NS85 PROGRAMMERS REFERENCE MANUAL

Document Number: 501-0004

Version 2.1.0

October 16, 2003

Version Number

Revision	Description	Date	Originator
1.0.0	Initial creation.	4/25/03	hwn
1.1.0	Major fixes, flags, pipeline, appendices	5/7/03	hwn
1.2.0	Add dmaconfig register, expand intro, explain repeat count, fix typos.	5/22/03	hwn
2.0.0	New instructions (alu-acu, acu-immediate, alu-mac1) and functions (alu cumulative subtraction, pipeline flush control on progflow, nop token in alu/mac, acu tests). Remapped opcodes. Instruction layout illustration added. XF flag added. New mac input & shift controls. Add labels. More descriptions.	8/15/03	hwn
2.1.0	Add iterate0/1 registers and iterate command. Add ACUs as ACU data source. Fix sm_b# input table errors. Fix typos.	10/16/03	hwn

TABLE OF CONTENTS

SECTION TITLE PAGE

1 Scope

This document describes the programming aspects of the NS85 CU (Compute Unit). While the machine architecture will be discussed, as apropos to the programming, the wise reader in need of more details will seek out the “NS85 DSP MicroArchitecture” document, 501-0006.

This document is mostly constructed of tables of reference material which are often duplicated in several places. This has been done in an attempt to minimise the need to flip back and forth between different areas of the document by placing relevant information close at hand to the reader. Apologies for both the places where this fails and any resultant sense of *déjà vu*.

2 Introduction

The NS85 CU (a.k.a. the NS85 DSP) consists of multiple processing units and multiple memories connected via several data-paths. It is all controlled by a Program Control Unit (PCU) based upon a truncated Very Long Instruction Word (VLIW) architecture. The syntax for programming the DSP is described in this document.

Section 1 describes the scope of this document.

Section 2 provides a brief introduction.

Section 3 of this document details the architecture and resources of the NS85 DSP, the operational characteristics of the pipeline, and a brief introduction to the instructions that the machine can execute.

First is presented an overview of the DSP architecture, illustrating all the major blocks and interconnects. After the top level, each of the major blocks is presented, starting with the PCU. The PCU resources are illustrated, but discussion of the actual operation is postponed until later in the section. Next is presented the architecture of the ALU and the two MAC computation modules – the presentation is of the structure of the modules, not their detailed functionality. The section continues with an architectural presentation of the Address Calculation Unit (ACU) for the Scratch Memory (SM), which are used together as a 2-level cache for memory pointers. Following that, the four ACUs associated with the Data Memories are presented.

Next is presented detailed information about the machine pipeline and a discussion of the “hazards” possible as the the machine architecture executes the stream of instructions. Finally, the different types of instructions are introduced.

Section 4 presents and defines the syntax and language constructs. This section further illuminates the machine architectural details by presenting the means by which the programmer can control such details. First among these details is the selection, for each register in each module, of one of the several possible data sources. Next presented are the details functional operations which are to be performed by each module. Finally is presented the way in which the programmer controls the writing of data to the various machine registers.

Section 5 presents the instruction set at a high level.

Sections 6 through 36 present the summary information for each of the instruction types, each in its own section.

3 Programmers Model

3.1 Introduction

The NS85 CU (a.k.a. the NS85 DSP) is illustrated in Figure 3.1. It consists of multiple compute modules and multiple memories connected via four data busses. It is all controlled by a Program Control Unit (PCU). The PCU contains several programmer accessible registers. General DSP configuration is controlled by the **`config`** and **`dmaconfig`** registers, and instruction repetition is controlled by the **`repeat`** and **`iterate`** registers. Changes in program flow are accomplished by assignment to **`pc`**, the program counter register.

Each DSP instruction controls a subset of the multiple processing units, the multiple memories, and the PCU. A truncated Very Long Instruction Word (VLIW) architecture has been adopted, so not all things possible in the architecture are supported by the instructions.

3.1.1 Compute Modules

The DSP consists primarily of a computation core interconnected via four data busses with several memories. The core of the DSP consists of three compute modules: one ALU, “ALU0” and two multiplier-accumulator (MAC) units, “MAC0” and “MAC1”, respectively. Syntactically, these are usually “**`a0`**”, “**`m0`**”, and “**`m1`**”.

Each of the computation modules has a group of associated registers which are tightly bound. These are the “**`x`**” and “**`y`**” input registers, the “**`s`**” register for computation output, and the “**`f`**” register for condition codes (a.k.a. “flags”). The syntactic names for these registers are constructed by utilizing the compute module name, as above, and then joining the register name to it with an underscore, e.g. “**`a0_x`**”, “**`m1_f`**”, etc.

The registers in each module all serve as selectable inputs to the register-bus, or r-bus, for that module. The r-bus is used for efficient local communications between the computation modules. The r-bus name is formed in the same manner as the names of the registers, e.g. “**`m0_r`**”. The r-bus is not, however, registered: it will dynamically pass the value of the register selected as its source.

3.1.2 Processor Data Memory

Four external data memories, each with an associated Address Calculation Unit (ACU) and data bus, provide data to the DSP core. There is an additional Scratch Memory (SM) and its associated ACU which form a two level pointer cache to service the data memory ACUs. The names of resources associated with these global busses are constructed by concatenating “**`b`**” and the bus number (0-3), an underscore and the name of the resource in the manner described above. E.g. “**`b0_w`**”, “**`b1_c`**”, etc.

3.2 DSP Top Level

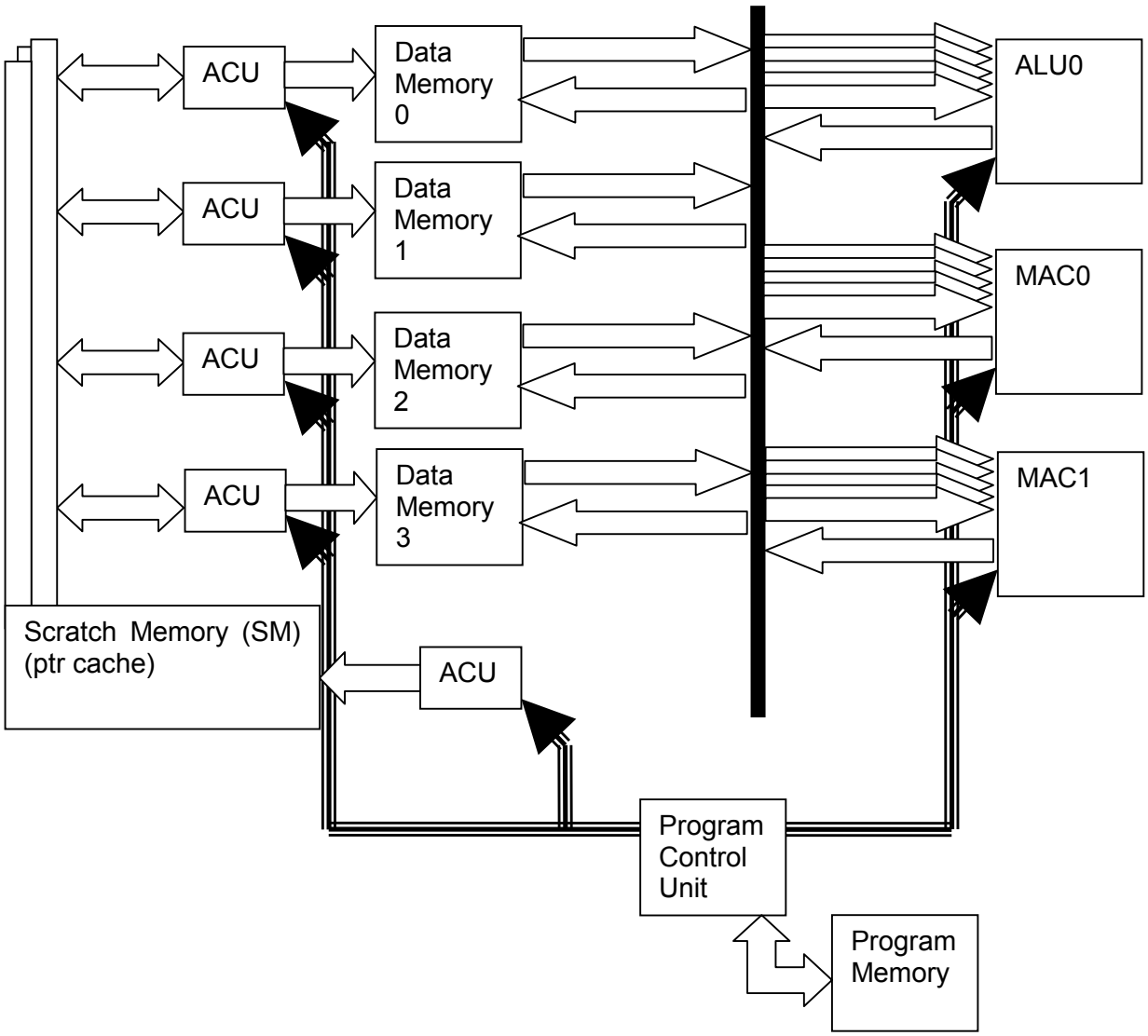


Figure 3.1 - High Level View of the DSP

3.3 Program Unit Resources

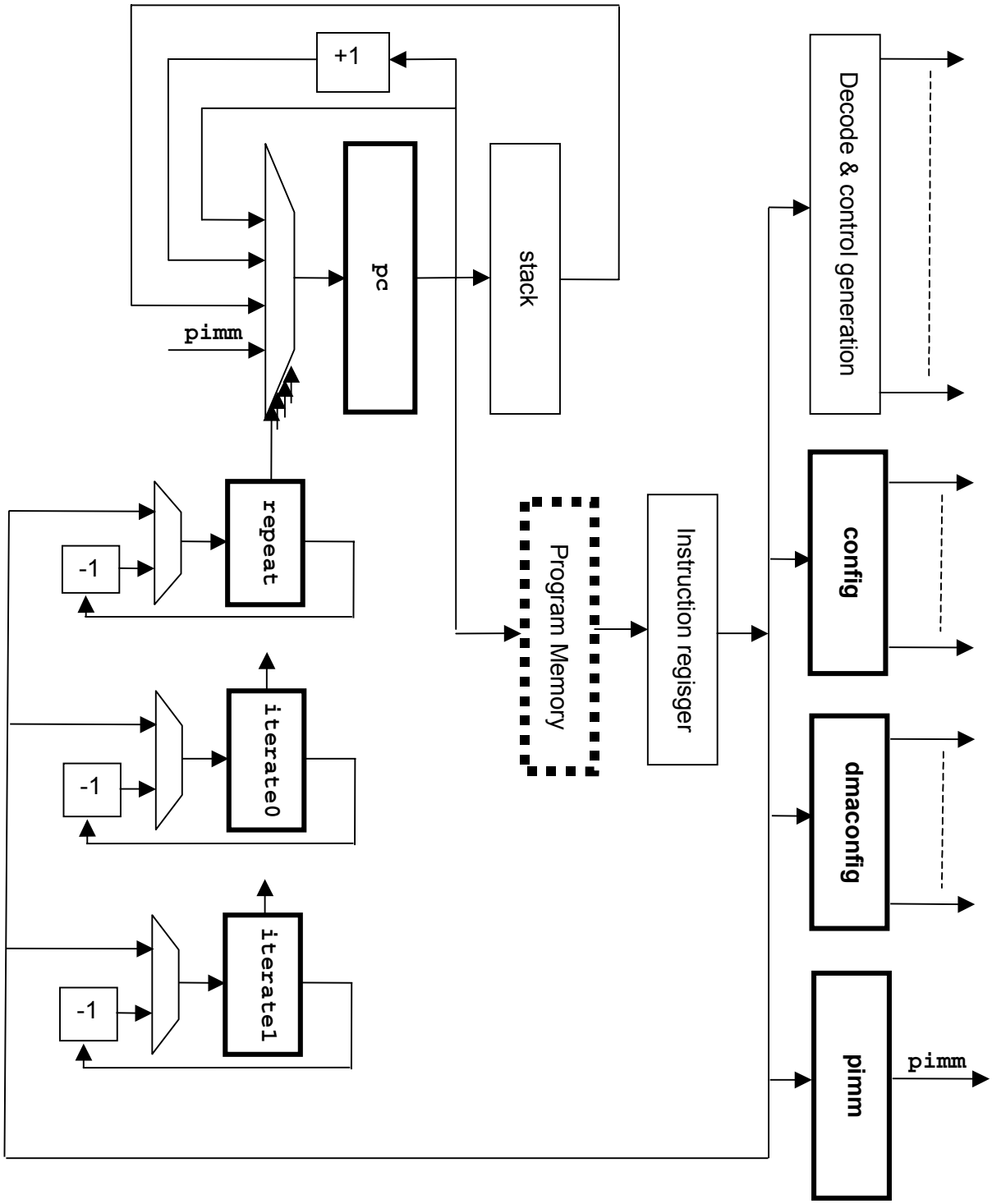


Figure 3.2 – Simplified PCU Block Diagram

3.3.1 Program Counter Register

description	token(s)
the Program Counter register	`pc`

Table 3.1 – Program Counter Register Syntax

3.3.2 Instruction Register

This register captures the instruction read from the program memory.

3.3.3 Pimm Register

description	token(s)
the pimm register (16 bit)	`pimm`

Table 3.2 - Pimm Register Syntax

The **`pimm`** (Program IMMediate) register is used to load constant data from the program space. A Pimm Clause (see 4.2.4.4) is used for this. Because of the pipeline operation, it is possible to load the **`pimm`** register and make use of the data in the same instruction.

3.3.4 Repeat Register

description	token(s)
the Repeat register (16 bit)	`repeat`

Table 3.3 - Repeat Register Syntax

The **`repeat`** register controls the repetition of the next instruction in the program space. The register is 16 bits wide, thus accommodating constants from 65535 (0xffff) down to 1 (zero is not valid.) The hardware does one more repetition than the number which is actually loaded into the register (but the assembler makes a sensible adjustment). The minimum repeat count at the source code level is 2, the maximum is 65536. See the Repeat instruction description in section 34 for more information.

