iniDSP

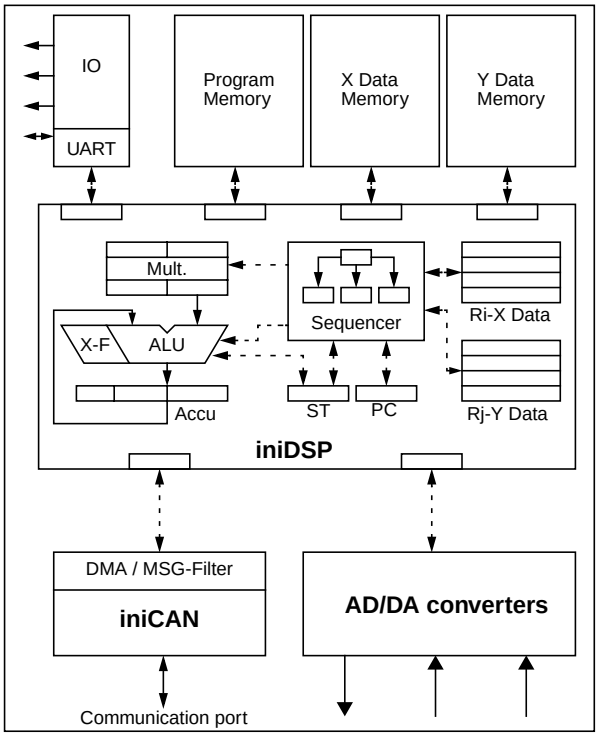
The fastest path for embedded DSP solutions

data sheet

Features:

- 16 bit fixed point customizable DSP
- Single cycle 16 bit signed/unsigned multiplier
- 1 or more 40bit accumulator(s)
- 64k X data, Y data and program memory range
- Supports pseudo floating point arithmetic
- 32bit barrel shifter
- Low latency interrupts and sleep mode
- Powerful built-in hardware debugger
- C-compiler, assembler, linker and debugger
- Straight forward system embedding
- Fully synchronous, technology independent
- VHDL model for synthesis and simulation
- Open and customizable architecture, instruction plug-in: iniDSP is fast where you need the power!
- **Lowest Power Consumption**: < 100uA / MIPS
- **80MIPS** on 0.18 micron ASIC proven
- Evaluation platform available (FPGA and ASIC)

Signal conditioning sensor with iniDSP:



INICORE - the reliable Core and System Provider. We provide high quality IP, design expertise and leading edge silicon to the industry.

INICORE delivers highly specialized IP and leading edge silicon to the industry. Our **iniDSP** family offers DSP power for innovative System-on-Chip solutions.

Low voltage - lowest power solutions, optimized for your specific algorithms, open new application fields and push the limits of today's standard parts. *You* define package, memory and periphery to fulfil the demands of innovative products.

Straight forward implementation techniques sqeeze down schedules. Avoid long and costly timing verification of technology dependent full custom solutions! The application proven bus interface, the synchronous clocking structure and a comprehensive, transparent architecture reduce decision time, evaluation and design-in efforts.

The smart, low overhead hardware debugger circuit gives you complete control over the iniDSP system for **on-chip real time debugging**.

Product family development, follow up products and second source is guaranteed by **INICORE's** design methodology and the consequent high level approach. The wide application range goes from **lowest power solutions** like hearing aids, signal conditioning in sensors, over mid range audio and coding applications

US Sales Office:
INICORE INC.
5600 Mowry School Road, Suite 180,
Newark, CA 94560
Tel: 510 445 1529 Fax: 510 656 0995
E-mail: ask_us@inicore.com
Web: www.inicore.com

INICORE AG
Mattenstrasse 6a,CH-2555 Brügg, Switzerland
Tel: ++41 32 374 32 00, Fax: ++41 32 374 32 01
E-mail: ask_us@inicore.ch
Web: www.inicore.ch

Ref.Nr.: 441-DS-10 / 6/99

1 Introduction

This document gives a short overview of the iniDSP concept. The concept targets low power and fast time to market support. The design is made for a straight forward high level design approach on modern integration technologies. This approach enables DSP embedding for system on silicon solutions without excessive engineering power, fast time to production and a fully technology independent approach. You can follow a standard design flow, giving you access to the most advanced ASIC technologies. There is no dealing with masks, there is no special effort needed to migrate your design to denser technologies!

1.1 Micro Power Aspects

The iniDSP concept is based on a micro power approach, which allows digital signal processing on battery operated applications. The low power aspect is reached by a low gate count and by the concept of a gated balanced clock tree, which allows a fast layout without excessive timing verification. This approach helps to bring down the toggle rate of rarely used resources. For the main data path in the ALU etc. where the data path is used for almost every instruction, the low power approach is handled by not changing unnecessarily control signals. This reduces heavily toggling rate in the data path.

1.2 High Performance Aspects

By keeping critical paths short, high speed integration can be achieved that result in high performance systems. Additionally, most of the instructions are executed in one cycle, where branches and interrupt handling need maximum two. The programmer does never have to take care of the pipelined architecture, compare / branch can be coded straight forward. iniDSP has a built-in interface to custom co-processors and custom instructions, which can be used to enhance the system performance for a specific application. This strategy makes iniDSP fast for your application, without the cost of an overall increased performance. Our software tools support a customized instruction set to have optimal integration support for them.

1.3 Fast time to production

The structured design approach (SD) and high level RTL implementation techniques lead to very short implementation time. This technology independent and synthesisable implementation guarantees optimal reusability, where follow-up products can be realized within a very short time. By being able to choose the best technology at a late stage of the ASIC integration, systems can be optimized in respect to cost, power consumption or performance. The technology independent solution allows to built fast prototypes using FPGA technology, where customized parts can be verified efficiently. The same database is used later for the ASIC integration, resulting in a reliable test environment that gives you confidence in the performed tests on the FPGA. Software development can be started a long time before you see the first ASIC prototype!

