Syntax: BSET.sz Dy,<ea>x
BSET.sz #<data >,<ea>x

Attributes: Size = byte, longword

Condition Codes:

X	N	Z	V	C
—	—	*	—	—

Z Set if the bit tested is zero; cleared otherwise

BCHG.sz Dy,<ea>x / BCHG.sz #<d>,<ea>x

Operation: ~ (<bit number> of Destination) → CCR[Z];
~ (<bit number> of Destination) → <bit number> of Destination

Assembler Syntax: BCHG.sz Dy,<ea>x
BCHG.sz #<data>,<ea>x

Attributes: Size = byte, longword

Condition Codes:	X	N	Z	V	C
	—	—	*	—	—

Z Set if the bit tested is zero; cleared otherwise

FF1.L Dx

Operation: Bit Offset of the First Logical One in Register → Destination

Assembler Syntax: FF1.L Dx

Attributes: Size = longword

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

Old Dx[31:0]	New Dx[31:0]
0x8000 0000	0x0000 0000
0x4000 0000	0x0000 0001
0x2000 0000	0x0000 0002
...	...
0x0000 0002	0x0000 001E
0x0000 0001	0x0000 001F
0x0000 0000	0x0000 0020

```
MOVE.L  #0x80000000, %D5        (gdb) p/x $d5
FF1.L   %D5                     $4 = 0x0

MOVE.L  #0x08000000, %D5        (gdb) p/x $d5
FF1.L   %D5                     $4 = 0x4

MOVE.L  #0x00500000, %D5        (gdb) p/x $d5
FF1.L   %D5                     $4 = 0x9

MOVE.L  #0x0000000, %D5         (gdb) p/x $d5
FF1.L   %D5                     $4 = 0x20
```

Operacje z użyciem stosu

PEA.L <ea>y

Operation: SP – 4 → SP; <ea>y → (SP)

```
MOVEA.L    #0xF0000000, %A0
MOVE.L     #0x100, %D0

PEA        -4(%A0, %D0*2)
```

```
(gdb) bt 8
#0  _stext () at gp.asm:27
#1  0xf00001fc     in ?? ()
#2  0x00000000     in ?? ()
#3  0x00000000     in ?? ()
#4  0xaffbcaff     in ?? ()
#5  0xafdfecff     in ?? ()
#6  0xeffffeff     in ?? ()
#7  0xdffeebff     in ?? ()
```

Condition Codes: Not affected

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	Source Effective Address					
										Mode			Register		

Tryby adresowania argumentu <ea>y:

(Ay), (d16,Ay), (d8,Ay,Xi), **(xxx).W/L**, (d16,PC), (d8,PC,Xi)

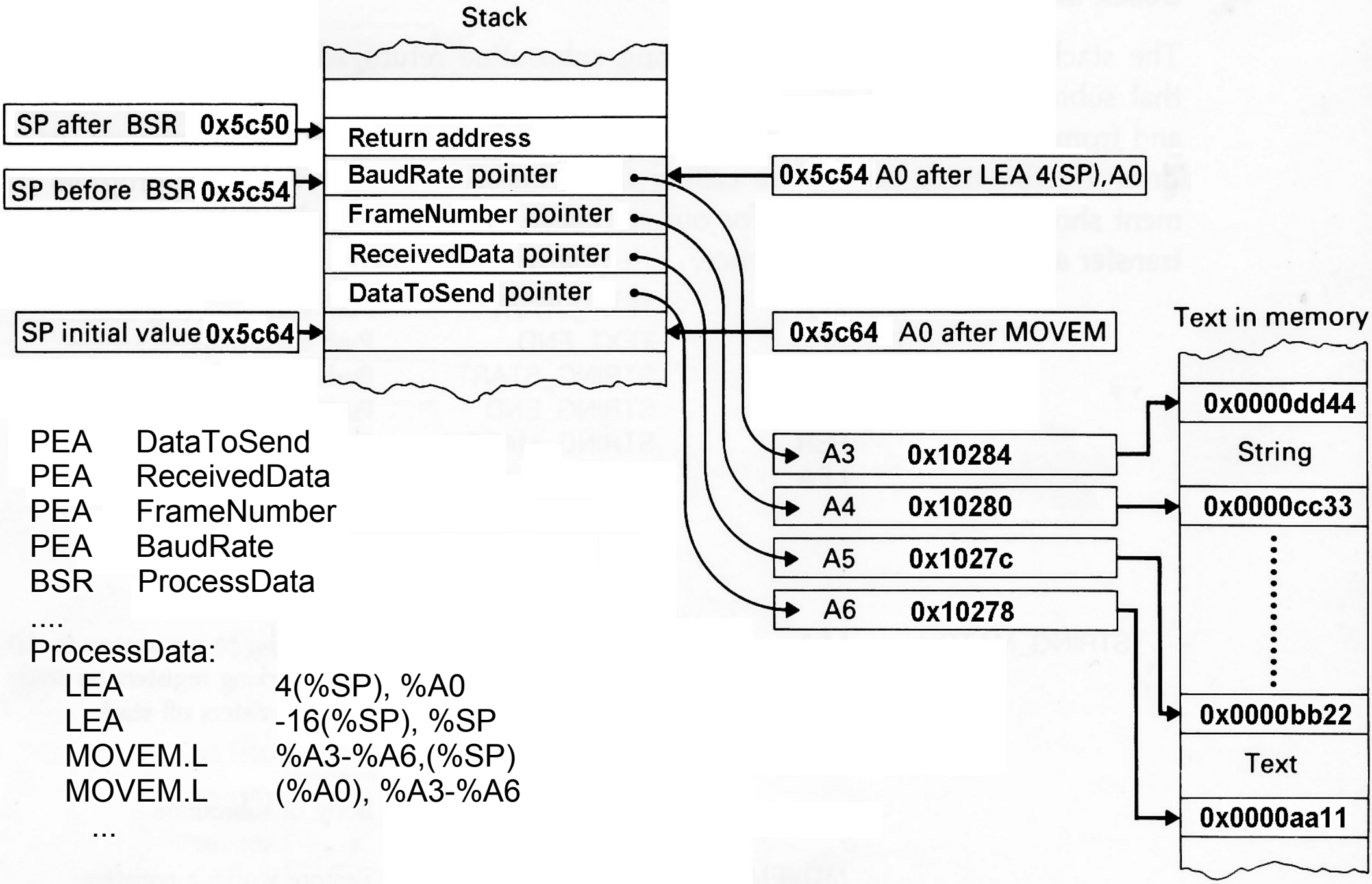
Przekazywanie parametrów do procedury (1)

```
void ProcessData (int &DataToSend,
                  int &ReceivedData,
                  int &FrameNumber,
                  int &BaudRate) {
    ...
    ...
}

void main (void) {
    int DataToSend=0xAA11;
    int ReceivedData=0xBB22;
    int FrameNumber=0xCC33;
    int BaudRate=0xDD44;

Process Data (DataToSend,ReceivedData,
FrameNumber, BaudRate);
}
```

```
    ...
    ...
DataToSend:     .long   0xAA11
ReceivedData:   .long   0xBB22
FrameNumber:    .long   0xCC33
BaudRate:       .long   0xDD44
.align 4
```

```
MOVEA.L     #0x0, %A0
MOVEA.L     #0x3, %A3
MOVEA.L     #0x4, %A4
MOVEA.L     #0x5, %A5
MOVEA.L     #0x6, %A6

PEA     DataToSend
PEA     ReceivedData
PEA     FrameNumber
PEA     BaudRate
BSR     ProcessData
LEA             16(%SP), %SP
HALT

ProcessData:
   LEA          4(%SP), %A0
   LEA          -16(%SP), %SP
   MOVEM.L      %A3-%A6,(%SP)
   MOVEM.L      (%A0), %A3-%A6
     ...
     ...
   MOVEM.L      (%SP), %A3-%A6
   LEA          16(%SP), %SP
   RTS
```