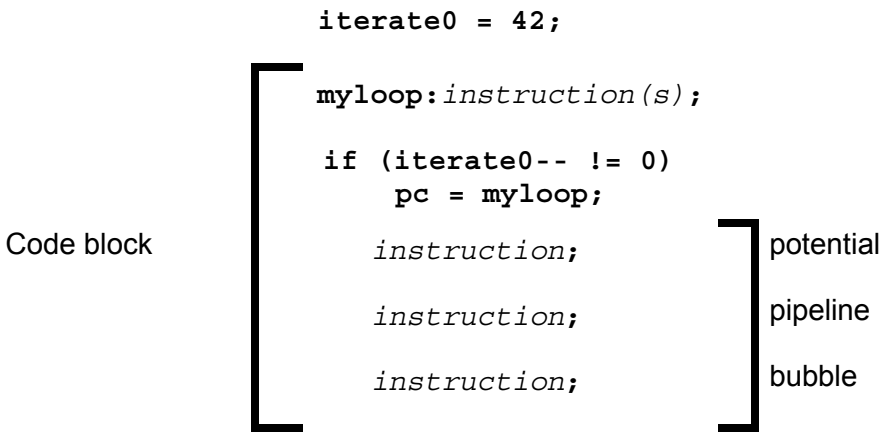
3.3.5 Iterate Registers

description	token(s)
iterate register #0 (16 bit)	**`iterate0`**
iterate register #1 (16 bit)	**`iterate1`**

Table 3.4 – Iterate Register Syntax

The iteration counter registers (**`iterate0`** and **`iterate1`**) provide a low-overhead means to implement the repetition of instruction blocks in the program space. Such program blocks may be nested. In use, the registers are assigned a value corresponding to the number of times the block is to be repeated, and the test/modification operations occur in a conditional instruction.

The registers are 16 bits wide, thus accommodating constants from 0 to 65535 (0xffff.) The minimum number of instructions in a block, including the conditional test instruction and the 3 which follow, is 5. This is illustrated below:



The 3 instructions after the conditional should consist of computation instructions. They cannot include program flow instructions such as **`repeat`**.

3.3.6 DMA Configuration Register

description	token(s)	
the DMA Configuration register	**dmaconfig**	
(see 4.2.4.6)	enable (1)	disable (0)
dmaconfig[0]: Configure ALU0 inputs to access DMA Address pointers instead of the pimm register	**see_dma(on)**	**see_dma(off)**
dmaconfig[1]: Configure A/D block write control	**aden(on)**	**aden(off)**
dmaconfig[2]: Configure D/A block write control	**daen(on)**	**daen(off)**
dmaconfig[3]:Select A/D DMA bus pair 0 (and 3) or 1 (and 2)	**adb(1)**	**adb(0)**
dmaconfig[4]:Select D/A DMA bus pair 0 (and 3) or 1 (and 2)	**dab(1)**	**dab(0)**
dmaconfig[9..5]: specify MAC static left shift value for MAC r_bus data	**msls(*i*)** *i={0,1,2,3,4…19}* *e.g.* **msls(3)**	
dmaconfig[10]: select the MAC r_bus data shift as computed (from a0_s[3:0]) or static (dmaconfig[9..5])	**msft(&a0_s)**	**msft(&msls)**

Table 3.5 – DMA Configuration Register Syntax

3.3.7 Configuration Register

description	token(s)	
the Configuration register	**config**	
(see 4.2.4.5)	**enable (1)**	**disable (0)**
Configure ALU0 SATURATION mode (**config[0]**)	**a0(sat)**	**a0(unsat)**
Configure MAC1 SATURATION mode (**config[1]**)	**m1(sat)**	**m1(unsat)**
Configure MAC1 ROUNDING mode (**config[2]**)	**m1(round)**	**m1(noround)**
Configure MAC1 to FRACTIONAL mode (**config[3]**)	**m1(fract)**	**m1(int)**
Configure MAC0 to SATURATION mode when set (**config[4]**)	**m0(sat)**	**m0(unsat)**
Configure MAC0 ROUNDING mode (**config[5]**)	**m0(round)**	**m0(noround)**
Configure MAC0 to FRACTIONAL mode (**config[6]**)	**m0(fract)**	**m0(int)**
Configure ACU 3 CIRCULAR mode (**config[7]**)	**b3(circ)**	**b3(lin)**
Configure ACU 2 CIRCULAR mode (**config[8]**)	**b2(circ)**	**b2(lin)**
Configure ACU 1 CIRCULAR mode (**config[9]**)	**b1(circ)**	**b1(lin)**
Configure ACU 0 CIRCULAR mode (**config[10]**)	**b0(circ)**	**b0(lin)**
Configure ACU 0 I/O vs Memory mode (**config[11]**)	**io0(on)**	**io0(off)**
Select external flag for testing (**config[15..12]**	**xf(*i*)** *i={0,1,2,3,4…15}* *e.g.* **xf(3)**	

Table 3.6- Configuration Register Syntax

3.4 ALU0 Resources

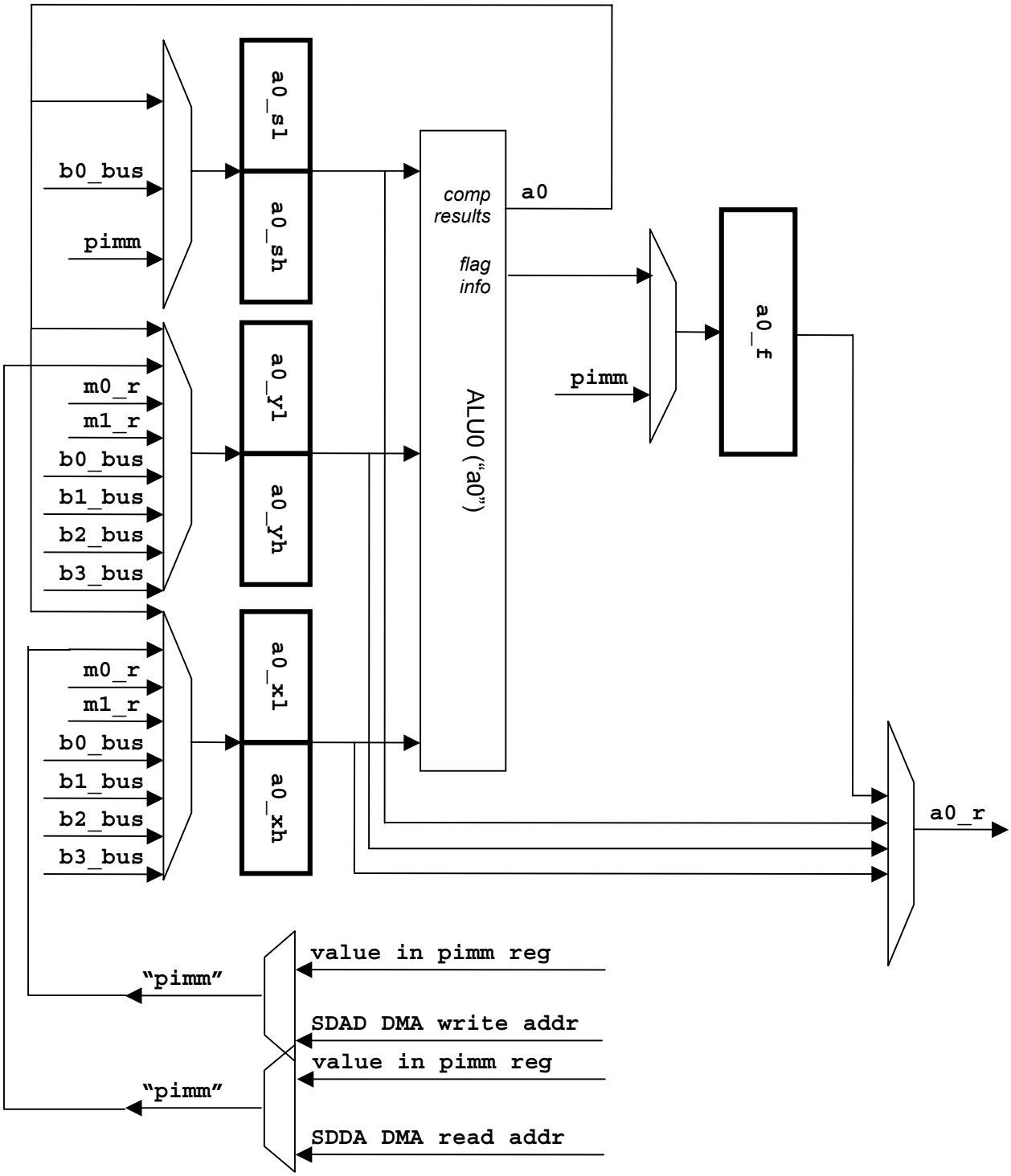


Figure 3.3 - ALU0 Block Diagram

The core of the ALU0 module is an arithmetic-logic unit. The module has two input registers, designated **a0_x** and **a0_y**, and two output registers designated **a0_s** (for results) and **a0_f** (for flag indications.)

Any one of the ALU0 module registers may be driven on to an output "register bus" (or "r-bus"), called **a0_r**, for delivery to the data memories and the other computation modules. Note that the **a0_r** bus is not a register.

In a single instruction cycle, the ALU0 module can load data from any one or two of the four data memory systems, via **b0_bus**, **b1_bus**, **b2_bus**, or **b3_bus**, respectively. Immediate data, via the Pimm Register, may also be input to the module. The contents of registers in the other computation modules may be accessed via the appropriate r-bus.

The ALU0 module also has special hardware to allow the loading of the current DMA addresses into the ALU0 input registers. This functionality is controlled by bit 0 of the DMA Configuration Register. When this function is selected, the see clause must select pimm, as illustrated in Figure 3.3.

3.4.1 ALU0 X register

description	token(s)

the entire ALU0 x register	`a0_x[31:0]`
	`a0_x`
	`{a0_xh,a0_xl}`

the high half of the ALU0 x register	`a0_x[31:16]` `a0_xh`
the low half of the ALU0 x register	`a0_x[15:0]` `a0_xl`

Table 3.7 - ALU0 X Register Syntax

description	token(s)
the program immediate register or the DMA write address used for incoming SDAD data as selected by “see_dma” dmaconfig register bit 0.	`pimm`
the ALU result	`a0`
the B0 bus from data memory	`b0_bus` `b0`
the B1 bus from data memory	`b1_bus` `b1`
the B2 bus from data memory	`b2_bus` `b2`
the B3 bus from data memory	`b3_bus` `b3`

the register bus from MAC0	`m0_r`
	`m0_reg`
	`m0_bus`

the register bus from mac1	`m1_r`
	`m1_reg`
	`m1_bus`

typical use (see 4.2.4.7.1)	`a0_x(&pimm)`

Table 3.8 - ALU0 X Register Inputs

3.4.2 ALU0 Y register

description	token(s)

the entire ALU0 y register	`a0_y[31:0]`
	`a0_y`
	`{a0_yh,a0_yl}`

the high half of the ALU0 y register	`a0_y[31:16]` `a0_yh`
the low half of the ALU0 y register	`a0_y[15:0]` `a0_yl`

Table 3.9 - ALU0 Y Register Syntax

description	token(s)
the program immediate register or the DMA read address used for outgoing SDDA data as selected by "see_dma" dmaconfig register bit 0.	`pimm`
the ALU result	`a0`
the B0 bus from data memory	`b0_bus` `b0`
the B1 bus from data memory	`b1_bus` `b1`
the B2 bus from data memory	`b2_bus` `b2`
the B3 bus from data memory	`b3_bus` `b3`

the register bus from MAC0	`m0_r`
	`m0_reg`
	`m0_bus`

the register bus from mac1	`m1_r`
	`m1_reg`
	`m1_bus`

typical use (see 4.2.4.7.1)	`a0_y(&m0)`

Table 3.10 - ALU0 Y Register Inputs

3.4.3 ALU0 S register

description	token(s)

the entire ALU0 s register	`a0_s[31:0]`
	`a0_s`
	`{a0_sh,a0_sl}`

the high half of the ALU0 s register	`a0_s[31:16]` `a0_sh`
the low half of the ALU0 s register	`a0_s[15:0]` `a0_sl`

Table 3.11 - ALU0 S Register Syntax

description	token(s)
the program immediate register	`pimm`
sign extended value from the program immediate register	`(int)pimm`
the B0 bus from data memory	`b0_bus` `b0`
the ALU result	`a0`
typical use (see 4.2.4.7.1)	`a0_s(&(int)pimm)`

Table 3.12 - ALU0 S Register Inputs

3.4.4 ALU0 Flags register

description	token(s)
the entire ALU0 Flags register	`a0_f` `a0_flag`
the OVERFLOW flag bit in the ALU0 Flags register (a0_f[0])	`a0ov`
the STICKY OVERFLOW flag bit in the ALU0 Flags register (a0_f[1])	`a0sov`
the ZERO flag bit in the ALU0 Flags register (a0_f[2])	`a0z`
the STICKY ZERO flag bit in the ALU0 Flags register (a0_f[3])	`a0sz`
the GREATER_THAN_OR_EQUAL_TO_ZERO flag bit in the ALU0 Flags register (a0_f[4])	`a0ge`
the STICKY GREATER_THAN_OR_EQUAL_TO_ZERO flag bit in the ALU0 Flags register (a0_f[5])	`a0sge`
the GREATER_THAN_ZERO flag bit in the ALU0 Flags register (a0_f[6])	`a0gt`
the STICKY GREATER_THAN_ZERO flag bit in the ALU0 Flags register (a0_f[7])	`a0sgt`
the CARRY_OUT flag bit in the ALU0 Flags register (a0_f[8])	`a0c`
the STICKY CARRY_OUT flag bit in the ALU0 Flags register (a0_f[9])	`a0sc`

Table 3.13 - ALU0 Flags Register Syntax

description	token(s)
the program immediate register	`pimm`
the ALU result	`a0`
typical use (see 4.2.4.7.1)	`a0_f(&a0)`

Table 3.14 - ALU0 Flags Register Inputs

3.4.5 ALU0 Register Bus

description	token(s)
the ALU0 register bus	**a0_r** **a0_reg** **a0_bus**

Table 3.15 - ALU0 Register Bus Syntax

description	token(s)
the high half of the ALU0 x register	**a0_x[31:16]** **a0_xh**
the low half of the ALU0 x register	**a0_x[15:0]** **a0_xl**
the high half of the ALU0 y register	**a0_y[31:16]** **a0_yh**
the low half of the ALU0 y register	**a0_y[15:0]** **a0_yl**
the high half of the ALU0 s register	**a0_s[31:16]** **a0_sh**
the low half of the ALU0 s register	**a0_s[15:0]** **a0_sl**
the entire ALU0 Flags register	**a0_f** **a0_flag**
typical use (see 4.2.4.7.1)	**a0_r(&a0_sh)**