1.4 Smooth System Integration

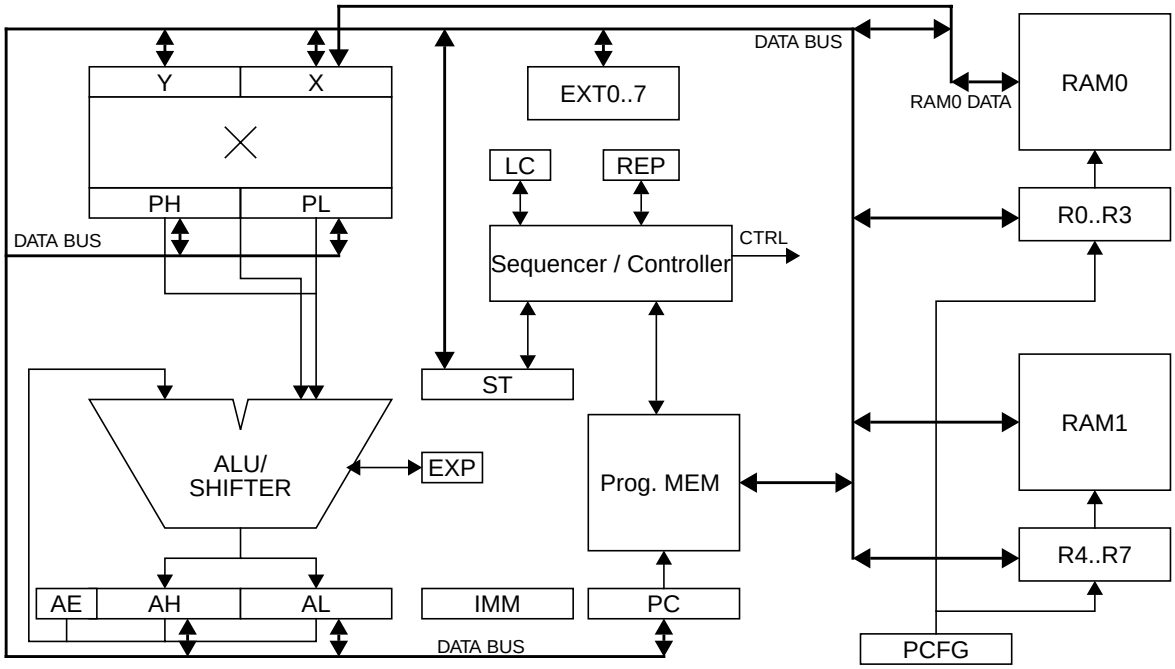
iniDSP is targeted for high level system embedding. To achieve a smooth system integration, we designed simple but smart interfaces towards the main system. A technology independent design guarantees easy system integration because all synthesis oriented design flows are supported. In addition, you can use the same database for simulation and synthesis. There is only one iniDSP reference, eliminating the risk of faulty simulation models.

1.5 Debugging features

iniDSP supports powerful hardware debugging concepts, where transparency is given through access to all registers, memories and single stepping. Permanent and dynamic break points help you to debug complex software at full speed! The debugger can be connected by a standard serial or any other interface.

2 General Architecture

The iniDSP is a pipelined single accumulator architecture with an explicit harvard structure. For most of the instructions, the CPI[1] is 1, for branches and some others, CPI is maximum 2. The following pictures shows the architecture:



2.1 ALU

2.1.1 Arithmetic Unit

The arithmetic path of the ALU has a total width of 40 bits, formed by the accu high (AH), accu low (AL) and the extension bits (AE). The main accu is formed by AE | AH (24 bits), and AL is included in the datapath for multiply-accumulate and other multiple precision operations. Remark that a 40 bit addition can be performed in one cycle without pipelining. The input paths are the 40 bit accumulator, the 32bit multiplier output (PH,PL) and the 16 bit data bus. All data is sign extended to 24 resp. 40 bits.

2.1.2 Logic Unit

The logic unit is 16bit wide and performs OR, AND, XOR and bit operations like SET, RES, TGL and TST.

2.1.3 Barrel Shifter

A barrel shifter operating on the full accumulator (AE|AH|AL) with a range of up to 31 bits left and 31 bits right shift allows powerful shifting. The shift amount and direction can be controlled by the instruction or the exponent register (EXP).

2.1.4 Floating Point Support

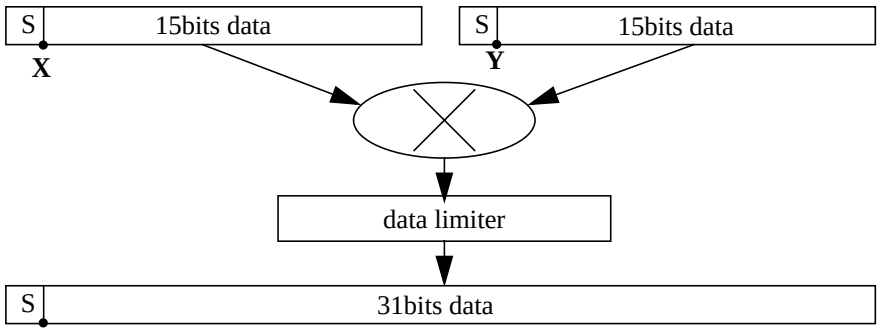
The CLB (count leading bits) instruction allows detecting the most significant bit in the 40bit accumulator and stores the result in the EXP register. Together with the barrel shifter, a floating point representation of a 40 bit number can be performed in 2 cycles.

1. Clock Per Instruction

2.2 Multiplier

The single cycle multiplier can perform multiplications from the input registers X and Y. Signed and unsigned data format is supported. A special feature of the iniDSP is the fact that the multiplication result can be directly stored into the accumulator (sign extended to 40 bits), eliminating additional copy cycles form the normal destination (PH|PL) in isolated multiplication instructions.

Data representation for *signed* multiplication is using 2's complement numbers with a virtual data range of -1 .. +~1[1] for the X and Y register as well as for the result in PH|PL. The special case for (X=-1=0x8000 by Y=-1=0x8000) results in PH|PL = +~1=0x7fff'ffff. The data sheme is shown in the following figure:

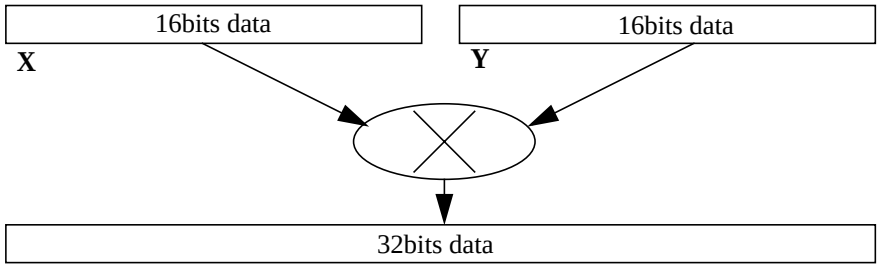


PH|PL

S = sign bit

• = virtual decimal point

For *unsigend* multiplication, both X and Y are using absolute data format, leading in a 32 bit result in PH|PL. Remark that the user has to handle the virtual data format himself in this case! The 'special' case of (X=0xffff by Y=0xffff) results in PH|PL=0xfffe0001. The



PH|PL

result is always positive, so no sign extension is done in case of accumulator as write destination (AE = 0).