Stos po wykonaniu skoku do podprogramu




```
PEA     DataToSend
PEA     ReceivedData
PEA     FrameNumber
PEA     BaudRate
BSR     ProcessData
....
ProcessData:
    LEA             4(%SP), %A0
    LEA             -16(%SP), %SP
    MOVEM.L         %A3-%A6,(%SP)
    MOVEM.L         (%A0), %A3-%A6
        ...
```

Stos po wykonaniu drugiej inst. MOVEM

Mapa pamięci stosu:

```
(gdb) bt 12
#0  ProcessData () at gp.asm:43
#1  0x00000003 in ?? ()
#2  0x00000004 in ?? ()
#3  0x00000005 in ?? ()
#4  0x00000006 in ?? ()
#5  0x00010034 in _stext () at gp.asm:35
#6  0x00010284 in BaudRate () at gp.asm:342
#7  0x00010280 in FrameNumber () at gp.asm:342
#8  0x0001027C in ReceivedData () at gp.asm:342
#9  0x00010278 in DataToSend () at gp.asm:342
#10 0x00000000 ?? ()
#11 0xe9affbff in ?? ()
```

Mapa pamięci w której przechowywane są zmienne:

```
(gdb) x/4x 0x10278
0x10278 <DataToSend>:   0x0000aa11      0x0000bb22
                        0x0000cc33      0x0000dd44
```

Rejestry procesora przed wykonaniem instrukcji MOVEM:

```
(gdb) info r
a0      0x5c54      0x5c54
a3      0x3         0x3
a4      0x4         0x4
a5      0x5         0x5
fp      0x6         0x6
sp      0x5c40      0x5c40
ps      0x2708      9992
pc      0x10046     0x10046
```

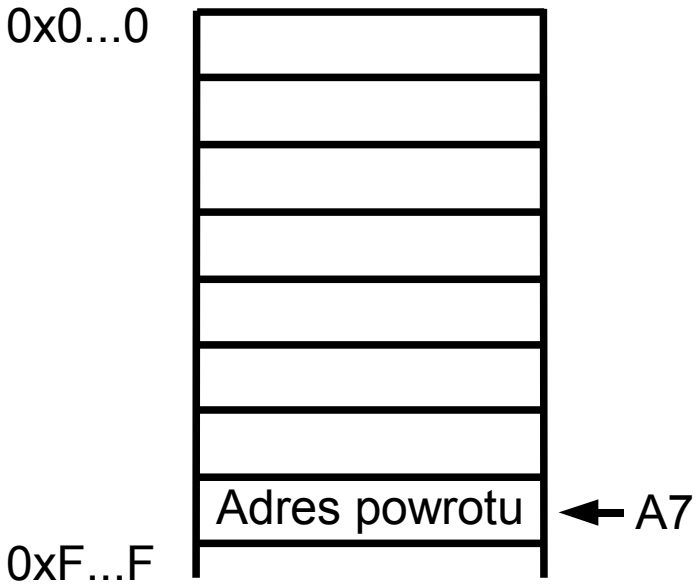
Rejestry procesora po wykonaniu drugiej instrukcji MOVEM:

```
(gdb) info r
a0      0x5c54      0x5c54
a3      0x10284     0x10284
a4      0x10280     0x10280
a5      0x1027c     0x1027c
fp      0x10278     0x10278
sp      0x5c40      0x5c40
ps      0x2704      9988
pc      0x1005e     0x1005e
```