Table 3.16 - ALU0 Register Bus Inputs

3.5 MAC0 Resources

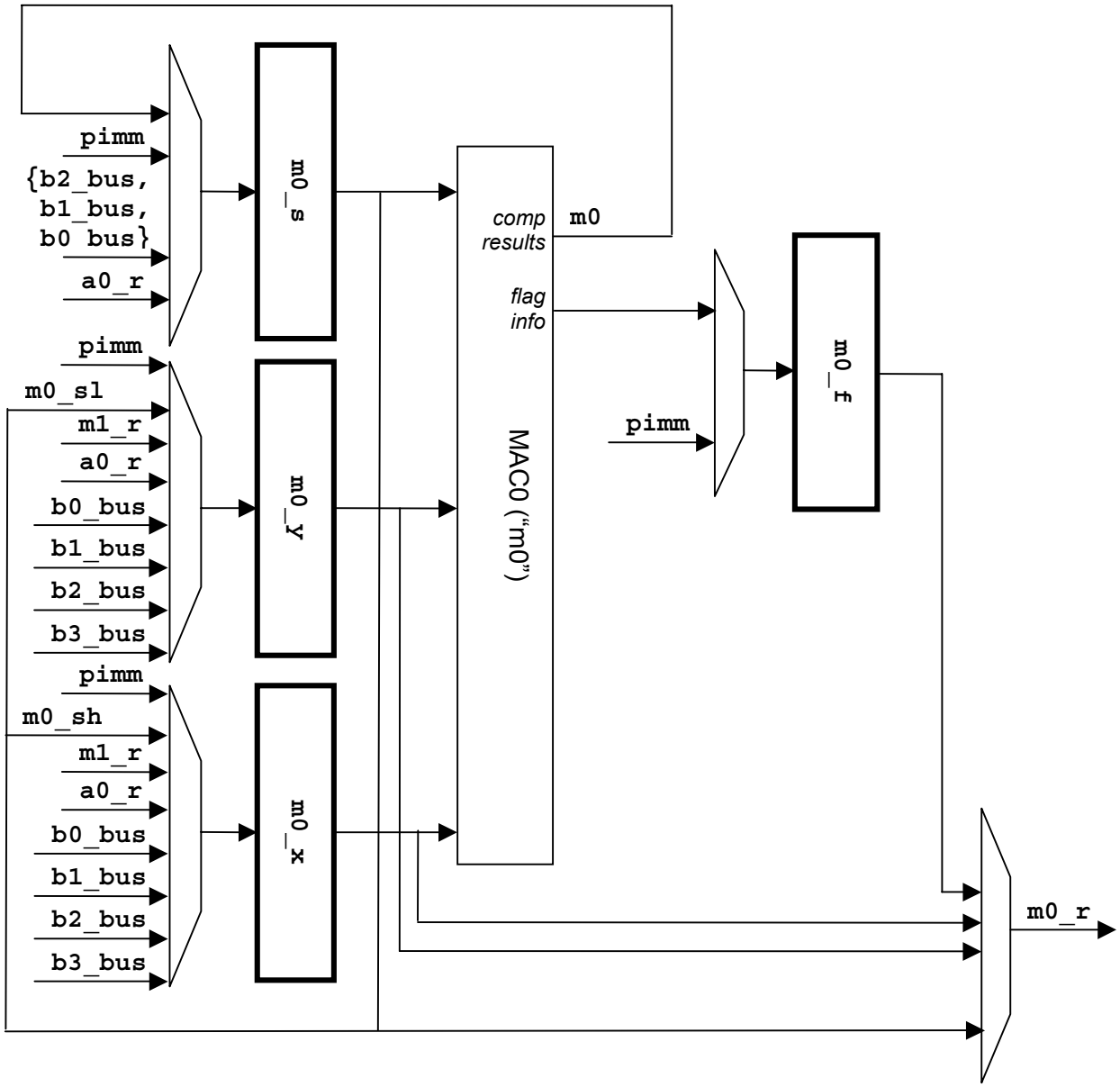


Figure 3.4 - MAC0 Block Diagram

The core of the MAC0 module is a multiplier and accumulator unit. The module has two input registers, designated **`m0_x`** and **`m0_y`**, and two output registers designated **`m0_s`** (for results) and **`m0_f`** (for flag indications.)

Any one of the MAC0 module registers may be driven on to an output "register bus" (or "r-bus"), called **`m0_r`**, for delivery to the data memories and the other computation modules. Note that the **`m0_r`** bus is not a register.

In a single instruction cycle, the MAC0 module can load data from any one or two of the four data memory systems, via **b0_bus**, **b1_bus**, **b2_bus**, or **b3_bus**, respectively. Immediate data, via the Pimm Register, may also be input to the module. The contents of registers in the other computation modules may be accessed via the appropriate r-bus.

3.5.1 MAC0 X register

description	token(s)
the entire MAC0 x register	**m0_x**

Table 3.17 - MAC0 X Register Syntax

description	token(s)
the program immediate register	**pimm**
the high half of the m0_s register	**m0_sh**
the B0 bus from data memory	**b0_bus** **b0**
the B1 bus from data memory	**b1_bus** **b1**
the B2 bus from data memory	**b2_bus** **b2**
the B3 bus from data memory	**b3_bus** **b3**
the register bus from MAC1	**m1_r** **m1_reg** **m1_bus**
the register bus from ALU0	**a0_r** **a0_reg** **a0_bus**
typical use (see 4.2.4.7.2)	**m0_x(&pimm)**

Table 3.18 - MAC0 X Register Inputs

3.5.2 MAC0 Y register

description	token(s)
the entire MAC0 y register	`m0_y`

Table 3.19 - MAC0 Y Register Syntax

description	token(s)
the program immediate register	`pimm`
the low half of the m0_s register	`m0_sl`
the B0 bus from data memory	`b0_bus` `b0`
the B1 bus from data memory	`b1_bus` `b1`
the B2 bus from data memory	`b2_bus` `b2`
the B3 bus from data memory	`b3_bus` `b3`

the register bus from MAC1	`m1_r`
	`m1_reg`
	`m1_bus`

the register bus from ALU0	`a0_r`
	`a0_reg`
	`a0_bus`

typical use (see 4.2.4.7.2)	`m0_y(&b1)`

Table 3.20 - MAC0 Y Register Inputs

3.5.3 MAC0 S register

description	token(s)
the entire MAC0 s register, including guard bits	`m0_s[39:0]`
the entire MAC0 s register, excluding guard bits	`m0_s[31:0]` `m0_s`
the guard bits of the MAC0 s register	`m0_s[39:32]` `m0_sg`
the high half of the MAC0 s register	`m0_s[31:16]` `m0_sh`
the low half of the MAC0 s register	`m0_s[15:0]` `m0_sl`

Table 3.21 - MAC0 S Register Syntax

description	token(s)
sign extended value from the program immediate register	`(long)pimm`
the {b2[7:0],b1,b0} buses from data memory	`b_bus`
the MAC result	`m0`

the register bus from ALU0 (sign extended value biased to m0_sh, m0_sl gets zeroes)	`a0_r`
	`a0_reg`
	`a0_bus`

typical use (see 4.2.4.7.2)	`m0_s(&(long)pimm)`

Table 3.22 - MAC0 S Register Inputs

3.5.4 MAC0 Flags register

description	token(s)
the entire MAC0 Flags register	`m0_f` `m0_flag`
the 40_BIT_OVERFLOW flag bit in the MAC0 Flags register (m0_f[0])	`m0eov`
the STICKY 40_BIT_OVERFLOW flag bit in the MAC0 Flags register (m0_f[1])	`m0seov`

Table 3.23 - MAC0 Flags Register Syntax

description	token(s)
the program immediate register	`pimm`
the MAC result	`m0`
typical use (see 4.2.4.7.2)	`m0_f(&m0)`

Table 3.24 - MAC0 Flags Register Inputs

3.5.5 MAC0 Register Bus

description	token(s)

the MAC0 register bus	`m0_r`
	`m0_reg`
	`m0_bus`

Table 3.25 - MAC0 Register Bus Syntax

description	token(s)
the MAC0 x register	`m0_x`
the MAC0 y register	`m0_y`
the guard bits of the MAC0 s register	`m0_s[39:32]` `m0_sg`
the high half of the MAC0 s register	`m0_s[31:16]` `m0_sh`
the low half of the MAC0 s register	`m0_s[15:0]` `m0_sl`
The sign extended version of the guard bits of the MAC0 s register	`(int)m0_s[39:32]` `(int)m0_sg`
the entire MAC0 Flags register	`m0_f`
typical use (see 4.2.4.7.2)	`m0_r(&m0_sh)`

Table 3.26 - MAC0 Register Bus Inputs

3.6 MAC1 Resources

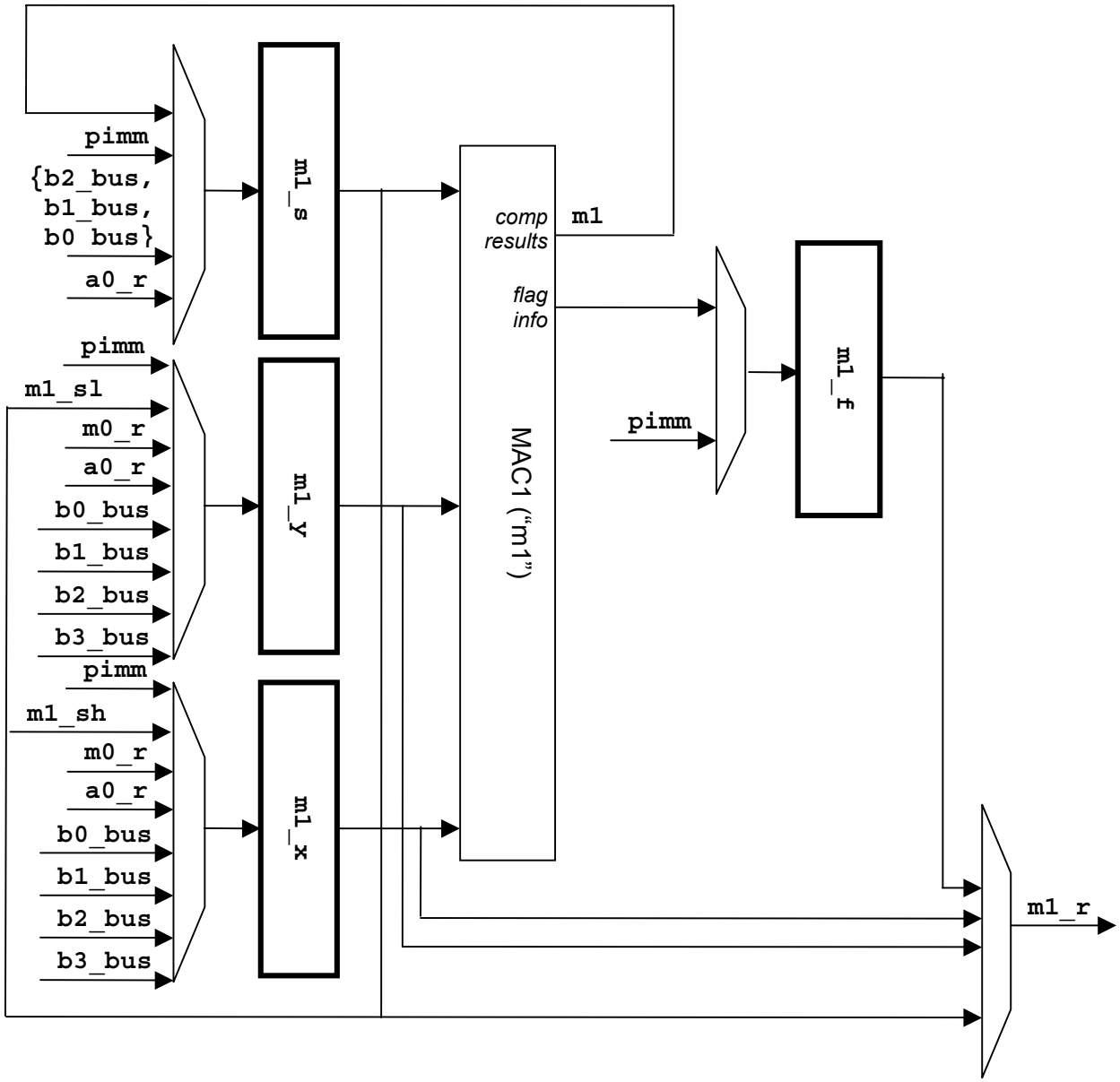


Figure 3.5 – MAC1 Block Diagram

The core of the MAC1 module is a multiplier and accumulator unit. The module has two input registers, designated **`m1_x`** and **`m1_y`**, and two output registers designated **`m1_s`** (for results) and **`m1_f`** (for flag indications.)

Any one of the MAC1 module registers may be driven on to an output "register bus" (or "r-bus"), called **`m1_r`**, for delivery to the data memories and the other computation modules. Note that the **`m1_r`** bus is not a register.

In a single instruction cycle, the MAC1 module can load data from any one or two of the four data memory systems, via **b0_bus**, **b1_bus**, **b2_bus**, or **b3_bus**, respectively. Immediate data, via the Pimm Register, may also be input to the module. The contents of registers in the other computation modules may be accessed via the appropriate r-bus.