1. +~1 means "nearly" 1, which is (1 - 2^-15) for 16bit values, (1 - 2^-31) for 32bit values.

2.3 Memory and pointers

2.3.1 Data Memories (RAM0, RAM1)

There are 2 identical data memories available, each with a maximum size of 64k words. Each data memory can be accessed by 4 address pointers, where 3 are general purpose and 1 is predefined as stack. The general purpose pointers (R0..R2, R4..R6) can individually be configured for different loop size and increment/decrement. Addressing modes are nop, post-increment, post-decrement and signed offset. For stack access, R3 and R7 can be accessed with pre-decrement without additional delay.

2.3.2 Program Memory (PRAM)

The program memory can be addressed by the PC, or any of the 8 address pointers. Due to the permanent access for instruction fetch, each data access to the program memory takes 2 cycles.

2.4 Repeat and Loop Support

For filter implementations, a repeat instruction (REP) allows significant reduction in code size and power. A hardware loop counter (LOOP), together with the “decrement and branch if not zero” instruction enables efficient implementation of larger loops.

2.5 External resources

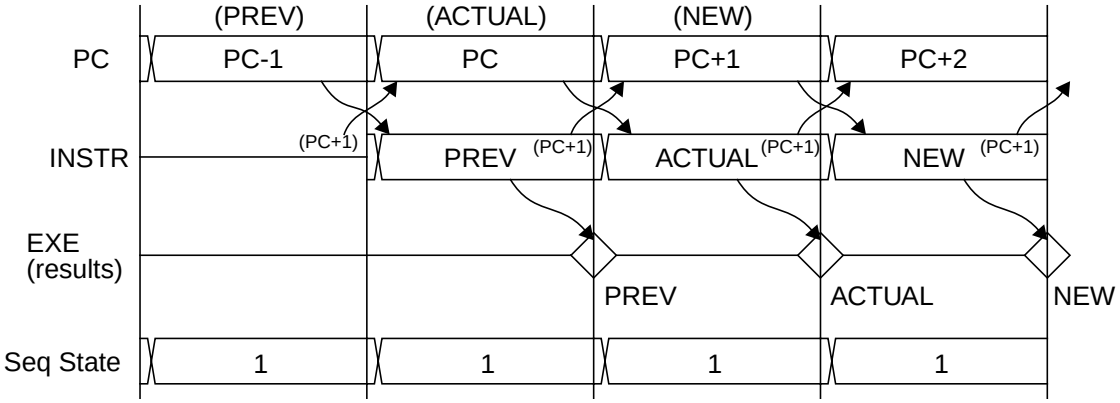
For external resources like peripherals, interrupt controller, etc. or for additional general purpose registers, there are 4 external registers available, each 16bits wide. They can be implemented internally or externally, or replaced by the registers of peripherals like UARTs etc.

2.6 Interrupts

The interrupt structure is designed for low latency, low effort processing of a large number of interrupt sources. Although the iniDSP provides only one interrupt line, including enable flag an external interrupt controller provides up to 15 interrupt lines with individual vectors and autonomous flag handling. This minimizes the overhead of interrupt processing. Interrupts need 2 cycles for vector fetch and 2 cycles for return from interrupt. The repeat instruction cannot be broken by interrupts.

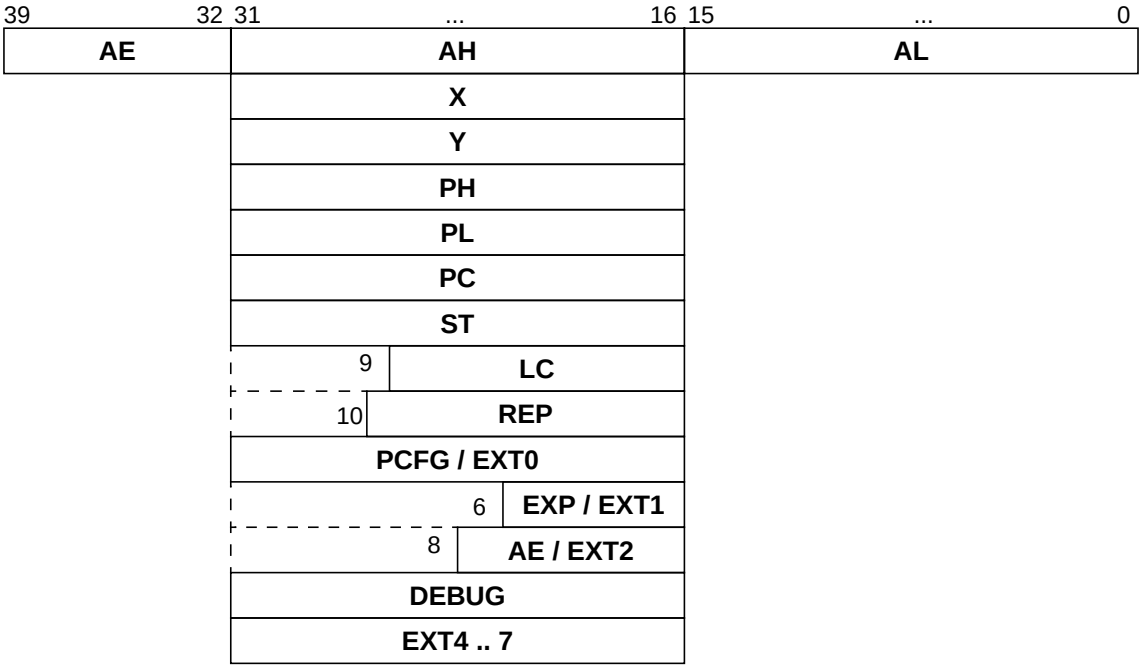
2.7 Sequencer

The sequential execution of instructions is partitioned in two steps, fetch / decode and execution / write back, using a 1 stage pipeline. Remark that there is no pipeline delay for compare/branch instruction flows!



3 Programming Model

This chapter describes the iniDSP programming model, internal registers etc. The following table shows the register available in the iniDSP.