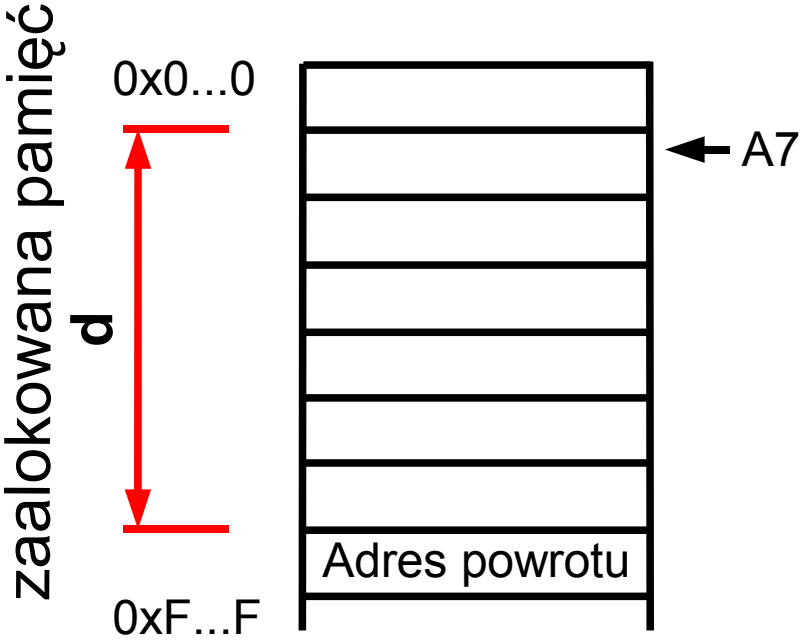
Alokacja pamięci na stosie

Podprogramy wykorzystują struktury umieszczone na stosie przeznaczone do przechowywania:

- parametrów przekazywanych do podprogramu (wartości oraz adresy),
- adres powrotu z podprogramu,
- kopia rejestrów roboczych podprogramu (D0-D7/A0-A5),
- kopia poprzedniej wartości rejestru ramki A6,
- zarezerwowany obszar pamięci na stosie przeznaczony na zmienne lokalne.