3.6.1 MAC1 X register

description	token(s)
the entire MAC1 x register	**m1_x**

Table 3.27 – MAC1 X Register Syntax

description	token(s)
the program immediate register	**pimm**
the high half of the m1_s register	**m1_sh**
the B0 bus from data memory	**b0_bus** **b0**
the B1 bus from data memory	**b1_bus** **b1**
the B2 bus from data memory	**b2_bus** **b2**
the B3 bus from data memory	**b3_bus** **b3**
the register bus from MAC0	**m0_r** **m0_reg** **m0_bus**
the register bus from ALU0	**a0_r** **a0_reg** **a0_bus**
typical use (see 4.2.4.7.3)	**m1_x(&b3_bus)**

Table 3.28 – MAC1 X Register Inputs

3.6.2 MAC1 Y register

description	token(s)
the entire MAC1 y register	`m1_y`

Table 3.29 – MAC1 Y Register Syntax

description	token(s)
the program immediate register	`pimm`
the low half of the m1_s register	`m1_sl`
the B0 bus from data memory	`b0_bus` `b0`
the B1 bus from data memory	`b1_bus` `b1`
the B2 bus from data memory	`b2_bus` `b2`
the B3 bus from data memory	`b3_bus` `b3`

the register bus from MAC0	`m0_r`
	`m0_reg`
	`m0_bus`

the register bus from ALU0	`a0_r`
	`a0_reg`
	`a0_bus`

typical use (see 4.2.4.7.3)	`m1_y(&b1)`

Table 3.30 – MAC1 Y Register Inputs

3.6.3 MAC1 S register

description	token(s)
the entire MAC1 s register, including guard bits	`m1_s[39:0]`
the entire MAC1 s register, excluding guard bits	`m1_s[31:0]` `m1_s`
the guard bits of the MAC1 s register	`m1_s[39:32]` `m1_sg`
the high half of the MAC1 s register	`m1_s[31:16]` `m1_sh`
the low half of the MAC1 s register	`m1_s[15:0]` `m1_sl`

Table 3.31 – MAC1 S Register Syntax

description	token(s)
sign extended value from the program immediate register	`(long)pimm`
the {b2[7:0],b1,b0} buses from data memory	`b_bus`
the MAC result	`m1`

the register bus from ALU0 (sign extended value biased to m1_sh, m1_sl gets zeroes)	`a0_r`
	`a0_reg`
	`a0_bus`

typical use (see 4.2.4.7.3)	`m1_s(&(long)pimm)`

Table 3.32 – MAC1 S Register Inputs

3.6.4 MAC1 Flags register

description	token(s)
the entire MAC1 Flags register	**m1_f** **m1_flag**
the 40_BIT_OVERFLOW flag bit in the MAC1 Flags register (m1_f[0])	**m1eov**
the STICKY 40_BIT_OVERFLOW flag bit in the MAC1 Flags register (m1_f[1])	**m1seov**

Table 3.33 – MAC1 Flags Register Syntax

description	token(s)
the program immediate register	**pimm**
the MAC result	**m1**
typical use (see 4.2.4.7.3)	**m1_f(&m1)**

Table 3.34 – MAC1 Flags Register Inputs

3.6.5 MAC1 Register Bus

description	token(s)

the MAC1 register bus	`m1_r`
	`m1_reg`
	`m1_bus`

Table 3.35 – MAC1 Register Bus Syntax

description	token(s)
the MAC1 x register	`m1_x`
the MAC1 y register	`m1_y`
the guard bits of the MAC1 s register	`m1_s[39:32]` `m1_sg`
the high half of the MAC1 s register	`m1_s[31:16]` `m1_sh`
the low half of the MAC1 s register	`m1_s[15:0]` `m1_sl`
The sign extended version of the guard bits of the MAC1 s register	`(int)m1_s[39:32]` `(int)m1_sg`
the entire MAC1 Flags register	`m1_f`
typical use (see 4.2.4.7.3)	`m1_r(&m1_sh)`

Table 3.36 – MAC1 Register Bus Inputs

3.7 SM ACU & Memory Resources

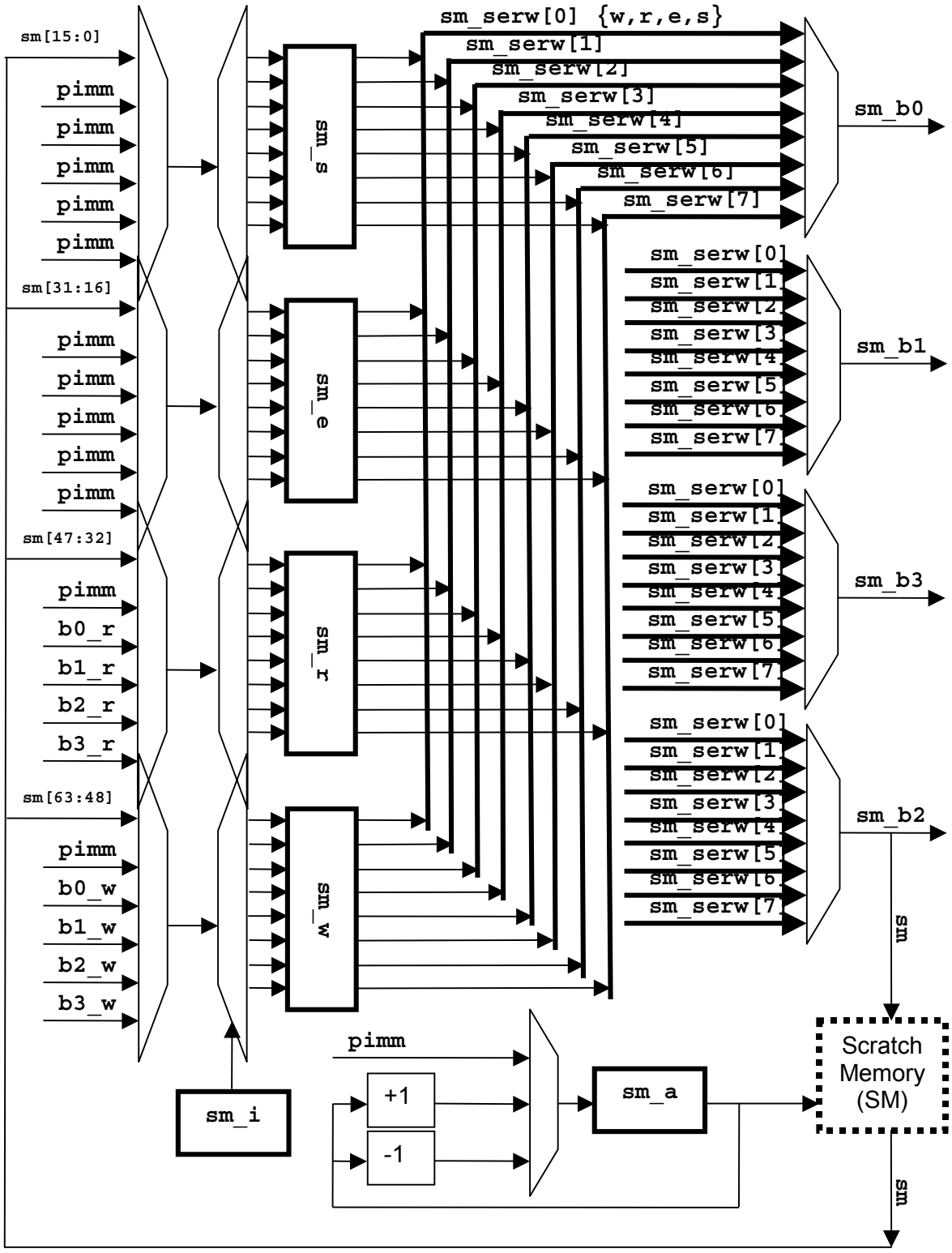


Figure 3.6 – SM ACU Block Diagram

The Scratch Memory (SM) consists of the SM RAM and the SM ACU. The SM ACU contains 8 cached sets of start (S), end (E), read (R), and write (W) registers for quick access by the Data Memory ACU units. The SM RAM holds 64 additional pointer sets which can be easily recalled to the cache. The SM RAM is 64 bits wide and handles a {S,E,R,W} set in a single cycle.

3.7.1 SM SERW

Each of the 4 Data Memory ACU units (B0, B1, B2, B3) is provided with an individually selectable set of associated pointers from the cache. The syntactic tokens for these are **sm_serw[0]** through **sm_serw[7]**.

3.7.2 SM A register

description	token(s)
the SM ACU Address (to SM RAM) register	`sm_a`
typical use (see 4.2.4.8.4)	`sm_a=pimm`

Table 3.37 – SM A Register Syntax

3.7.3 SM I register

description	token(s)
The SM ACU Cache Write Index register specifies the cached register set to be written. (valid: 0-7)	`sm_i`
typical use (see 4.2.4.7.4)	`sm_i=0`

Table 3.38 – SM I Register Syntax

3.7.4 SM S register

description	token(s)
the pointer cache SM S register set	`sm_s`

Table 3.39 – SM S Register Set Syntax

description	token(s)
the program immediate register	`pimm`
pointer data from SM memory	`sm`
typical use (see 4.2.4.7.4)	`sm_s(&sm)`

Table 3.40 – SM S Register Set Inputs

3.7.5 SM E register

description	token(s)
the pointer cache SM E register set	`sm_e`

Table 3.41 – SM E Register Set Syntax

description	token(s)
the program immediate register	`pimm`
pointer data from SM memory	`sm`
typical use (see 4.2.4.7.4)	`sm_e(&sm)`

Table 3.42 – SM E Register Set Inputs

3.7.6 SM R register

description	token(s)
the pointer cache SM R register set	`sm_r`

Table 3.43 – SM R Register Set Syntax

description	token(s)
the program immediate register	`pimm`
pointer data from SM	`sm`
the ACU0 read pointer register	`b0_r`
the ACU1 read pointer register	`b1_r`
the ACU2 read pointer register	`b2_r`
the ACU3 read pointer register	`b3_r`
typical use (see 4.2.4.7.7)	`sm_r(&b0_r)`

Table 3.44 – SM R Register Set Inputs

3.7.7 SM W register

description	token(s)
the pointer cache SM W register set	`sm_w`

Table 3.45 – SM R Register Set Syntax

description	token(s)
the program immediate register	`pimm`
pointer data from SM	`sm`
the ACU0 write pointer register	`b0_w`
the ACU1 write pointer register	`b1_w`
the ACU2 write pointer register	`b2_w`
the ACU3 write pointer register	`b3_w`
typical use (see 4.2.4.7.7)	`sm_w(&b0_w)`

Table 3.46 – SM W Register Set Inputs

3.7.8 SM B0 bus

description	token(s)
read-back bus for pointer data cached in the SM register sets to the ACU0 unit	`sm_b0`

Table 3.47 – SM B0 Bus Syntax

description	token(s)
pointer set (s, e, r, and w registers) number 0 from the SM cache	`sm_serw[0]`
cached pointer set number 1	`sm_serw[1]`
cached pointer set number 2	`sm_serw[2]`
cached pointer set number 3	`sm_serw[3]`
cached pointer set number 4	`sm_serw[4]`
cached pointer set number 5	`sm_serw[5]`
cached pointer set number 6	`sm_serw[6]`
cached pointer set number 7	`sm_serw[7]`
typical use (see 4.2.4.7.4)	`sm_b0(&sm_serw[0])`

Table 3.48 – SM B0 Bus Inputs

3.7.9 SM B1 bus

description	token(s)
read-back bus for pointer data cached in the SM register sets to the ACU1 unit	`sm_b1`

Table 3.49 – SM B1 Bus Syntax

description	token(s)
pointer set (s, e, r, and w registers) number 0 from the SM cache	`sm_serw[0]`
cached pointer set number 1	`sm_serw[1]`
cached pointer set number 2	`sm_serw[2]`
cached pointer set number 3	`sm_serw[3]`
cached pointer set number 4	`sm_serw[4]`
cached pointer set number 5	`sm_serw[5]`
cached pointer set number 6	`sm_serw[6]`
cached pointer set number 7	`sm_serw[7]`
typical use (see 4.2.4.7.4)	`sm_b1(&sm_serw[0])`

Table 3.50 – SM B1 Bus Inputs

3.7.10 SM B2 bus

description	token(s)
read-back bus for pointer data cached in the SM register sets to the ACU2 unit	`sm_b2`

Table 3.51 – SM B2 Bus Syntax

description	token(s)
pointer set (s, e, r, and w registers) number 0 from the SM cache	`sm_serw[0]`
cached pointer set number 1	`sm_serw[1]`
cached pointer set number 2	`sm_serw[2]`
cached pointer set number 3	`sm_serw[3]`
cached pointer set number 4	`sm_serw[4]`
cached pointer set number 5	`sm_serw[5]`
cached pointer set number 6	`sm_serw[6]`
cached pointer set number 7	`sm_serw[7]`
typical use (see 4.2.4.7.4)	`sm_b2(&sm_serw[0])`

Table 3.52 – SM B2 Bus Inputs

3.7.11 SM B3 bus

description	token(s)
read-back bus for pointer data cached in the SM register sets to the ACU3 unit	`sm_b3`

Table 3.53 – SM B3 Bus Syntax

description	token(s)
pointer set (s, e, r, and w registers) number 0 from the SM cache	`sm_serw[0]`
cached pointer set number 1	`sm_serw[1]`
cached pointer set number 2	`sm_serw[2]`
cached pointer set number 3	`sm_serw[3]`
cached pointer set number 4	`sm_serw[4]`
cached pointer set number 5	`sm_serw[5]`
cached pointer set number 6	`sm_serw[6]`
cached pointer set number 7	`sm_serw[7]`
typical use (see 4.2.4.7.4)	`sm_b3(&sm_serw[0])`