3.1 Accumulator

The accumulator of the iniDSP has totally 40 bits, partitioned in AE, AH and AL. It is the target register for math and logic operations and may also be target for multiplication results. The three registers have following functions:

AE: Extension or guard bits. Used for compensating a temporary overflow. AE is always written/modified when AH is written. In case of load instructions, AE is written as sign extension of AH. AE can be read and written as sign extended value on the EXT2 register(15:0). The significant bits are located in (7:0), where bits (15:8) are sign extended.

AH: High accumulator. This is the main part of the accumulator. All logical and math operations results are stored into this register.

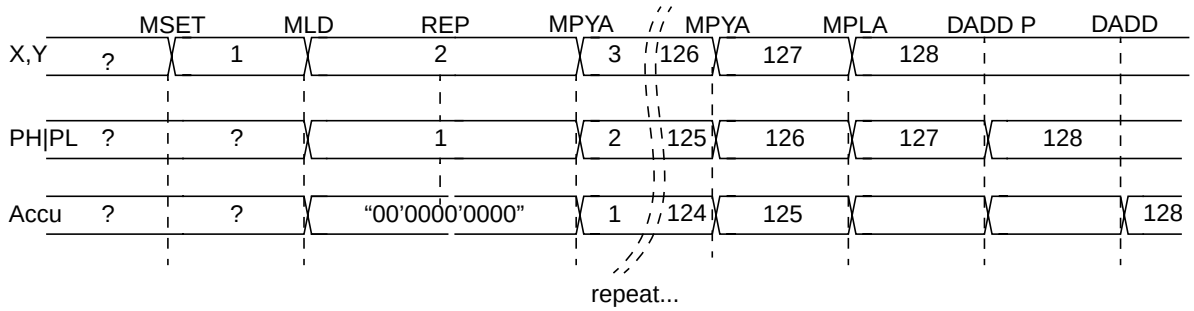
AL: Low accumulator. Used together with all double precision math operations, like MPLA, MPLS, DADD, DSUB etc. Can be used as universal data register.

3.2 MAC Unit

Multiply-accumulate instructions are executed using the multiplier and the ALU in a pipelined way. To implement a MAC stream (e.g. FIR filter), the following sequence may be used:

```
MSET   (Rj+), (Ri+)   #set up first X,Y values
MLD    (Rj+), (Ri+)   #load new X,Y, multiply old X,Y, Accu=0
REP 127               #repeat n times (or place a few MPYA's)
MPYA   (Rj+), (Ri+)   #do the multiply-accumulations
DADD P                #calculate the last product and add A
DADD                  #add the last product to A
```

For the timings, see the following diagram:



3.3 Registers

The iniDSP has a set of 16bit data registers, with dedicated functions:

X: The multiplier input register 1, signed or unsigned, triggers the multiplier

Y: The multiplier input register 2, signed or unsigned, triggers also the multiplier

PH: The multiplier high output register, signed or unsigned

PL: The multiplier low output register, unsigned

PC: The program counter

LC: The 9bit loop counter register, unsigned, hidden register

REP: The 10bit repeat counter register, unsigned, hidden register.

EXT0 .. EXT7: A set of general purpose data registers with application dependent functionality:

Name	Access	Size	Functionality
PCFG (or EXT0)	R/W	16	Pointer configuration for R0..R7, controlled by A0..A2 of ST register
EXP (or EXT1)	R/W	6/16	Exponent register for ‘CLB A’ and ‘SHFT A, EXP’ instruction. When EXP is read or written, it is sign extended to 16 bits
AE (or EXT2)	R/W	8/16	Guard bits, extension of accumulator. When EXP is read or written, it is sign extended to 16 bits
DEBUG (or EXT3)	not recommended	16	Debug register. It is not recommended to use this register (restricted debugging features)
EXT4	user defined		can be used for high performance peripheral units, auxiliary registers, interrupt controller etc.
EXT5	user defined		
EXT6	user defined		
EXT7	user defined		

3.4 Status Register and Flags

The iniDSP has a set of standard and special flags for extended usage. Flags are only affected in logical and math operations. The status register looks like this:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
N	V	Z	C	E	L	RiC	OP	-	IE	-	SM	-	A2	A1	A0

N: Negative flag; valid after math operations, equals bit7 of AE (sign bit)

V: Overflow flag; sticky bit, set after math operations, when the an overflow occurred on the extension bits (AE) and the result in the accumulator left the valid range. It has to be cleared by writing a ‘0’ to the ST register.

Z: Zero flag;
Logical operations: Set if AH = 0, cleared otherwise.
Math operations: Set if AE|AL = 0, cleared otherwise.

C: Carry flag; the carry out of AE is valid after math operations

E: Extension flag; valid after math operations, when the AE register is in use.

L: Limiter flag; sticky bit, set after write operations with AH as source, when a clipping occurred (AE is in use, AH is set to positive or negative maximum). L is not affected in logical operations.

RiC: The pseudo carry flag for the address pointers. Set when a pointer wraps around the loop. When both pointer groups are used, the RiC is the ored value of the Ri and Rj pointer carry flag.

OP: Overflow protect: when set, values in the accumulator (AE|AH) exceeding the range of +1/-1 are fixed to the maximum value (when AH is written to any destination).

IE: The interrupt enable flag:
0: Interrupt disabled (after interrupt acknowledge and reset)
1: Interrupt enabled

SM: The multiplier sign mode:
‘0’ : multiply signed by signed (sign extension on PH|PL to 40bits)
‘1’ : multiply unsigned by unsigned (complete with zeros PH|PL to 40bits)

A2, A1, A0: The pointer configure address pointers, see table in 3.5.3.

3.5 Address Pointers

This table shows the functionality of the pointer modifications and the possible loop ranges etc.

3.5.1 Pointer use

Pointers can be modified in the same instruction as they are used. The modification is always done after usage, except for the stack registers R3 and R7, where pre-increment addressing is used to handle the stack correctly.
When a pointer wraps around (pos or neg), the RiC flag in the ST is set. If both pointers Ri and Rj are modified, RiC takes the ored value from Ri. and Rj.