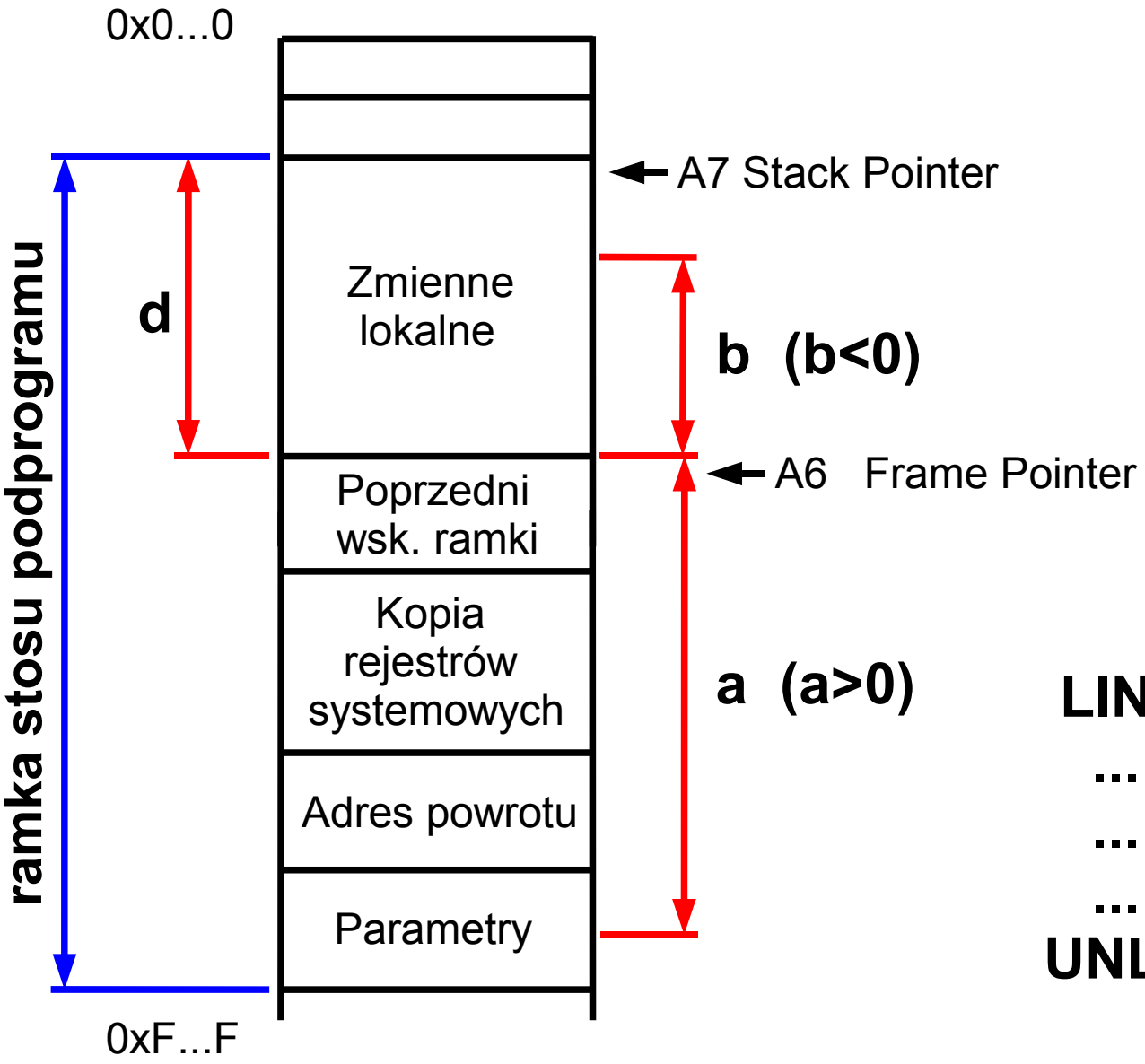


BSR



LEA -d(%A7), %A7

Ramka stosu




```
LINK    %FP, # -size
...
...
...
UNLINK  %FP
```

Link.W Ay, #<displacement> / Unlink Ax

Alokacja ramki stosu – LINK Ay, # rozmiar

Operation: $SP - 4 \rightarrow SP;\ Ay \rightarrow (SP);\ SP \rightarrow Ay;\ SP + d_n \rightarrow SP$

Condition Codes: Not affected

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	1	0	Register, Ay		
Word Displacement															

Dealokacja ramki stosu – UNLINK Ax

Operation: $Ax \rightarrow SP;\ (SP) \rightarrow Ax;\ SP + 4 \rightarrow SP$

Condition Codes: Not affected

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	1	1	Register, Ax		

Przykład

Napisać procedurę obliczającą wartość R równania:

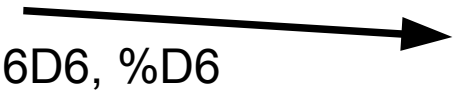
$$R = (P^2 + Q^2)/(P^2 - Q^2).$$

Parametry P oraz Q przekazywane są przez stos, natomiast obliczona wartość R zwracana jest przez referencję.

```
CALC (short P, short Q , short &R)
{
        int temp_1;
        int temp_2;

        temp_1 = P * P;
        temp_2 = temp_1;
        temp_1 = temp_1 + Q * Q;
        temp_2 = temp_2 - Q * Q;

        R = temp_1 / temp_2;
}
```

Program główny

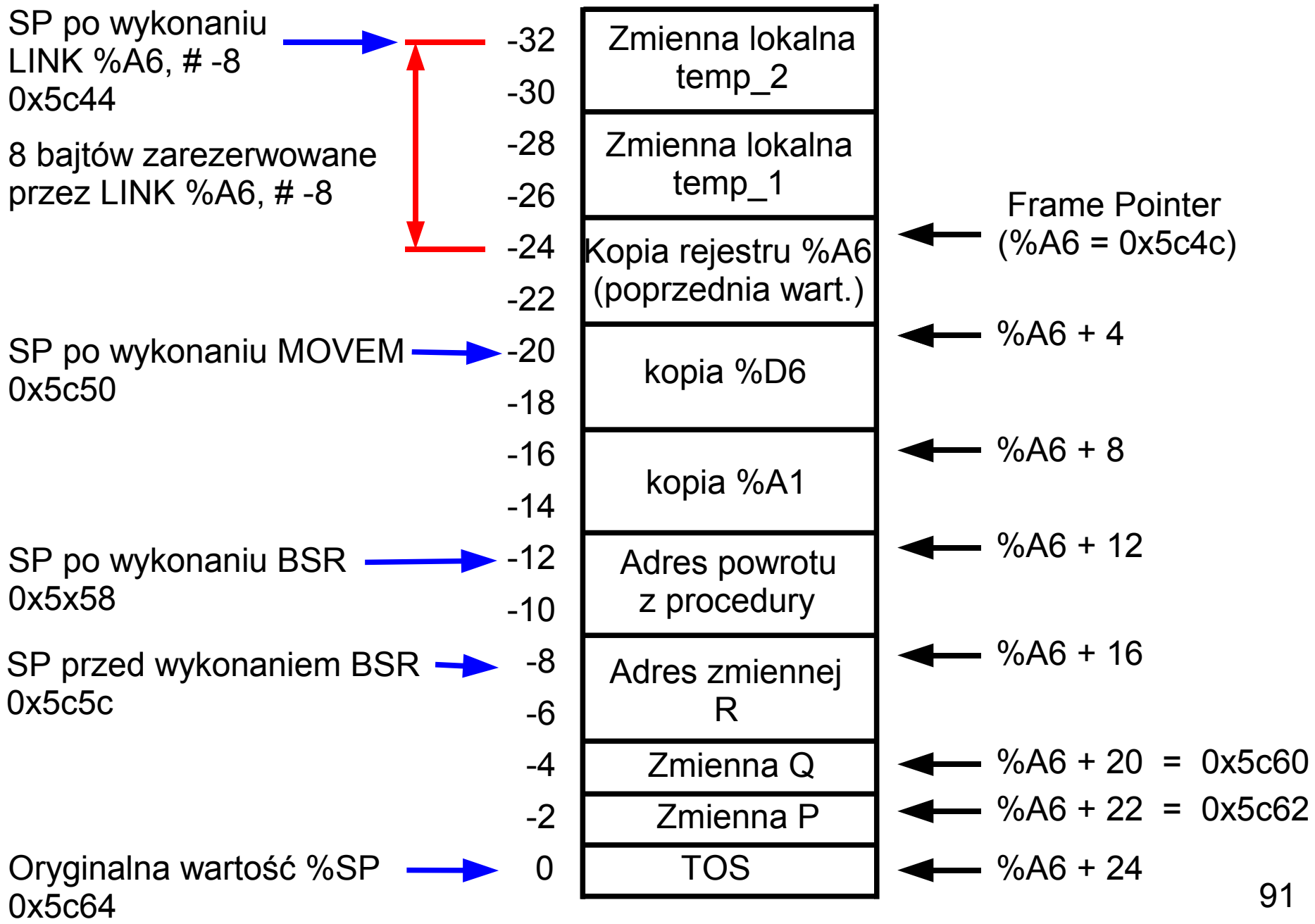
```
MAIN:

    MOVEA.L     # 0x00A6A600, %A6
    MOVEA.L     # 0xA10000A1, %A1
    MOVEQ.L     # 5, %D0
    MOVEQ.L     # 4, %D1
    MOVE.L      # 0xD6D6D6D6, %D6

    MOVE.W      %D0,-(%SP)          | odłóż zmienna P na stos
    MOVE.W      %D1,-(%SP)          | odłóż zmienna Q na stos
    PEA         R                   | odłóż referencję do R na stosie
    BSR         CALC                | wywołanie procedury CALC

    LEA         8(%SP),%SP          | usuń zmienne ze stosu P,Q,R –
                                    | dealokacja pamięci na stosie
    HALT

R:          .word  0xABCD
.align 4
```