Table 3.54 – SM B3 Bus Inputs

3.8 ACU 0 Data Memory & ACU Resources

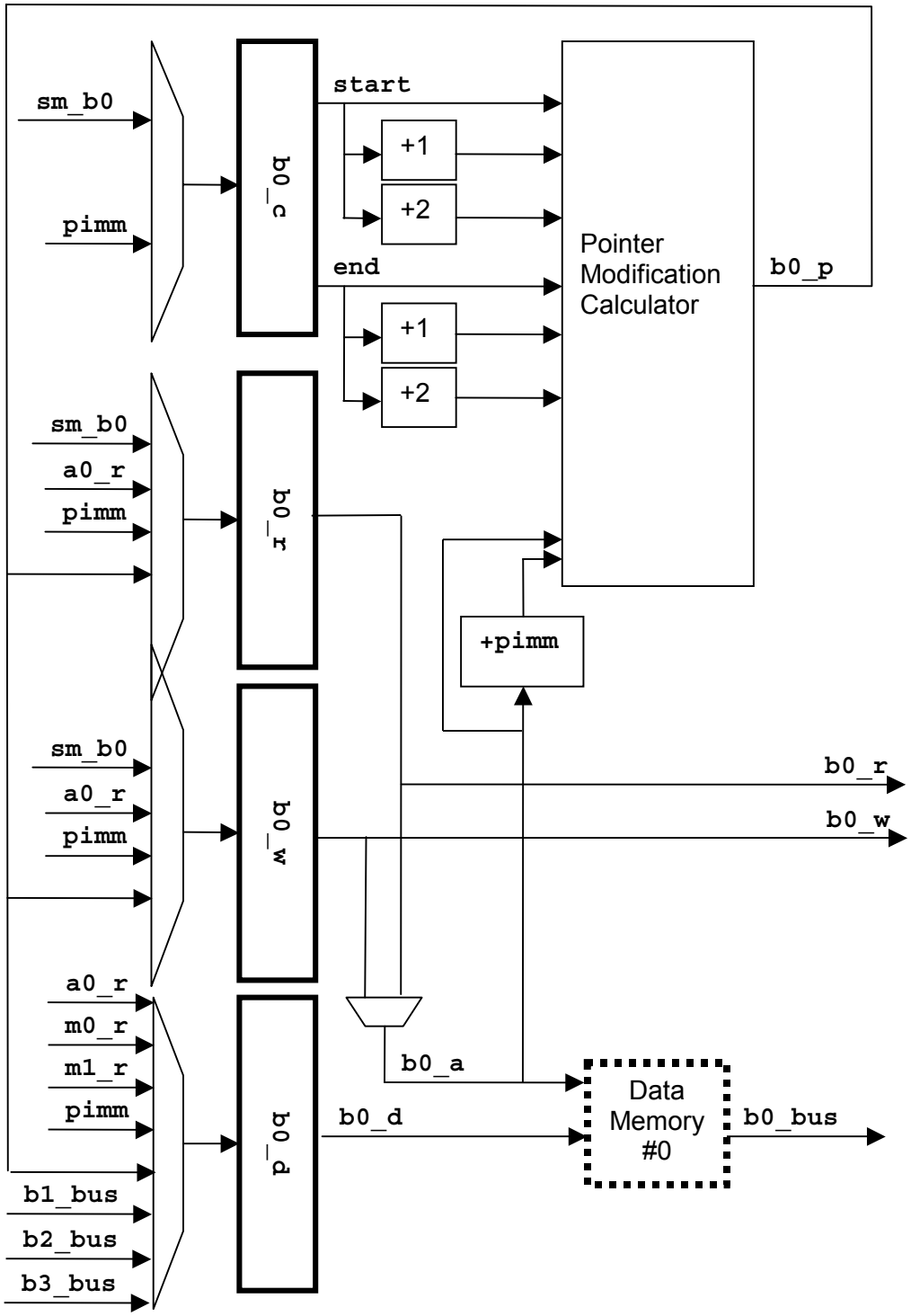


Figure 3.7 – B0 ACU & Memory Block Diagram

3.8.1 ACU0 C register

description	token(s)
the ACU0 c register pair (circular start and end addresses)	`b0_c`
the ACU0 circular end address register, available only for testing in the conditional (“if”) instruction	`b0_e`

Table 3.55 – ACU0 C Register Syntax

description	token(s)
the program immediate register	`pimm`
Cached pointer data from SM	`sm`
typical use (see 4.2.4.7.7)	`b0_c(&sm)`

Table 3.56 – ACU0 C Register Inputs

3.8.2 ACU0 R register

description	token(s)
the ACU0 r register (read ptr)	`b0_r`

Table 3.57 – ACU0 R Register Syntax

description	token(s)
the program immediate register	`pimm`
cached pointer data from SM	`sm`
the ALU0 register bus	`a0_r`
pointer as modified by ACU0	`b0_p`
typical use (see 4.2.4.7.7)	`b0_r(&sm)`

Table 3.58 – ACU0 R Register Inputs

3.8.3 ACU0 W register

description	token(s)
the ACU0 w register (write ptr)	`b0_w`

Table 3.59 – ACU0 W Register Syntax

description	token(s)
the program immediate register	`pimm`
cached pointer data from SM	`sm`
the ALU0 register bus	`a0_r`
pointer as modified by ACU0	`b0_p`
typical use (see 4.2.4.7.7)	`b0_w(&sm)`

Table 3.60 – ACU0 W Register Inputs

3.8.4 ACU0 D register

description	token(s)
the ACU0 d register (write data)	`b0_d`

Table 3.61 – ACU0 D Register Syntax

description	token(s)
the program immediate register	`pimm`
the ALU0 register bus	`a0_r`
the MAC0 register bus	`m0_r`
the MAC1 register bus	`m1_r`
pointer as modified by ACU0	`b0_p`
Data Memory Bus 1	`b1_bus`
Data Memory Bus 2	`b2_bus`
Data Memory Bus 3	`b3_bus`
typical use (see 4.2.4.7.7)	`b0_d(&pimm)`

Table 3.62 – ACU0 D Register Inputs

3.8.5 ACU0 Post-modified Pointer

description	token(s)
pointer as modified by ACU0	`b0_p`
typical use (see 4.2.4.8.5)	`b0_p = b0_r + 2`

Table 3.63 – ACU0 Post-modified Pointer

3.9 ACU 1 Data Memory & ACU Resources

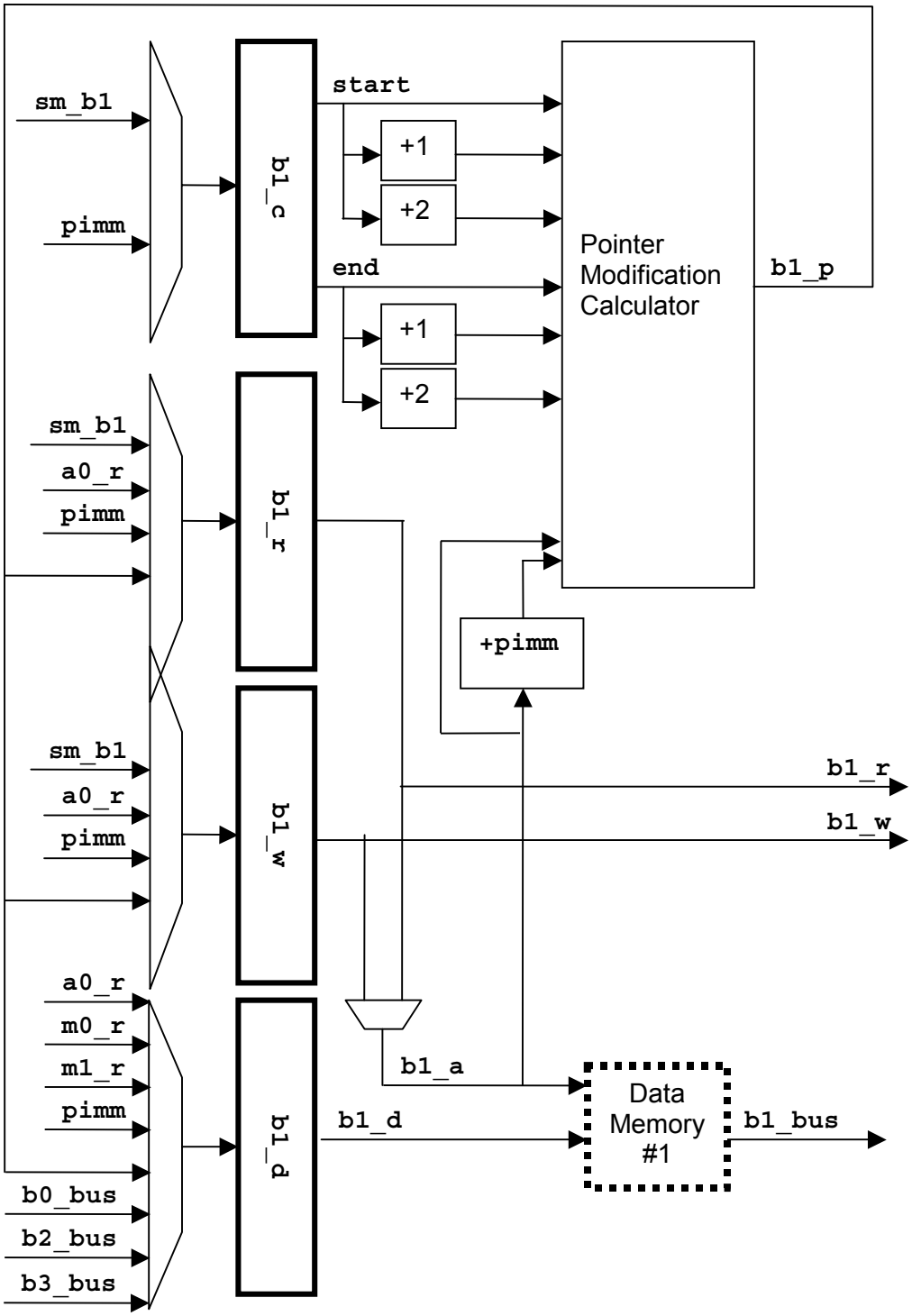


Figure 3.8 – B1 ACU Block Diagram

3.9.1 ACU1 C register

description	token(s)
the ACU1 c register pair (circular start and end addresses)	`b1_c`
the ACU1 circular end address register, available only for testing in the conditional (“if”) instruction	`b1_e`

Table 3.64 – ACU1 C Register Syntax

description	token(s)
the program immediate register	`pimm`
Cached pointer data from SM	`sm`
typical use (see 4.2.4.7.9)	`b1_c(&sm)`

Table 3.65 – ACU1 C Register Inputs

3.9.2 ACU1 R register

description	token(s)
the ACU1 r register (read ptr)	`b1_r`

Table 3.66 – ACU1 R Register Syntax

description	token(s)
the program immediate register	`pimm`
cached pointer data from SM	`sm`
the ALU0 register bus	`a0_r`
pointer as modified by ACU1	`b1_p`
typical use (see 4.2.4.7.9)	`b1_r(&sm)`

Table 3.67 – ACU1 R Register Inputs

3.9.3 ACU1 W register

description	token(s)
the ACU1 w register (write ptr)	`b1_w`

Table 3.68 – ACU1 W Register Syntax

description	token(s)
the program immediate register	`pimm`
cached pointer data from SM	`sm`
the ALU0 register bus	`a0_r`
pointer as modified by ACU1	`b1_p`
typical use (see 4.2.4.7.9)	`b1_w(&sm)`

Table 3.69 – ACU1 W Register Inputs

3.9.4 ACU1 D register

description	token(s)
the ACU1 d register (write data)	`b1_d`

Table 3.70 – ACU1 D Register Syntax

description	token(s)
the program immediate register	`pimm`
the ALU0 register bus	`a0_r`
the MAC0 register bus	`m0_r`
the MAC1 register bus	`m1_r`
pointer as modified by ACU1	`b1_p`
Data Memory Bus 0	`b0_bus`
Data Memory Bus 2	`b2_bus`
Data Memory Bus 3	`b3_bus`
typical use (see 4.2.4.7.9)	`b1_d(&b0_bus)`

Table 3.71 – ACU1 D Register Inputs

3.9.5 ACU1 Post-modified Pointer

description	token(s)
pointer as modified by ACU1	`b1_p`
typical use (see 4.2.4.8.6)	`b1_p = b1_r + 1`

Table 3.72 – ACU1 Post-modified Pointer

3.10 ACU 2 Data Memory & ACU Resources

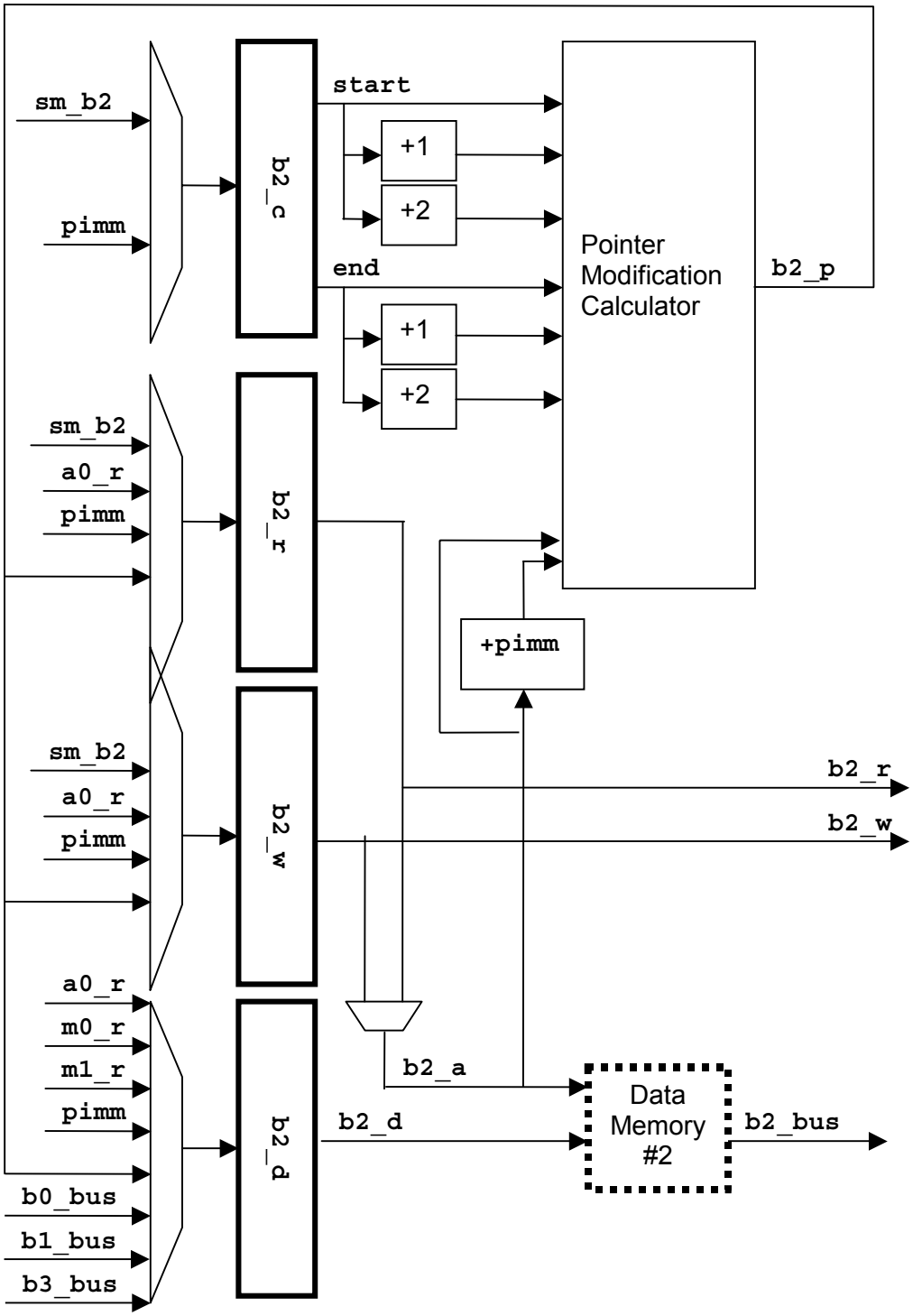


Figure 3.9 – B2 ACU Block Diagram

3.10.1 ACU2 C register

description	token(s)
the ACU2 c register pair (circular start and end addresses)	`b2_c`
the ACU2 circular end address register, available only for testing in the conditional (“if”) instruction	`b2_e`