The following table gives an overview of the possible addressing modes:

(Rij)	Binary code	Operation in linear mode	Operation in power of 2 mode
(Rij)	“00”	no modification	no modification
(Rij+)	“01”	Rij = (Rij + 1) MOD LoopSize	Rij = (Rij + 1) MOD LoopSize
(Rij-)	“10”	Rij = (Rij - 1) MOD LoopSize	Rij = (Rij - 1) MOD LoopSize
(Rij+!)	“11”	not allowed	Rij = Rij + **I**, MOD LoopSize

3.5.2 Pointer loop definition

Each pointer can be programmed individually, except R3 and R7 which work always as stack pointers (pre-increment and post decrement by 1). The config word takes 16 bits:

LoopMode	Description
Power of 2 Mode with programmable increment / decrement value.	**“SSSS”:** The size field of the loop: A 4bit binary number which defines the loop size (and the base). The pointer will loop at 2^{SSSS}. The range is “0001” to “1011” which results in 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 and 2048 word loops. The loop base is defined by the remaining most significant bits of the pointer remain stable. The value “0000” is used to disable the loop function (LoopSize = 65536) **“M”:** The mode flag defines the loop mode. In power of 2 mode, it must be ‘0’. **“IIIIIIIIII”:** The increment field defines the increment of the pointer modification. It is a signed number with a range of -1024 to +1023. This value does not have to be a divisor of the loop base.
Linear Mode with fixed increment / decrement by 1	**“SSSS”:** The base field of the loop: A 4bit binary number which defines the loop base. This field defines the base of the linear loop by 2^{SSSS}. The range is “0001” to “1011” which results in a base of 2,4,8,16,32,64,128,256,512, 1024 and 2048. The value “0000” is used to disable the loop function (LoopSize = 65536). This field has to be accorded with the loop size field. **“M”:** The mode flag defines the loop mode. In linear mode, it must be ‘1’. **“IIIIIIIIII”:** The loop size field. The loop size is defined by this unsigned binary number and has a range of 1 to 2048. This value must be accorded for correct operation with the base field (SSSS).

“SSSSMIIIIIIIIII”

3.5.3 Configure pointers

To configure the pointers, external registers (e.g. EXT0) and the A(2:0) bits in the ST register are used. A2, A1, A0 in the ST register select the destination when the EXT0 register

is written. To configure a pointer, select first the address (see table below)., then write the pointer configuration into EXT0.

A2	A1	A0	Configure pointer
0	0	0	R0
0	0	1	R1
0	1	0	R2
0	1	1	R3 is fixed as stack pointer
1	0	0	R4
1	0	1	R5
1	1	0	R6
1	1	1	R7 is fixed as stack pointer

3.5.4 Pointer examples

Pointer example for power of 2 mode: (M = 0) "0011000000000011"

Loop Size = "0011" = 2^3 = 8(dec)
Loop increment = "00000000011" = 3(dec)

The pointer wraps around every 2^3 boundary. The base is defined by the pointer itself with its bits (15:3). The remaining pointer bits (2:0) will loop in the following manner:

(e.g. (R2+)): R2 = R2 + ((R2 + 3) MOD 8) : R2 = 0, 3, 6, 1, 4, 7, 2, 5, 0, etc.

Pointer example for linear mode: (M = 1) "0100100000001100"

Loop Base = "0100" = 2^4 = 16(dec)
Loop size+1 = "0000001100" = 13(dec)
Increment is always +/- 1

The pointer base may be placed in every 2^4 boundary. The base is defined by the pointer itself with its bits (15:4). The remaining pointer bits (3:0) will loop in the following manner:

(e.g. (R4+)): R4 = R4 + ((R4 + 1) MOD 13) :
R4 = 0,1,2,3,4,5,6,7,8,9,10,11,12,0,1,2, etc.

Remark: To set a base in the linear mode is needed to enable different loops in the same memory block. The selected base must always be larger than the linear loop size. If this is not the case, the pointer behavior will be undefined.

4 Instruction set

The instruction set is summarized in the following table, grouped by the instruction word. [nn:mm] show the relevant bits of the instruction word.

	[15:13]							
[12:9]	000	001	010	011	100	101	110	111
0000	LD Reg, Reg	SUB A, Reg	LDLC SIMM	CMP A, Reg	ADD A, Reg	AND A, Reg	OR A, Reg	EOR A, Reg
0001	LD Reg, (Rij)	SUB A, (Rij)	LD (Rij), Reg	CMP A, (Rij)	ADD A, (Rij)	AND A, (Rij)	OR A, (Rij)	EOR A, (Rij)
0010	SHFT n	MPYS Reg, (Ri)	MSET Reg, (Ri)		MPYA Reg, (Ri)	MLD Reg, (Ri)		
0011	LD A, DRAM	SUB A, DRAM	LD DRAM, A	CMP A, DRAM	ADD A, DRAM	AND A, DRAM	OR A, DRAM	EOR A, DRAM
0100	LDI Rij, IMM LDI Reg, IMM	SUBI A, IMM	CALL PRAM, Cond	CMPI A, IMM	ADDI A, IMM	ANDI A, IMM	ORI A, IMM	EORI A, IMM
0101	LD Reg, (Rij)p	SUB A, (Rij)p	LD (Rij)p, Reg	CMP A, (Rij)p	ADD A, (Rij)p	AND A, (Rij)p	OR A, (Rij)p	EOR A, (Rij)p
0110	LDI (Rij), IMM	DEC (Rij), Cond	BRA PRAM,Cond 2)		INC (Rij), Cond	RES (Rij), Bit	SET (Rij), Bit	TGL (Rij), Bit
0111		DEC Reg, Cond	MUL A,P,Cond	MOD OpA, Cond 1)	INC Reg, Cond	RES Reg, Bit	SET Reg, Bit	TGL Reg, Bit
1000	SHFT A,INV, Cond		CLB A		RND A	TST (Rij), Bit		
1001	LD Reg, Rij	SUB A, Rij	LD Rij, Reg	CMP A, Rij	ADD A, Rij	AND A, Rij	OR A, Rij	EOR A, Rij
1010	CLR Reg	DSUB P, Cond	SLEEP IE SET_IE	DCMP	DADD P, Cond	TST Reg, Bit		
1011	MODR Rj, Ri	MPYS (Rj), (Ri)	MSET (Rj), (Ri)		MPYA (Rj), (Ri)	MLD (Rj), (Ri)		
1100	LDSI A, SIMM	SUSI A, SIMM	REP n	CMSI A, SIMM	ADSI A, SIMM	ANSI A, SIMM	ORSI A, SIMM	EOSI A, SIMM
1101								
1110	LDSI R0, SIMM	LDSI R1, SIMM	LDSI R2, SIMM	LDSI R3, SIMM	LDSI R4, SIMM	LDSI R5, SIMM	LDSI R6, SIMM	LDSI R7, SIMM
1111								

1) MOD OpA, Cond can be:
- SWAP A, Cond
- NEG A, Cond
- ABS A, Cond
- CMPL A, Cond

2) BRA PRAM, Cond can be:
- BRA PRAM, Cond
- DBNZ PRAM

Remark that the assembler supports aliases for PUSH, PULL, etc. in order to make the code more readable.