Podprogram

```
CALC:
    LEA         -8(%SP), %SP        | rezerwacja miejsca na stosie (kopia D6, A1)
    MOVEM.L     %D6/%A1,(%SP)       | wykonanie kopi rejestrów D6, A1
    LINK        %A6, # -8           | rezerwacja miejsca na zmienne temp_1, temp_2
    MOVE.W      22(%A6), %D6        | zapisanie zmiennej P do D6
    MULU        %D6,%D6             | P * P
    MOVE.L      %D6, -4(%A6)        | zapisanie wyniku mnożenia do temp_1
    MOVE.L      %D6, -8(%A6)        | zapisanie wyniku mnożenia do temp_2
    MOVE.W      20(%A6), %D6        | odczytanie zmiennej Q do D6
    MULU        %D6, %D6            | Q * Q
    ADD.L       %D6, -4(%A6)        | dodanie temp_1 do D6 (P^2 + Q^2)
    SUB.L       %D6, -8(%A6)        | odjęcie D6 od temp_2 (P^2 - Q^2)
    MOVE.L      -4(%A6), %D6        | zapisanie (P^2 + Q^2) w D6
    DIVU        -6(%A6), %D6        | obliczenie (P^2 + Q^2)/(P^2 - Q^2)
    LEA         16(%A6), %A1        | wyznaczenie miejsca na stosie, gdzie znajduje
                                    | się adres zmiennej R
    MOVEA.L     (%A1),%A1           | odczytanie adresu zmiennej R
    MOVE.W      %D6, (%A1)          | zapisanie wyniku dzielenia (część całkowita)
    UNLK        %A6                 | usuniecie ramki stosu, dealokacja pamięci na stosie
    MOVEM.L     (%SP), %D6/%A1      | odtworzenie wartości rejestrów D6, A1
    LEA         8(%SP), %SP         | dealokacja pamięci na stosie (2 x 4B)
    RTS                             | powrót z podprogramu
```

Struktura stosu po zainicjowaniu zmiennych



Zawartość stosu oraz rejestrów przed obliczeniem Q^2

```
bt 10
#0  CALC () at gp.asm:46
#1  0x00000019 in ?? ()                          | zmienna temp_2
#2  0x00000019 in ?? ()                          | zmienna temp_1
#3  0x00a6a600 in ?? ()                          | kopia rejestru A6
#4  0xd6d6d6d6 in ?? ()                          | kopia rejestru D6
#5  0xa10000a1 in ?? ()                          | kopia rejestru A1
#6  0x00010026 in MAIN () at gp.asm:32           | adres powrotu z proc. CALC
#7  0x000102d8 in R () at gp.asm:386             | referencja do zmiennej R
#8  0x00040005 in ?? ()                          | zmienne Q.W oraz P.W
#9  0x933beaf7 in ?? ()                          | wartość odłożona przed
                                                 | wywołaniem MAIN
```

```
(gdb) info r
d0             0x5         5
d1             0x4         4
d6             0x4         4
fp             0x5c4c      0x5c4c
sp             0x5c44      0x5c44
ps             0x2700      9984
pc             0x1004e     0x1004e <CALC+34>
```

```
D6  = 0xD6D6D6D6
A1  = 0xA10000A1
A6  = 0x00A6A600
P   = 5
Q   = 4
```

```
(gdb) x/w 0x000102d8
0x102d8 <R>:    0xabcd
```