Table 3.73 – ACU2 C Register Syntax

description	token(s)
the program immediate register	`pimm`
Cached pointer data from SM	`sm`
typical use (see 4.2.4.7.11)	`b2_c(&sm)`

Table 3.74 – ACU2 C Register Inputs

3.10.2 ACU2 R register

description	token(s)
the ACU2 r register (read ptr)	`b2_r`

Table 3.75 – ACU2 R Register Syntax

description	token(s)
the program immediate register	`pimm`
cached pointer data from SM	`sm`
the ALU0 register bus	`a0_r`
pointer as modified by ACU2	`b2_p`
typical use (see 4.2.4.7.11)	`b2_r(&sm)`

Table 3.76 – ACU2 R Register Inputs

3.10.3 ACU2 W register

description	token(s)
the ACU2 w register (write ptr)	`b2_w`

Table 3.77 – ACU2 W Register Syntax

description	token(s)
the program immediate register	`pimm`
cached pointer data from SM	`sm`
the ALU0 register bus	`a0_r`
pointer as modified by ACU2	`b2_p`
typical use (see 4.2.4.7.11)	`b2_w(&sm)`

Table 3.78 – ACU2 W Register Inputs

3.10.4 ACU2 D register

description	token(s)
the ACU2 d register (write data)	`b2_d`

Table 3.79 – ACU2 D Register Syntax

description	token(s)
the program immediate register	`pimm`
the ALU0 register bus	`a0_r`
the MAC0 register bus	`m0_r`
the MAC1 register bus	`m1_r`
pointer as modified by ACU2	`b2_p`
Data Memory Bus 0	`b0_bus`
Data Memory Bus 1	`b1_bus`
Data Memory Bus 3	`b3_bus`
typical use (see 4.2.4.7.11)	`b2_d(&pimm)`

Table 3.80 – ACU2 D Register Inputs

3.10.5 ACU2 Post-modified Pointer

description	token(s)
pointer as modified by ACU2	`b2_p`
typical use (see 4.2.4.8.7)	`b2_p = b2_w + 2`

Table 3.81 – ACU2 Post-modified Pointer

3.11 ACU 3 Data Memory & ACU Resources

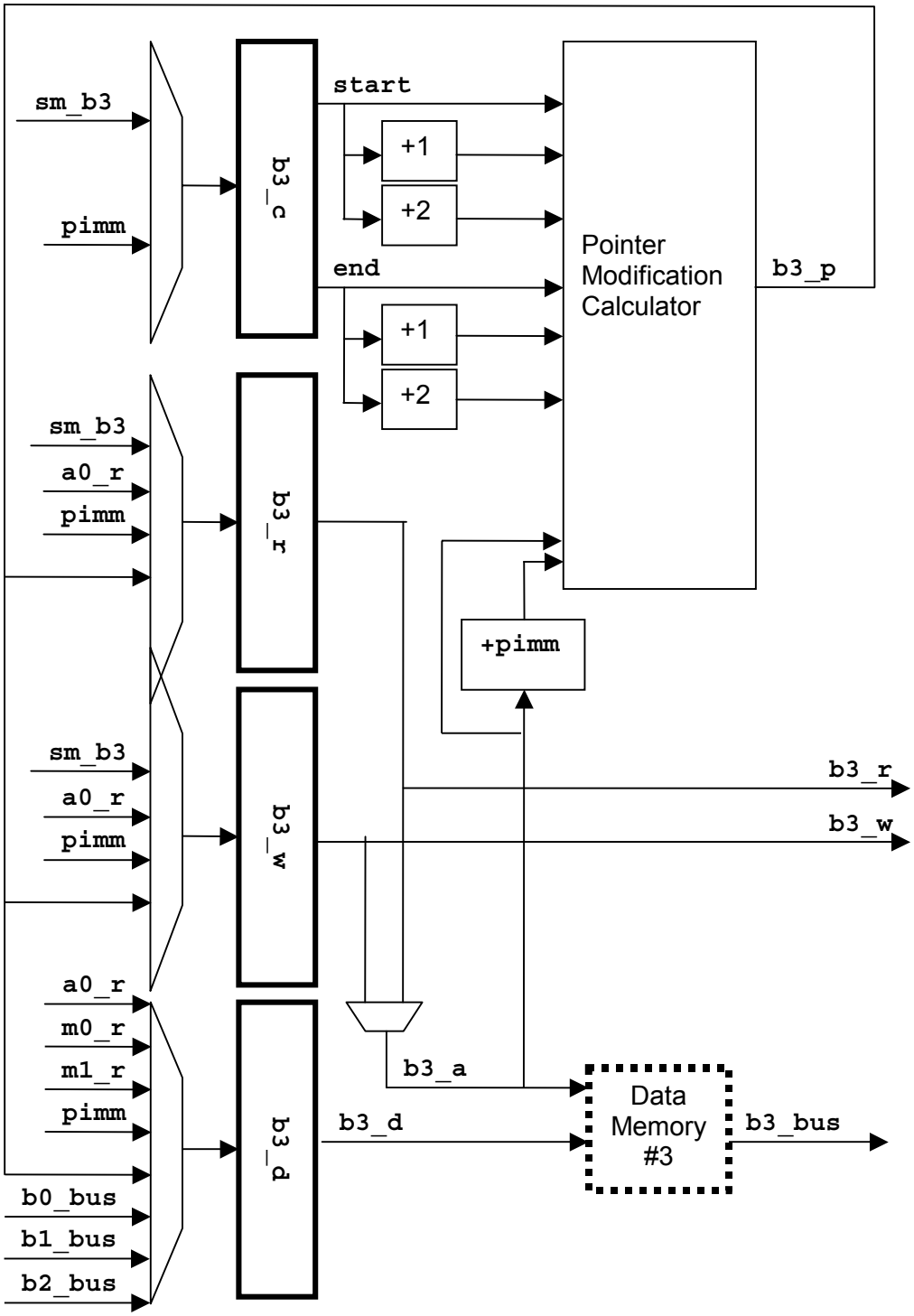


Figure 3.10 – B3 ACU Block Diagram

3.11.1 ACU3 C register

description	token(s)
the ACU3 c register pair (circular start and end addresses)	`b3_c`
the ACU3 circular end address register, available only for testing in the conditional (“if”) instruction	`b3_e`

Table 3.82 – ACU3 C Register Syntax

description	token(s)
the program immediate register	`pimm`
Cached pointer data from SM	`sm`
typical use (see 4.2.4.7.13)	`b3_c(&sm)`

Table 3.83 – ACU3 C Register Inputs

3.11.2 ACU3 R register

description	token(s)
the ACU3 r register (read ptr)	`b3_r`

Table 3.84 – ACU3 R Register Syntax

description	token(s)
the program immediate register	`pimm`
cached pointer data from SM	`sm`
the ALU0 register bus	`a0_r`
pointer as modified by ACU3	`b3_p`
typical use (see 4.2.4.7.13)	`b3_r(&sm)`

Table 3.85 – ACU3 R Register Inputs

3.11.3 ACU3 W register

description	token(s)
the ACU3 w register (write ptr)	`b3_w`

Table 3.86 – ACU3 W Register Syntax

description	token(s)
the program immediate register	`pimm`
cached pointer data from SM	`sm`
the ALU0 register bus	`a0_r`
pointer as modified by ACU3	`b3_p`
typical use (see 4.2.4.7.13)	`b3_w(&sm)`

Table 3.87 – ACU3 W Register Inputs

3.11.4 ACU3 D register

description	token(s)
the ACU3 d register (write data)	`b3_d`

Table 3.88 – ACU3 D Register Syntax

description	token(s)
the program immediate register	`pimm`
the ALU0 register bus	`a0_r`
the MAC0 register bus	`m0_r`
the MAC1 register bus	`m1_r`
pointer as modified by ACU3	`b3_p`
Data Memory Bus 0	`b0_bus`
Data Memory Bus 1	`b1_bus`
Data Memory Bus 2	`b2_bus`
typical use (see 4.2.4.7.13)	`b3_d(&pimm)`

Table 3.89 – ACU3 D Register Inputs

3.11.5 ACU3 Post-modified Pointer

description	token(s)
pointer as modified by ACU3	`b3_p`
typical use (see 4.2.4.8.8)	`b3_p = b3_w + 1`

Table 3.90 – ACU3 Post-modified Pointer

3.12 Pipeline Information

The DSP has a five stage pipeline operation, as illustrated below. An explanation follows.

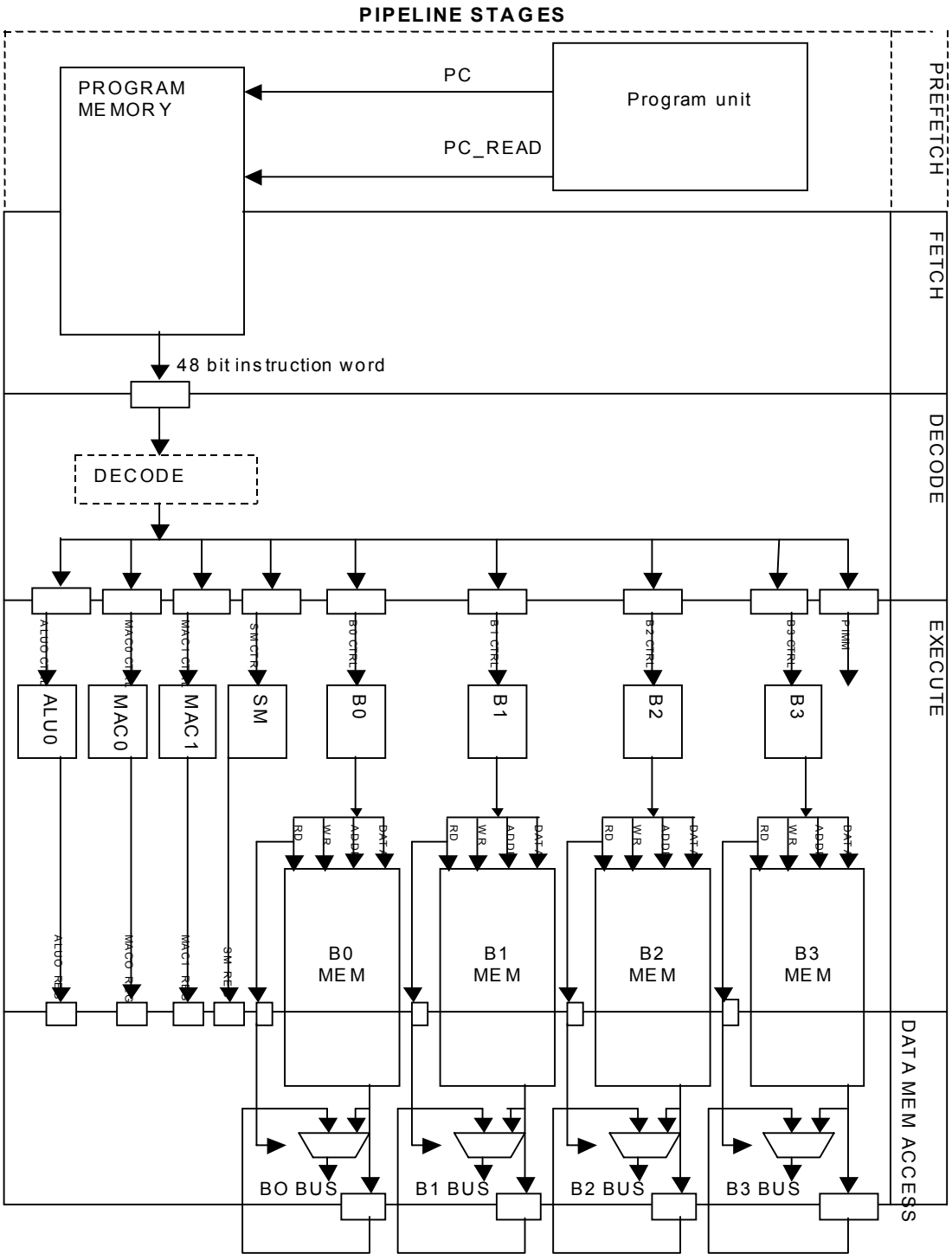


Figure 3.11 - Pipeline structure

3.12.1 Pre-Fetch Stage

The Pre-Fetch stage represents the opportunity for the assertion, by the Program Control Unit (PCU), of a valid Program Memory Address to the program memory.

The program unit controls the program fetch and issue and maintains the Program Counter (PC). Every cycle the program unit determines if there is a need to fetch a new instruction from the program memory. The program unit then asserts a read signal to the memory and uses the PC as the address to the memory. This read request is issued in a pipeline stage that is called “Pre-Fetch stage”.

3.12.2 FETCH Stage

The Fetch stage represents the opportunity for the assertion of valid Program Data by the program memory, and its capture into the Instruction Register of the PCU.

The memory has an inherent one cycle read latency. A read request issued in a [Pre-Fetch] cycle “N” will get the corresponding data from the memory in cycle “N+1”. In the “fetch stage” of the pipeline, the program memory outputs the instruction corresponding to the read request that happens in the previous “pre-fetch stage”.

3.12.3 DECODE Stage

The instruction fetched from the program memory in the “Fetch stage” of the pipeline is decoded in the “Decode stage”.

A separate “Decode Stage” enables NS85 DSP to determine to decode the instruction with no effect on the overall chip timing. There will be no decode related timing problems in the hardware chip implementation if complex instructions (requiring complex decodes) are added in the future versions of this chip.

3.12.4 EXECUTE Stage

The Execute stage of the pipeline is where all the functional operation is executed. All the units (ALU0, MAC0, MAC1, SM, B0, B1, B2, B3) execute the function that is specified by the decoded instruction. All aspects of the functions units operation are controlled during this stage.

Execute stage operations include the setting up of the selects for all the multiplexers in the functional units, as specified by the various source code “Register-See” clauses of the decoded instruction. At the end of this pipeline stage the appropriate registers in the functional units are updated based on the write enables specified in the decoded instruction. (The writes are specified at the source code level by the “Register Update” clause.) The execute stage also evaluates the branch condition to determine if there is going to be a branch mis-predict. The execute stage also issues the data memory access signals (read, write, address, write data) forward into the pipeline.

3.12.5 DMEM Stage

The DMEM stage represents the opportunity for the assertion of valid Data Memory data values on the various data busses (b0-b3) and its capture into the PCU.