4.1 Alphabetical list of instructions:

Instruction	Instruction code	#	C	Operation	N	V	Z	C	E	L	R	Comment
ABS A, Cond, DW	0110111M1111cccc	1	2	A = ABS(A)	↕	↗	↕	❍	↕	❍	❍	DW = 1: incl. AL
ADD A, Reg, C	10000000C000rrrr	1	1	A = A + Reg + C	↕	↗	↕	↕	↕	↗	❍	Input is sign ext.
ADD A, (Rij), C	1000001BC000PPPP	1	1	A = A + (Rij) + C	↕	↗	↕	↕	↕	❍	↕	Input is sign ext.
ADD A, DRAM, B	1000011BDDDDDDDD	1	1	A = A + (DRAM)	↕	↗	↕	↕	↕	❍	❍	Input is sign ext.
ADD A, (Rij)p, C	1000101BC000PPPP	2	3	A = A + (Rij)p + C	↕	↗	↕	↕	↕	❍	↕	Input is sign ext.
ADD A, Rij, C	1001001BC00000pP	1	1	A = A + Rij + C	↕	↗	↕	↕	↕	❍	❍	Input is sign ext.
ADDI A, IMM, C	10001000C1110000 IIIIIIIIIIIIIIII	2	5	A = A + IMM + C	↕	↗	↕	↕	↕	❍	❍	Input is sign ext.
ADSI A, SIMM	100110SSSSSSSSSS	1	1	A = A + SIMM	↕	↗	↕	↕	↕	❍	❍	Input is sign ext.
AND A, Reg	1010000000000rrrr	1	1	AH = AH and Reg	❍	❍	↕	❍	↕	↗	❍	
AND A, (Rij)	1010001B0000PPPP	1	1	AH = AH and (Rij)	❍	❍	↕	❍	↕	❍	↕	
AND A, DRAM, B	1010011BDDDDDDDD	1	1	AH = AH and (DRAM)	❍	❍	↕	❍	↕	❍	❍	
AND A, (Rij)p	1010101B0000PPPP	2	3	AH = AH and (Rij)p	❍	❍	↕	❍	↕	❍	↕	
AND A, Rij	1011001B000000pP	1	1	AH = AH and Rij	❍	❍	↕	❍	↕	❍	❍	
ANDI A, IMM	1010100000000000 IIIIIIIIIIIIIIII	2	5	AH = AH and IMM	❍	❍	↕	❍	↕	❍	❍	
ANSI A, SIMM	101110SSSSSSSSSS	1	1	AH = AH and SIMM	❍	❍	↕	❍	↕	❍	❍	
BRA PRAM, Cond	010011000000cccc IIIIIIIIIIIIIIII	2	7	IF Cond = True: PC = IMM	❍	❍	❍	❍	❍	❍	❍	
BREAK	0000000000100010	1	1	Stop the DSP (debug)	❍	❍	❍	❍	❍	❍	❍	Wake up by debug
CALL PRAM, Cond	0100100B0111cccc	2	8	If Cond = True: PC = IMM Old PC = (R3/R7-)	❍	❍	❍	❍	❍	❍	↕	ST is not saved
CLB A	0101000000000000	1	1	EXP = Pos significant bit A	↕	❍	↕	❍	❍	❍	❍	A = AE|AH|AL
CLR A, DW	0001010M00010000	1	1	A = 0	❍	❍	❍	❍	❍	❍	❍	
CLR Reg	00010100RRRR0000	1	1/4	Reg = 0	❍	❍	❍	❍	❍	❍	❍	
CMP A, Reg, C	01100000C000rrrr	1	1	A - Reg - C	↕	↗	↕	↕	❍	↗	❍	Input is sign ext.
CMP A, (Rij), C	0110001BC000PPPP	1	1	A - (Rij) - C	↕	↗	↕	↕	❍	❍	↕	Input is sign ext.
CMP A, DRAM, B	0110011BDDDDDDDD	1	1	A - (DRAM)	↕	↗	↕	↕	❍	❍	❍	Input is sign ext.
CMP A, (Rij)p, C	0110101BC000PPPP	2	3	A - (Rij)p - C	↕	↗	↕	↕	❍	❍	↕	Input is sign ext.
CMP A, Rij, C	0111001BC00000pP	1	1	A - Rij - C	↕	↗	↕	↕	❍	❍	❍	Input is sign ext.
CMPI A, IMM, C	01101000C1110000 IIIIIIIIIIIIIIII	2	5	A - IMM - C	↕	↗	↕	↕	❍	❍	❍	Input is sign ext.
CMSI A, SIMM	011110SSSSSSSSSS	1	1	A - SIMM	↕	↗	↕	↕	❍	❍	❍	Input is sign ext.
CMPL A, Cond, DW	0110111M0110cccc	1	2	A = NOT(A)	↕	↗	↕	❍	↕	❍	❍	DW = 1: incl. AL
DADD P, Cond	1001010000P0cccc	1	1	A = A + PH|PL, write X*Y	↕	↗	↕	↕	↕	❍	❍	sign ext dep. on SM
DBNZ PRAM	0100110100000000 IIIIIIIIIIIIIIII	2	7	LC = LC - 1 PC = IMM if LC(old) = 0	❍	❍	❍	❍	❍	❍	❍	
DCMP	0111010000000000	1	1	A - PH|PL	↕	↗	↕	↕	↕	❍	❍	sign ext dep. on SM
DEC A, Cond, DW	0010111M0001cccc	1	2	A = A - 1	↕	↗	↕	↕	↕	❍	❍	DW = 1: incl. AL
DEC Reg, Cond	00101110RRRRcccc	2	6	Reg = Reg - 1	↕	↗	↕	↕	↕	❍	❍	Reg /= AH
DEC (Rij), Cond	0010110BPPPPcccc	2	6	(Rij) = (Rij) -1	↕	↗	↕	↕	❍	❍	↕	
DSUB P, Cond	0011010000P0cccc	1	1	A = A - PH|PL, write X*Y	↕	↗	↕	↕	↕	❍	❍	sign ext dep. on SM
EOR A, Reg	1110000000000rrrr	1	1	AH = AH eor Reg	❍	❍	↕	❍	↕	↗	❍	
EOR A, (Rij)	1110001B0000PPPP	1	1	AH = AH eor (Rij)	❍	❍	↕	❍	↕	❍	↕	
EOR A, DRAM, B	1110011BDDDDDDDD	1	1	AH = AH eor (DRAM)	❍	❍	↕	❍	↕	❍	❍	