Zawartość stosu oraz rejestrów przed wykonaniem RTS

```
(gdb) bt 7
#0  CALC () at gp.asm:59
#1  0x00010026 in MAIN () at gp.asm:32
#2  0x000102d8 in R at gp.asm:386
#3  0x00040005 in ?? ()
#4  0x933beaf7 in ?? ()
#5  0xe9afbbfc in ?? ()
#6  0xa5a34aff in ?? ()
```

```
(gdb) info r
d0             0x5              5
d1             0x4              4
d6             0xd6d6d6d6       -690563370
a1             0xa10000a1       0xa10000a1
fp             0xa6a600         0xa6a600
sp             0x5c58           0x5c58
ps             0x2700           9984
pc             0x10072          0x10072 <CALC+70>
```

```
(gdb) x/w 0x000102d8
0x102d8 <R>:    0x0004
```

Zawartość stosu oraz rejestrów po usunięci zmiennych P, Q, R ze stosu

```
(gdb) bt 5
#0  MAIN () at gp.asm:34
#1  0x933beaf7 in ?? ()
#2  0xe9afbbfc in ?? ()
#3  0xa5a34aff in ?? ()
#4  0xafdee4e7 in ?? ()
```

```
(gdb) info r
d0             0x5              5
d1             0x4              4
d6             0xd6d6d6d6       -690563370
a1             0xa10000a1       0xa10000a1
fp             0xa6a600         0xa6a600
sp             0x5c64           0x5c64
ps             0x2700           9984
pc             0x1002a          0x1002a <MAIN+38>
```

```
(gdb) x/w 0x000102d8
0x102d8 <R>:    0x0004
```

Systemowe instrukcje sterujące (1)

Instruction	Operand Syntax	Operand Size	Operation	ISA, (E)MAC
Privileged				
HALT	none	none	Halt processor core (synchronizes pipeline)	ISA_A
MOVE from SR	SR,Dx	W	SR → Destination	ISA_A
MOVE to SR	<ea>y,SR	W	Source → SR; Dy or #<data> source only (synchronizes pipeline)	ISA_A
MOVEC	Ry,Rc	L	Ry → Rc (synchronizes pipeline)	ISA_A
RTE	none	none	2 (SP) → SR; 4 (SP) → PC; SP + 8 →SP Adjust stack according to format (synchronizes pipeline)	ISA_A
STOP	#<data>	none	Immediate Data → SR; STOP (synchronizes pipeline)	ISA_A
STLDSR	#<data>	W	SP – 4 → SP; zero-filled SR → (SP); Immediate Data → SR	ISA_A+, ISA_C
WDEBUG	<ea>y	L	Addressed Debug WDMREG Command Executed (synchronizes pipeline)	ISA_A
Debug Functions				
PULSE	none	none	Set PST = 0x4	ISA_A
WDDATA	<ea>y	B, W, L	Source → DDATA port	ISA_A
Trap Generating				
ILLEGAL	none	none	SP – 4 → SP; PC → (SP) → PC; SP – 2 → SP; SR → (SP); SP – 2 → SP; Vector Offset → (SP); (VBR + 0x10) → PC	ISA_A
TRAP	#<vector>	none	1 → S Bit of SR; SP – 4 → SP; nextPC → (SP); SP – 2 → SP; SR → (SP) SP – 2 → SP; Format/Offset → (SP) (VBR + 0x80 +4*n) → PC, where n is the TRAP number	ISA_A

HALT

Operation: If Supervisor State
Then Halt the Processor Core
Else Privilege Violation Exception

Assembler Syntax: HALT

Attributes: Unsized

W trybie superużytkownika instrukcja wstrzymuje pracę procesora po ukończeniu wszystkich rozpoczętych instrukcji lub cykli zapis/odczyt. Daje możliwość łatwego debugowania procesora przez interfejs BDM. Przejście do stanu bezczynności jest sygnalizowane stanem 0xF na liniach diagnostycznych PST. Instrukcja synchronizuje pipeline.
Procesor może wznowić prace po odebraniu rozkazu GO z modułu BDM.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	1	1	0	0	1	0	0	0

MOVE.W <ea>y, SR

Operation: If Supervisor State
Then Source → SR
Else Privilege Violation Exception

Assembler Syntax: MOVE.W <ea>y,SR

Assembler Syntax: MOVE.W SR,Dx

Attributes: Size = word

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

```
(gdb) p/x $ps
$4 = 0x2704
(gdb) s
25              MOVE.W 0xFFFF, %SR
1: x/i $pc  0x10008 <MAIN+4>:   movew 0xffff,%sr
(gdb) p/x $ps
$5 = 0xb71f
```

System Byte **Condition Code Register (CCR)**

15	14	13	12	11	10 … 8	7	6	5	4	3	2	1	0
T	0	S	M	0	I	0	0	0	X	N	Z	V	C

Tryby adresowania argumentu źródłowego <ea>y: Dy, #<data>
Instrukcja synchronizuje potok.