The data memory has an inherent one cycle delay. A read request issued in a cycle “N” will get the corresponding data from the memory in cycle “N+1”. At the end of this pipeline stage we also register the data from the memory. The programmer can use the data that the memory returns in the DM pipeline stage itself. The data that is returned from the memory will be maintained till another access happens to the same data memory. A programmer can set up a read to the data memory in cycle “N” and use the data from cycle “N+1” till cycle “N+M” where M >= 2 as long as there is no new read request to memory. If the programmer does consecutive reads to the memory in cycle “N”, “N+1”,…, he would use the corresponding data from the memory in cycle “N+1, “N+2”…

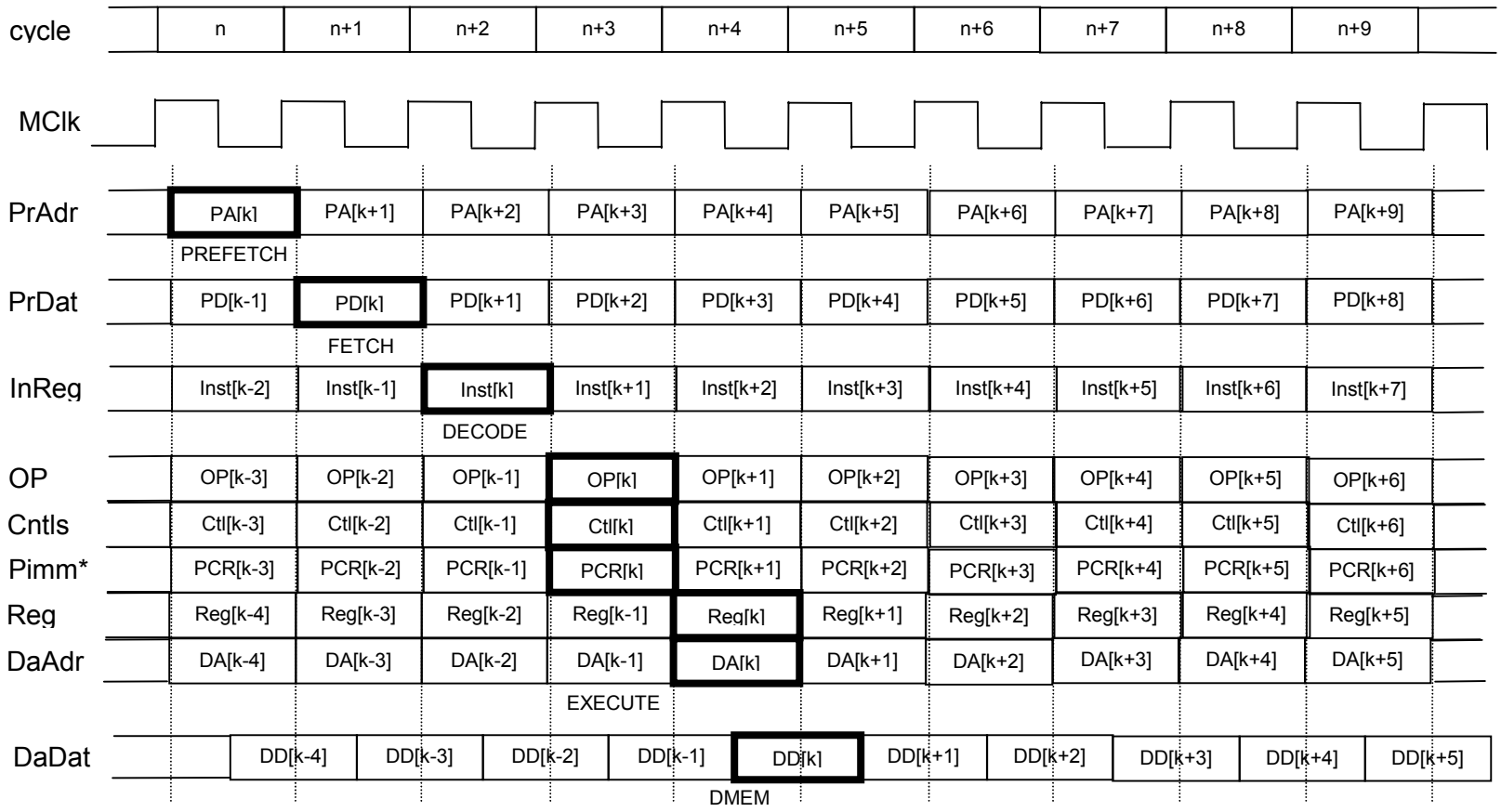
3.12.6 Pipeline Operation

In the ideal case, each of the pipeline stages is busy during a machine cycle: when some instruction is being decoded, the next instruction is being fetched from the program memory and the memory is getting the address for the instruction after that one; simultaneously the previous instruction to the one being decoded is executing, and the one before that is accessing data memory. There are some exceptions to that description, however.

In the case of a branch instruction where the branch is to be taken, part of the pipeline (which is a “branch not taken” based design) must be flushed and restarted on the new instruction stream. The branch penalty is 3 cycles.

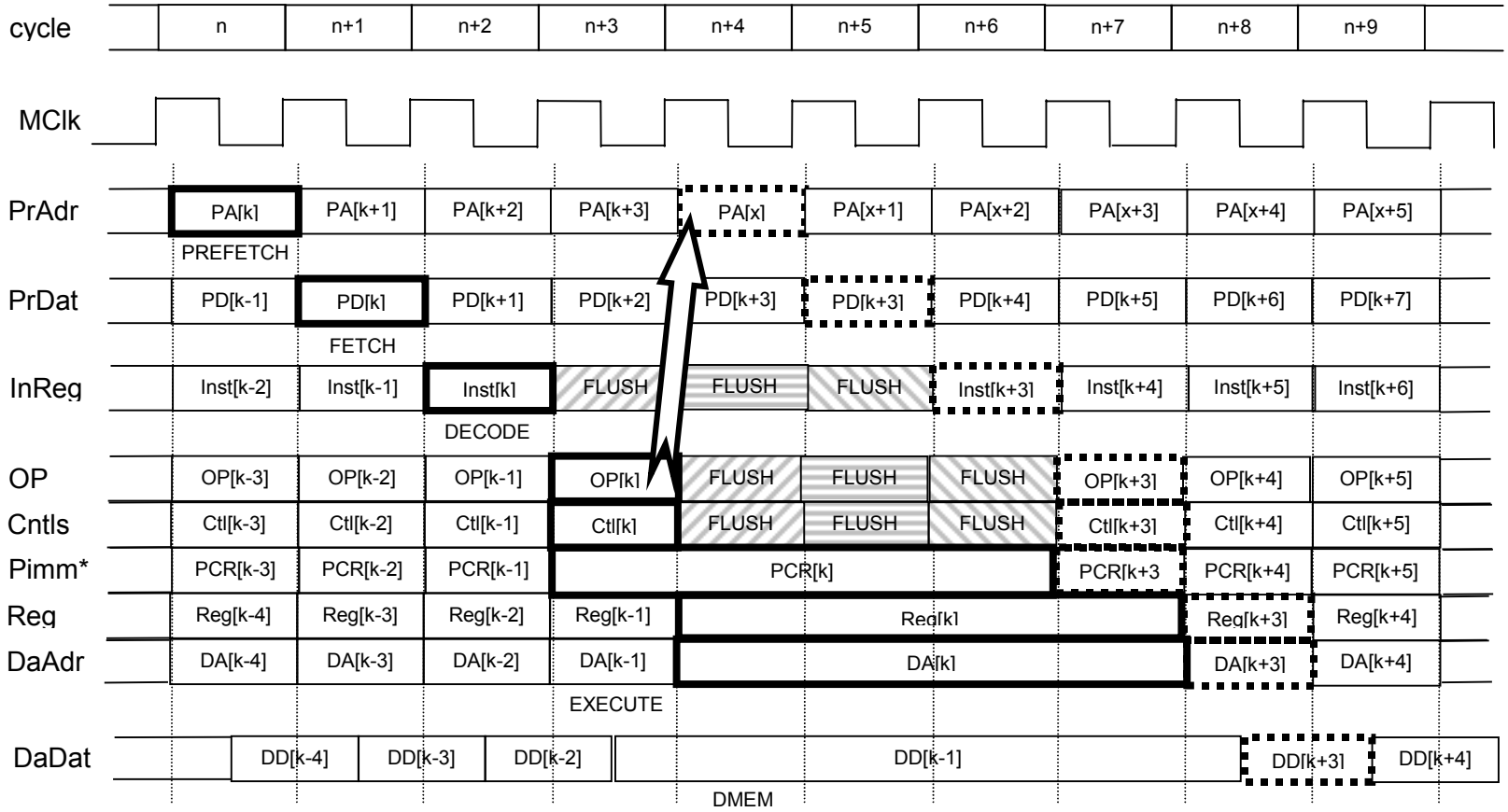
In the case of a repeat instruction, the PCU will stop the Pre-Fetch and Fetch stages until the repeat operation reaches the point of exhaustion and additional instructions will be required. The stopped stages are restarted in a manner which guarantees that there will be no bubbles in the pipeline.

So, what does it look like in time? The basic pipeline sequencing is illustrated in Figure 3.12, below. The pipeline in the case of a branch is illustrated in Figure 3.13. Figure 3.14 illustrates the pipeline as executes a repeated instruction.



* Pimm, Config & Repeat

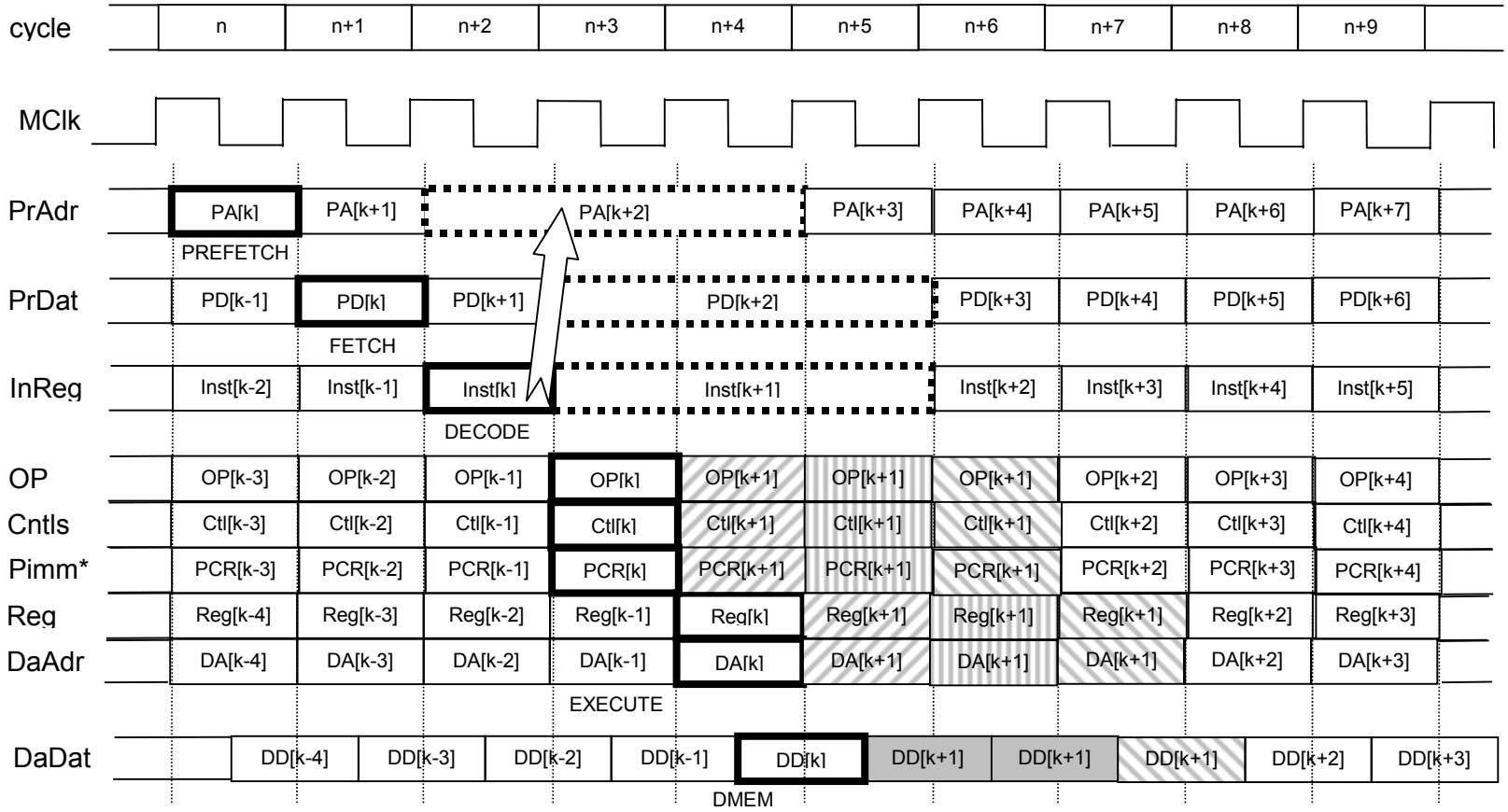
Figure 3.12 - Pipeline Operation



* Pimm, Config & Repeat

Inst[k]: `if (TRUE) pc = {x}; /* branch taken */`

Figure 3.13 – Effect of Branch on the Pipeline



* Pimm, Config & Repeat

Inst[k]: `repeat = 0x3;`

Figure 3.14 - Effect of Repeat on the Pipeline

3.12.7 Hazards

As shown, the PCU controls the pipeline to ensure proper activity at the machine level. Life is not all good and simple, however! While read cycles to and write cycles from data memory are managed by the PCU, the data utilised in those cycles is not pipelined by the machine. Instead they are fully managed by the programmer. The current instruction might include a write to memory of the data produced by the previous instruction and/or a pre-read of data from memory to a module input registers in preparation for the following instruction.

Because of the pipelined nature of the machine, there are hazards. What hazards are there in programming? Madness, at the very least. There are three types of machine hazards: control hazards, structural hazards and data hazards.

3.12.7.1 Control Hazards

Control hazards occur when instructions, such as branch which utilises the PC, are pipelined. This type of hazard was illustrated above, in Figure 3.13, and examination will show the loss of efficiency which results. The PCU deals with control hazards. The programmer should be aware of them. Remember: this machine as designed is a “branch not taken” architecture; taking a branch means taking a penalty.