Instruction	Instruction code	#	C	Operation	N	V	Z	C	E	L	R	Comment
EOR A, (Rij)p	1110101B0000PPPP	2	3	AH = AH eor (Rij)p	❍	❍	↕	❍	↕	❍	↕	
EOR A, Rij	1111001B000000pP	1	1	AH = AH eor Rij	❍	❍	↕	❍	↕	❍	❍	
EORI A, IMM	1110100000000000 IIIIIIIIIIIIIIII	2	5	AH = AH eor IMM	❍	❍	↕	❍	↕	❍	❍	
EOSI A, SIMM	111110SSSSSSSSSS	1	1	AH = AH eor SIMM	❍	❍	↕	❍	↕	❍	❍	
INC A, Cond, DW	100011100001cccc	1	2	A = A + 1	↕	↗	↕	↕	↕	❍	❍	DW = 1: incl. AL
INC Reg, Cond	10001110RRRRcccc	2	6	Reg = Reg + 1	↕	↗	↕	↕	↕	❍	❍	Reg /= AH
INC (Rij), Cond	1000110BPPPPcccc	2	6	(Rij) = (Rij) +1	↕	↗	↕	↕	❍	❍	↕	
LDLC SIMM	0100000SSSSSSSSS	1	1	LC = SIMM (9bit)	❍	❍	❍	❍	❍	❍	❍	
LD Reg, Reg	00000000RRRRrrrr	1†	1/4	Dest Reg = Source Reg	❍	❍	❍	❍	↕	↗	❍	Input is sign ext. (A)
LD Reg, (Rij)	0000001BRRRRPPPP	1†	1/4	Reg = (Rij)	❍	❍	❍	❍	↕	❍	↕	Input is sign ext. (A)
LD (Rij), Reg	0100001BrrrrPPPP	1	1	(Rij) = Reg	❍	❍	❍	❍	❍	↗	↕	
LD A, DRAM, B	0000011BDDDDDDDD	1	1	A = (DRAM)	❍	❍	❍	❍	❍	❍	❍	Input is sign ext.
LD DRAM, A, B	0100011BDDDDDDDD	1	1	(DRAM) = A	❍	❍	❍	❍	❍	↗	❍	Input is sign ext.
LD Reg, (Rij)p	0000101BRRRRPPPP	2	3/4	Reg = (Rij)p	❍	❍	❍	❍	❍	❍	↕	Input is sign ext. (A)
LD (Rij)p, Reg	0100101BrrrrPPPP	2	3	(Rij)p = Reg	❍	❍	❍	❍	❍	↗	↕	
LD Reg, Rij	0001001BRRRR00pP	1†	1/4	Reg = Rij	❍	❍	❍	❍	↕	❍	❍	Input is sign ext. (A)
LD Rij, Reg	0101001Brrrr00pP	1	1	Rij = Reg	❍	❍	❍	❍	❍	↗	❍	
LDI Reg, IMM	00001000RRRR0000 IIIIIIIIIIIIIIII	2	4/5	Reg = IMM	❍	❍	❍	❍	↕	❍	❍	Input is sign ext. (A)
LDI Rij, IMM	0000100B000010pP IIIIIIIIIIIIIIII	2	5	Rij = IMM	❍	❍	❍	❍	❍	❍	❍	
LDI (Rij), IMM	0000110B0000PPPP IIIIIIIIIIIIIIII	2	5	(Rij) = IMM	❍	❍	❍	❍	❍	❍	↕	
LDSI A, SIMM	000110SSSSSSSSSS	1	1	A = SIMM	❍	❍	❍	❍	❍	❍	❍	SIMM is signed
LDSI Rij, SIMM	BpP111SSSSSSSSSS	1	1	Rij = SIMM	❍	❍	❍	❍	❍	❍	❍	SIMM is unsigned
MLD (Rj), (Ri), SQ	1011011SPPPPpppp	1	9	X = (Ri) Y = (Rj) {S=0}; (Ri) {S=1} AE, AH, AL = 0 C = 0 update PH|PL	0	❍	1	0	0	❍	↕	SQ = 1: square option
MLD Reg, (Ri), SQ	1010010Srrrrpppp	1	9	X = (Ri) {S=0}; Reg {S=1} Y = Reg AE, AH, AL = 0 C = 0	0	❍	1	0	0	❍	↕	SQ = 1: square option
MPYA (Rj), (Ri), SQ	1001011SPPPPpppp	1	9	X = (Ri) Y = (Rj) {S=0}; (Ri) {S=1} A = A + PH|PL	↕	↗	↕	↕	↕	❍	↕	sign ext dep. on SM SQ = 1: square option
MPYA Reg, (Ri), SQ	1000010Srrrrpppp	1	9	X = (Ri) {S=0}; Reg {S=1} Y = Reg A = A + PH|PL	↕	↗	↕	↕	↕	❍	↕	sign ext dep. on SM SQ = 1: square option
MPYS (Rj), (Ri), SQ	0011011SPPPPpppp	1	9	X = (Ri) Y = (Rj) {S=0}; (Ri) {S=1} A = A - PH|PL	↕	↗	↕	↕	↕	❍	↕	sign ext dep. on SM SQ = 1: square option
MPYS Reg, (Ri), SQ	0010010Srrrrpppp	1	9	X = (Ri) {S=0}; Reg {S=1} Y = Reg A = A - PH|PL	↕	↗	↕	↕	↕	❍	↕	sign ext dep. on SM SQ = 1: square option
MSET (Rj), (Ri), SQ	0101011SPPPPpppp	1	9	X = (Ri) Y = (Rj) {S=0}; (Ri) {S=1}	❍	❍	❍	❍	❍	❍	↕	SQ = 1: square option
MSET Reg, (Ri), SQ	0100010Srrrrpppp	1	9	X = (Ri) {S=0}; Reg {S=1} Y = Reg	❍	❍	❍	❍	❍	❍	↕	SQ = 1: square option
MODR Rj, Ri	00010110PPPPpppp	1	1	Ri = Ri mod; Rj = Rj mod	❍	❍	❍	❍	❍	❍	↕	