MOVEC.L Ry, Rc

Rejestry systemowe mogą być wyłącznie zapisywane. Jedyną możliwością ich odczytu jest użycie trybu diagnostycznego, np. BDM.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	1	0	1	1
A/D	Register, Ry			Control Register, Rc											

Name	CPU Space Assignment	Register Name
Memory Management Control Registers		
CACR	0x002	Cache control register
ACR0	0x004	Access control registers 0
ACR1	0x005	Access control registers 1
Processor Miscellaneous Registers		
VBR	0x801	Vector base register
PC	0x80F	Program counter
Local Memory and Module Control Registers		
FLASHBAR	0xC04	RAM base address register 0
RAMBAR	0xC05	RAM base address register 1
MBAR	0xC0F	Primary module base address register

RTE

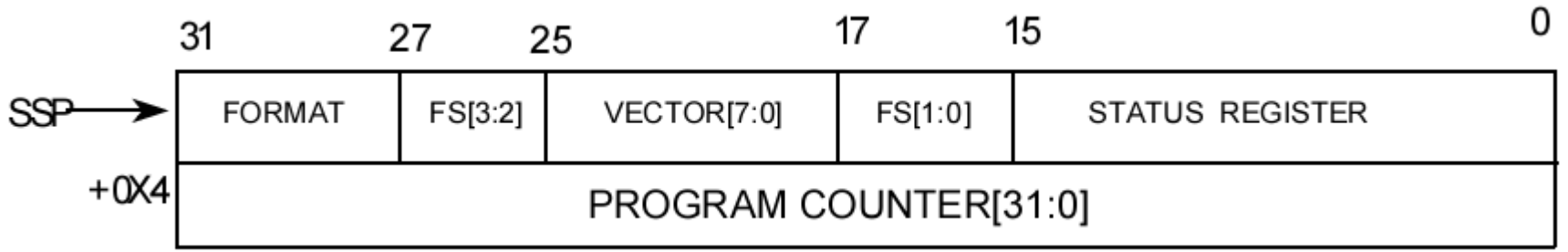
Operation:

If Supervisor State
    Then 2 + (SP) → SR; 4 + (SP) → PC; SP + 8 → SP
    Adjust stack according to format
Else Privilege Violation Exception

Condition Codes: Set according to the condition code bits in the status register value restored from the stack.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	1



STOP #<data>

Instrukcja zapisuje 16 bitowy operand do rejestru statusowego (w zależności od uprawnień użytkownika) oraz powoduje zakończenie wykonywanego zadania. Zawartość licznika programu jest zwiększana, jednak pobieranie nowej instrukcji jest wstrzymane. Instrukcja synchronizuje potok. Po wykonaniu instrukcji STOP procesor przechodzi do trybu obniżonego poboru mocy. Procesor może wznowić pracę w przypadku zaistnienia wyjątku, przerwania lub resetu.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	0
Immediate Data															

STRLDSR #<data>

Operation: If Supervisor State
Then SP - 4 → SP; zero-filled SR → (SP); immediate data → SR
Else TRAP

Assembler Syntax: STRLDSR #<data>

Instrukcja umieszcza kopię rejestru statusowego na stosie, a następnie ładuje rejestr wartością natychmiastową. Jeżeli podczas wykonywania instrukcji bit 13 jest wyzerowany (przejście do trybu użytkownika) nastąpi wygenerowanie wyjątku privilege violation.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	1	1	0	0	1	1	1
0	1	0	0	0	1	1	0	1	1	1	1	1	1	0	0
Immediate Data															

ILLEGAL

Operation: SP - 4 → SP; PC → (SP) (forcing stack to be longword aligned)
SP - 2 → SP; SR → (SP)
SP - 2 → SP; Vector Offset → (SP)
(VBR + 0x10) → PC

Assembler Syntax: ILLEGAL

Attributes: Unsized

Wykonanie instrukcji powoduje wywołanie wyjątku niedozwolonej instrukcji (illegal instruction exception).

Vector Number(S)	Vector Offset (Hex)	Stacked Program Counter	Assignment
4	0x010	Fault	Illegal instruction

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	1	1	1	1	1	1	0	0

Trap #<vector>

Wykonanie polecenia **Trap** powoduje wygenerowanie wyjątku o numerze podanym w postaci operandu.

TRAP #<vector>

1 → S-Bit of SR
SP – 4 → SP; nextPC → (SP); SP – 2 → SP;
SR → (SP); SP – 2 → SP; Format/Offset → (SP);
(VBR + 0x80 + 4*n) → PC
where n is the TRAP vector number