Instruction	Instruction code	#	C	Operation	N	V	Z	C	E	L	R	Comment
MUL A,P,Cond	0100111000Pacccc	1	1	write A, P with X*Y	❍	❍	❍	❍	↕	❍	❍	sign ext dep. on SM
NEG A, Cond, DW	0110111M1110cccc	1	2	A = NEG(A)	↕	↗	↕	❍	↕	❍	❍	DW = 1: incl. AL
OR A, Reg	110000000000rrrr	1	1	AH = AH or Reg	❍	❍	↕	❍	↕	↗	❍	
OR A, (Rij)	1100001B0000PPPP	1	1	AH = AH or (Rij)	❍	❍	↕	❍	↕	❍	↕	
OR A, DRAM, B	1100011BDDDDDDDD	1	1	AH = AH or (DRAM)	❍	❍	↕	❍	↕	❍	❍	
OR A, (Rij)p	1100101B0000PPPP	2	3	AH = AH or (Rij)p	❍	❍	↕	❍	↕	❍	↕	
OR A, Rij	1101001B000000pP	1	1	AH = AH or Rij	❍	❍	↕	❍	↕	❍	❍	
ORI A, IMM	1100100000000000 IIIIIIIIIIIIIIII	2	5	AH = AH or IMM	❍	❍	↕	❍	↕	❍	❍	
ORSI A, SIMM	110110SSSSSSSSSS	1	1	AH = AH or SIMM	❍	❍	↕	❍	↕	❍	❍	
REP n	010110SSSSSSSSSS	1	10	Repeat next instr. SIMM x	❍	❍	❍	❍	❍	❍	❍	
RES AH, Bit	101011100001bbbb	1	1	Bit of AH = 0	❍	❍	↕	❍	↕	❍	❍	
RES Reg, Bit	10101110RRRRbbbb	2	6	Bit of Reg = 0	❍	❍	↕	❍	↕	❍	❍	
RES (Rij), Bit	1010110BPPPPbbbb	2	6	Bit of (Rij) = 0	❍	❍	↕	❍	❍	❍	↕	
RET	0000001B01010111	2	4	PC = (R3/R7+)	❍	❍	❍	❍	❍	❍	↕	= LD PC, (R3/7+)
RND A	1001000000000000	1	1	Round A with AL(15)	↕	↗	↕	↕	↕	❍	❍	
SET AH, Bit	110011100001bbbb	1	1	Bit of AH = 1	❍	❍	↕	❍	↕	❍	❍	
SET Reg, Bit	11001110RRRRbbbb	2	6	Bit of Reg = 1	❍	❍	↕	❍	↕	❍	❍	
SET (Rij), Bit	1100110BPPPPbbbb	2	6	Bit of (Rij) = 1	❍	❍	↕	❍	❍	❍	↕	
SHFT n	0000010000NNNNNN	1	1	Shift A by n bits	↕	❍	↕	❍	↕	❍	❍	
SHFT A, INV,Cond	000100000I00cccc	1	1	Shift A by +/- EXP bits	↕	❍	↕	❍	↕	❍	❍	
SLEEP	0101010100000000	1	11	Go into sleep mode	❍	❍	❍	❍	❍	❍	❍	
SUB A, Reg, C	00100000C000rrrr	1	1	A = A - Reg - C	↕	↗	↕	↕	↕	↗	❍	Input is sign ext.
SUB A, (Rij), C	0010001BC000PPPP	1	1	A = A - (Rij) - C	↕	↗	↕	↕	↕	❍	↕	Input is sign ext.
SUB A, DRAM, B	0010011BDDDDDDDD	1	1	A = A - (DRAM)	↕	↗	↕	↕	↕	❍	❍	Input is sign ext.
SUB A, (Rij)p, C	0010101BC000PPPP	2	3	A = A - (Rij)p - C	↕	↗	↕	↕	↕	❍	↕	Input is sign ext.
SUB A, Rij, C	0011001BC00000pP	1	1	A = A - Rij - C	↕	↗	↕	↕	↕	❍	❍	Input is sign ext.
SUBI A, IMM, C	00101000C1110000 IIIIIIIIIIIIIIII	2	5	A = A - IMM - C	↕	↗	↕	↕	↕	❍	❍	Input is sign ext.
SUSI A, SIMM	001110SSSSSSSSSS	1	1	A = A - SIMM	↕	↗	↕	↕	↕	❍	❍	Input is sign ext.
SWAP A, Cond	011011110111cccc	1	2	AH <=> AL	↕	↕	↕	❍	↕	❍	❍	
TST Reg, Bit	10110100RRRRbbbb	1	1	Z = not(Bit of Reg)	❍	❍	↕	❍	❍	❍	❍	
TST (Rij), Bit	1011000BPPPPbbbb	1	1	Z = not(Bit of (Rij))	❍	❍	↕	❍	❍	❍	↕	
TGL AH, Bit	111011100001bbbb	1	1	Bit of AH = inversed	❍	❍	↕	❍	↕	❍	❍	
TGL Reg, Bit	11101110RRRRbbbb	2	6	Bit of Reg = inversed	❍	❍	↕	❍	↕	❍	❍	
TGL (Rij), Bit	1110110BPPPPbbbb	2	6	Bit of (Rij) = inversed	❍	❍	↕	❍	❍	❍	↕	

Comments:

A Represents extended accumulator (AE|AH) or (AE|AH|AL)
a 1 = write A with option in MUL
C Select Carry: ‘0’: ignore C flag; ‘1’: add/sub C flag
RRRR Destination register
rrrr Source register
PPPP Address pointer use, PPPP = Ri / Rj
ppppPPPP Address pointer use, pppp = Ri, PPPP = Rj
pP Pointer use, pP = Ri / Rj
B Data RAM bank, 0=i, 1=j

cccc Condition field
bbbbBit position
DDDDDDDD Direct RAM address
IIIIIIIIIIIIIIIIImmediate Data 16 bits
INNNNNN Shift indicator: 0..31 = right shift; -1 .. -31 = left shift
SSSSSSSSSS short immediate data, MSBs = ‘0’, signed or unsigned
M 1 = Double word (A= AE|AH|AL)
0=Single word (A = AE|AL)
P 1 = write PH|PL with option in DADD, DSUB, MUL
m not affected
set / cleared parameter dependent
/ set only, cleared by ST manipulation
= 1† When PC = Destination, add one cycle.