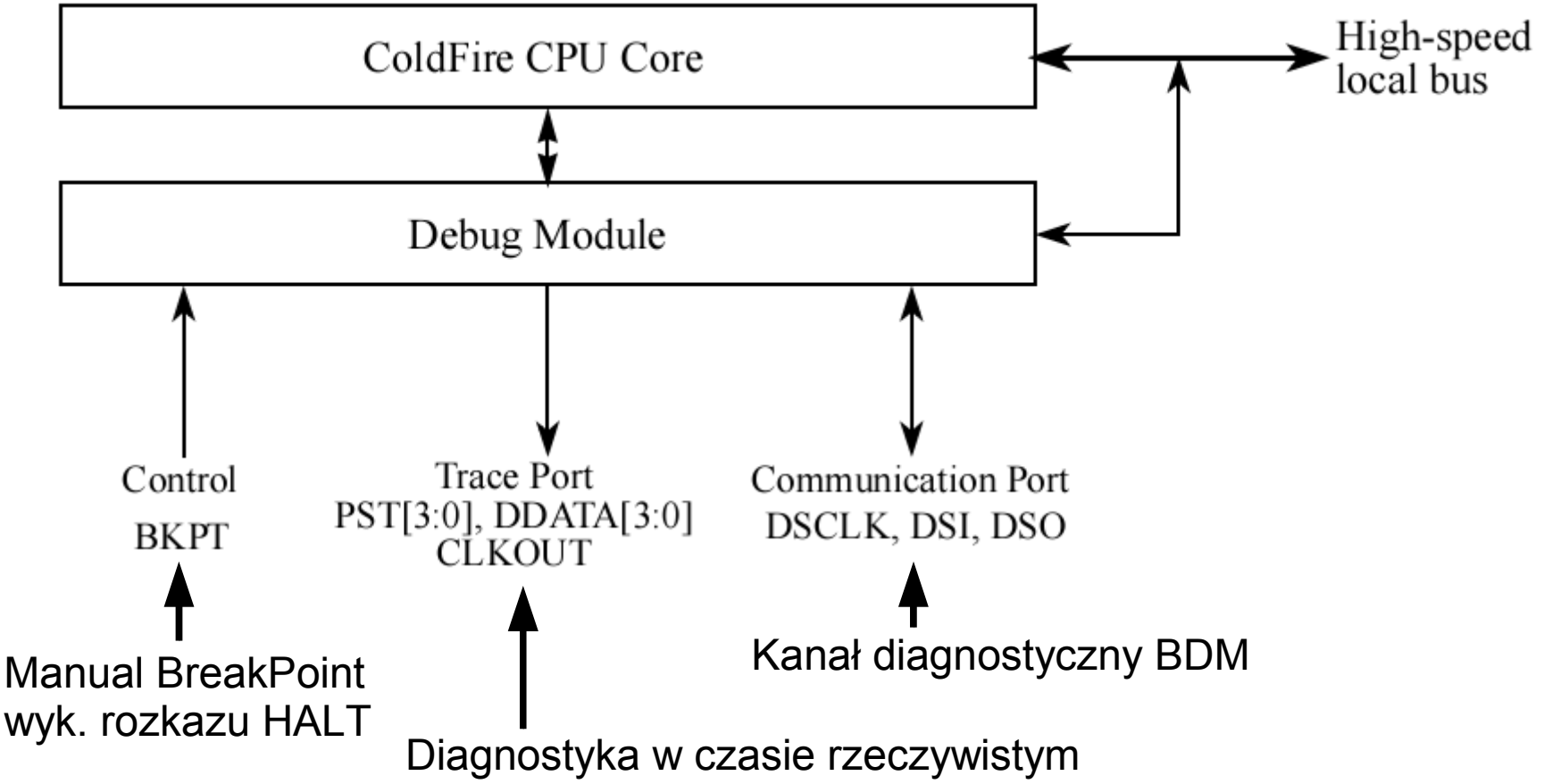
Adres wiersza w tablicy wyjątków obsługującego dany wyjątek można policzyć zgodnie z równaniem:

0x80 + 4 * #<vector>

Moduł procesora ColdFire umożliwia obsłużenie do 16 zdefiniowanych zdarzeń

Pulse

Wykonanie instrukcji powoduje zwiększenie licznika programu PC ustawienie wyjścia diagnostycznego PST na wartość równą 4. Wyjście wykorzystywane jest jako zewnętrzny trigger, co umożliwia debugowanie w czasie rzeczywistym oraz pomiar wydajności systemu.



Condition Codes: Not affected

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	1	1	0	0	1	1	0	0

WDDATA.sz <ea>y

Wykonanie instrukcji powoduje zwiększenie licznika programu PC, ustawienie wyjścia diagnostycznego DDATA (4 bit) na wartość zapisaną w pamięci pod adresem <ea> oraz ustawienie trigger'a PST = 4. Daje to możliwość podglądu pamięci lub pobranych operandów w czasie rzeczywistym.

Attributes: Size = byte, word, longword

Condition Codes: Not affected

Instruction Format:

15	14	13	12	11	10	9	8	7 6	5 4	3 2 1 0
1	1	1	1	1	0	1	1	Size	Source Effective Address	
									Mode	Register

Tryby adresowania argumentu źródłowego <ea>y:

(Ay), (Ay)+, -(Ay), (d16,Ay), (d8,Ay,Xi), (xxx).W/L

WDEBUG.L <ea>y

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	1	1	Source Effective Address					
										Mode			Register		
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

Instrukcja pobiera dwa długie słowa umieszczone w pamięci pod adresem <ea> ((Ay), (d16, Ay)) i konfiguruje wskazany rejestr Drc (Debug Control Register). Instrukcja synchronizuje potok. Instrukcja musi zostać wyrównana do granicy 4 bajtów (4 słowa). Umożliwia to wykonanie rozkazów modułu BDM bezpośrednio z procesora ColdFire.

Debug Command Organization in Memory:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	0	0	1	0	0	DRc				
Data[31:16]															
Data[15:0]															
Unused															

Sterujące rejestry trybu debugowania

DRc[4–0]	Register Name
0x00	Configuration/status register
0x01–0x0	Reserved
0x04	PC breakpoint ASID control
0x05	BDM address attribute register
0x06	Address attribute trigger register
0x07	Trigger definition register
0x08	Program counter breakpoint register
0x09	Program counter breakpoint mask register
0x0A–0x0B	Reserved
0x0C	Address breakpoint high register
0x0D	Address breakpoint low register
0x0E	Data breakpoint register
0x0F	Data breakpoint mask register

DRc[4–0]	Register Name
0x10–0x1	Reserved
0x14	PC breakpoint ASID register
0x15	Reserved
0x16	Address attribute trigger register 1
0x17	Extended trigger definition register
0x18	Program counter breakpoint 1 register
0x19	Reserved
0x1A	Program counter breakpoint register 2
0x1B	Program counter breakpoint register 3
0x1C	Address high breakpoint register 1
0x1D	Address low breakpoint register 1
0x1E	Data breakpoint register 1
0x1F	Data breakpoint mask register 1