3.12.7.2 Structural Hazards

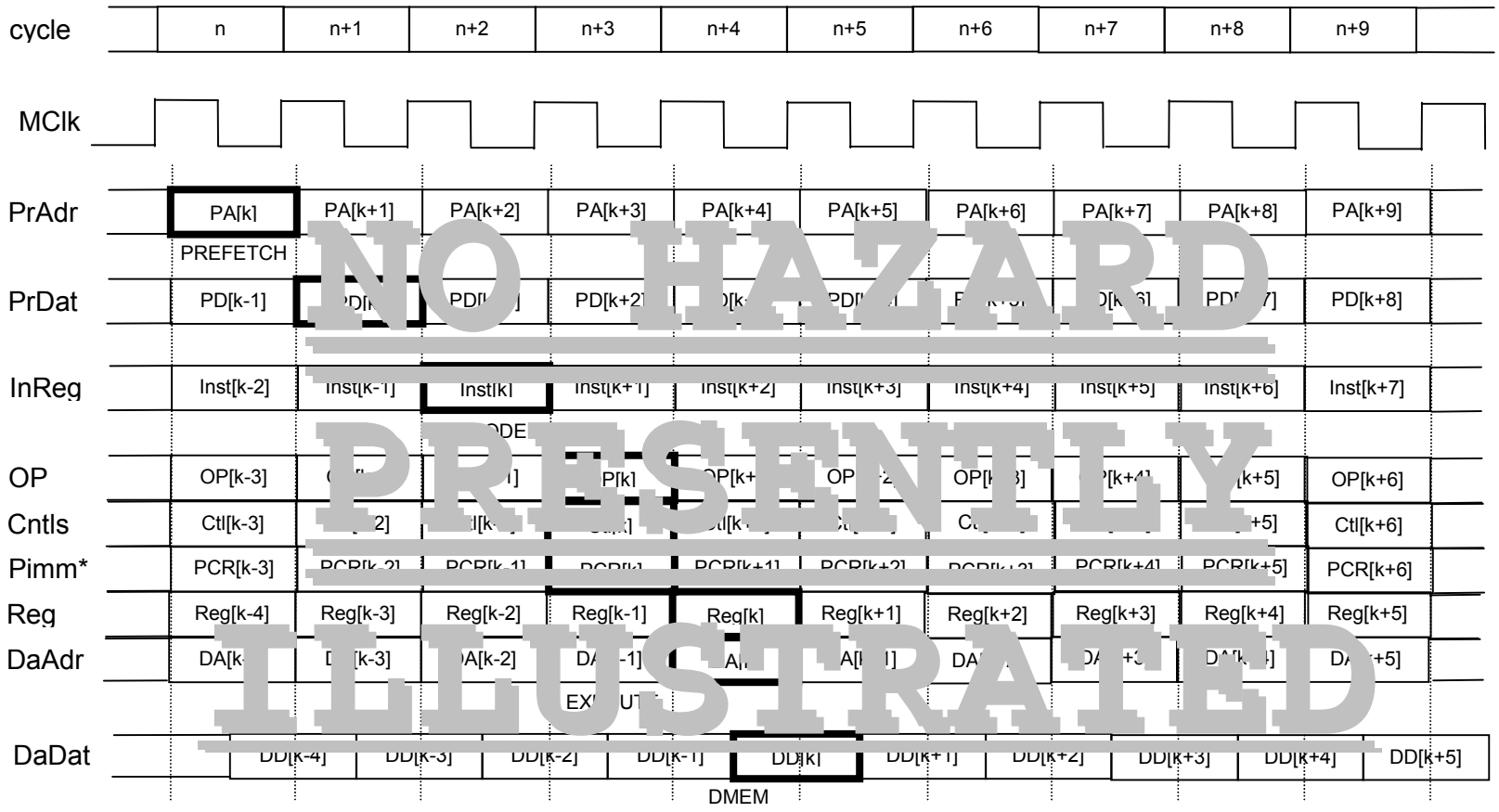
Structural hazards are cases where hardware resource conflicts arise because of pipelining. E.g. pushing the PC onto the stack in the same cycle that a value is being popped from the stack represents a structural hazard: generally stacks cannot be pushed and popped at the same time. The architecture and instruction set of the DSP have been designed to prevent, or at least minimize, structural hazards. No structural hazard is presently illustrated in Figure 3.15. (The illustration will be updated later.)

As structural hazards are identified, the assembler may be upgraded provide some help.

3.12.7.3 Data Hazards

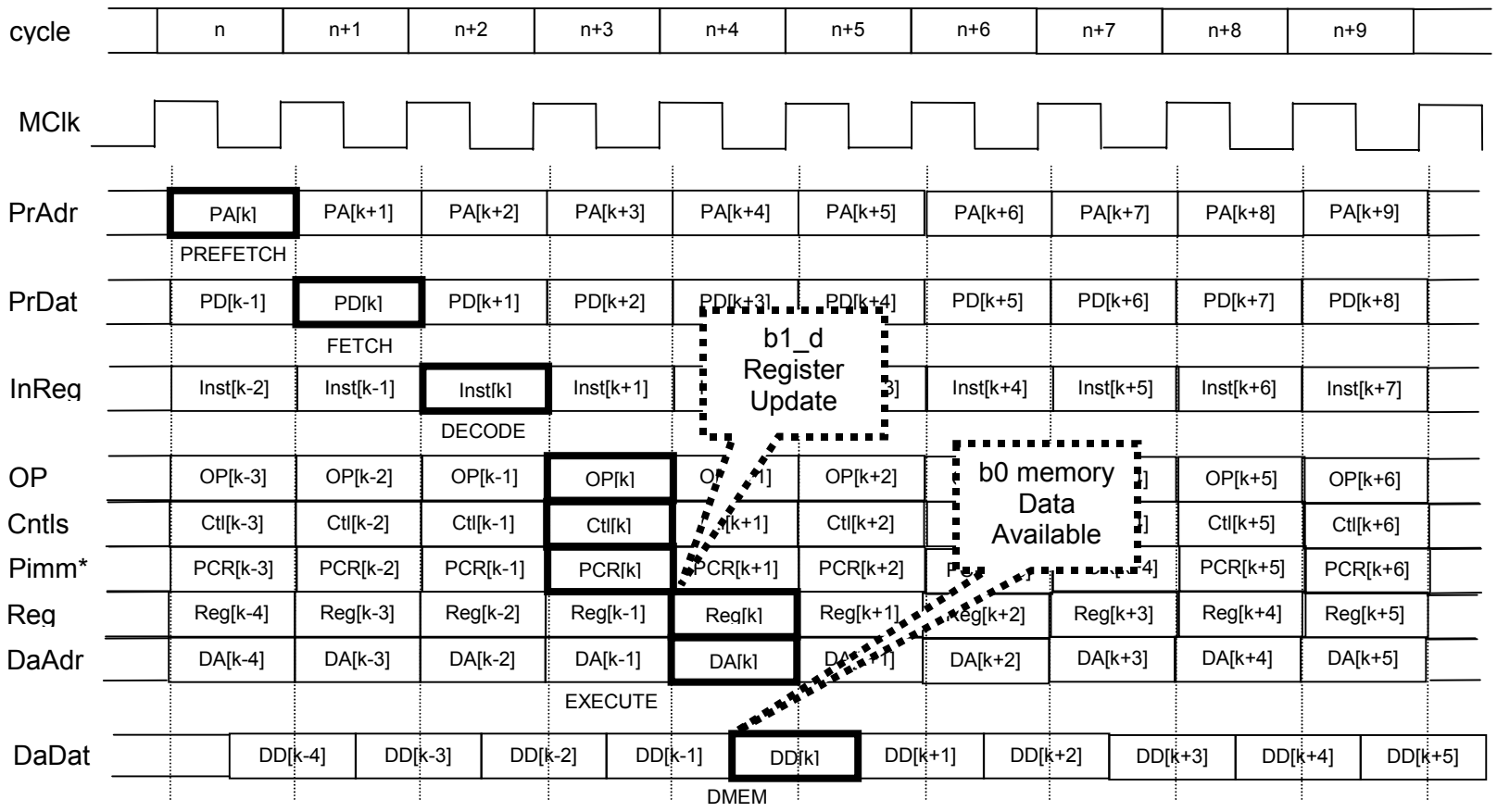
Data hazards involve interactions between or within instructions with respect to data. They arise when the pipelined operation of one or two instructions causes data timing issues to occur. E.g. attempting to read a data item before it has been written by a previous instruction. (Note: such data hazards can be intentionally exploited by the programmer in the optimization of the code.) In this DSP, data hazards arise primarily because the machine registers are updated at the end of the Execute cycle and memory data is valid one cycle later, at the end of the DMEM cycle.

The hazard is not presently illustrated in Figure 3.16. (The illustration will be updated later.) The programmer is fully responsible for avoiding or otherwise resolving, ”data hazards” by careful programming.



* Pimm, Config & Repeat

Figure 3.15 - Structural Hazard Illustration



* Pimm, Config & Repeat

Inst[k]:
```
b0_d(&pimm), b0_bus=*b0_r, b0_p = b0_r + 1,
b1_d(&b0_bus), nop, b1_d(next);
```

Figure 3.16 - Data Hazard Illustration

3.13 Instruction Types

The instructions which deal with the global operation of the DSP are categorized as "Control Instructions" and are loosely grouped into the "Configuration Instructions" and "Program Flow Instructions" sub-categories.

Instructions that configure or utilise either the data memories or the SM are called "Data Flow Control Instructions".

The compute modules (ALU0, MAC0, and MAC1) operate in parallel. However, to limit the instruction width and due to data bus limitations, some compromises have been made, the result being that not all conceivable parallel operations are allowed. Instructions that control the operation of the CU core compute modules are generally categorized as "Compute Instructions". The instructions are further sub-grouped according to the compute modules involved.

4 Syntax & Language Constructs

4.1 Introduction

The DSP uses a truncated VLIW architecture controlled by 48 bit instructions. There are no on-chip resource management facilities: the programmer is responsible for all aspects of execution. With this in mind, the assembly language of the NS85 DSP has been designed to ease the task of the programmers: the creation, debugging, and documentation of the code.

The syntax of the NS85 DSP assembly language is based on the 'c' programming language. It must be stressed, however, that it is ***not*** 'c'. The language has the following characteristics:

- Comments, both the c-style (bracketed with "`/*`" and "`*/`" tokens) and the c++ style (from the "`//`" token to the end-of-line) are allowable between instructions. NB: comments are not presently allowed inside instructions – massive quantities of comments are expected to be found between instructions.

- Symbolics are now supported: address references may be provided as identifiers or hexadecimal numbers.

- An "identifier" must start with an alpha character or an underscore character [a-zA-Z_] and subsequent characters can be alpha characters, underscores, or numerals [a-zA-Z_0-9].

- A program has no blocking or scoping.

- Instructions are terminated by a semicolon '`;`' character.

- The '`,`' comma character is generally used to separate clauses within the instruction. NB: however that the comma is also used to separate sub-clauses within clauses.

- Clauses consisting of like items are generally insensitive to the ordering of items within the clause.

- Whitespace (tabs, spaces, and carriage returns) are generally syntactically insignificant (NB: generally, not strictly: the end of line is significant as part of a c++ style comment!)

- An RTL (Register Transfer Language) style approach provides maximal visibility into the program activity at the hardware resource level.

4.2 Syntax

4.2.1 Program

A program is made up of comments and instructions. There is no blocking or scoping: everything is global and must be managed by the programmer.

4.2.2 Comments

Comments are very important and should be generously provided. The language supports both the c-style and the c++ style comments. E.g.:

```
/* this is a c-style comment
       which occupies multiple lines.
   it starts with a slash-star structure
       and terminates with a star-slash. */

// this is a c++ style comment: it extends to the end-of-line)
```

Comments are placed between instructions. **NB**: *comments are not presently allowed inside instructions*.

4.2.3 Instructions

Instructions consist of a selection of clauses, which are explained below. Instructions are terminated with a semicolon '**;**' character.

Instructions are not restricted to a single line. Whitespace (i.e. space characters, tab characters, newline characters, etc.) is of no syntactic relevance (except in the case of the c++ style comment as noted above.) The experienced programmer will liberally use whitespace to make the program easier to read and maintain.

4.2.4 Instruction clauses

Instruction clauses consist of like items grouped together within an instruction. Clauses are *generally* separated from each other by comma '**,**' characters. However, the comma character does not uniquely identify the end of a clause: the items within a clause are also often separated by commas.

The first clause in every instruction is a Label Define Clause. Subsequent clauses are identified by function. Clauses which control the selection of input data to registers are called Register-See Clauses. Operation Clauses specify the particular function to be performed. The writing of new data to the registers is specified by an Update Clause.

Generally, the instruction clauses for a hardware module follow the form:

```
<see_clause> <operation_clause> <update_clause>
```

Instructions often control the operation of multiple hardware modules that operate in parallel. When multiple modules are involved, the instruction parts for the modules are not intermixed in free-form; rather they are grouped together. E.g.: **<clauses for module a> <clauses for module b>;**.

4.2.4.1 Label Define Clause

A Label Define Clause is used to assign a convenient label or tag to an address in memory. It is the first clause in any instruction. A Label Define Clause is not separated from other clauses by a comma ',' character. The separator is a space ' ' character instead.

A Label Define Clause takes one of two forms. The first form consists of an identifier followed by a colon ':' character, e.g.

```
my_label42:
```

The second form is empty: no identifier and no colon ':' character. Consider:

```
nop();
nop();
```

Each of those two instructions has a Label Define Clause: an *empty* Label Define Clause.

4.2.4.2 Label Use Clause

A Label Use Clause is employed to make use of a convenient label or tag in an instruction. A Label Use Clause consists of an identifier, which is elsewhere specified in a Label Define Clause, or a constant value. These example instructions each include a Label Use Clause:

```
pc = my_start;
pc = 0x1234; // magic numbers are usually bad.
if (a0gt != TRUE) pc = agc_loop_top;
pimm = my_buf_start,
          b0_r(&pimm),nop(),b0_r(next), b1_r(&sm),nop();
call(my_sub);
```

4.2.4.3 Immediate Clause

An immediate clause is written as a direct assignment of a constant value, or an expression which resolves to a constant (e.g. a Label Use Clause in cases where a memory address is apropos), into an allowable DSP resource. These resources include:

`pc`	the program counter register
`pimm`	the program immediate register
`config`	the configuration register
`iterate0`	Block iteration count register #0
`iterate1`	Block iteration count register #1
`repeat`	the program instruction repeat counter

Some instructions in the "Control Instructions" group consist of a single immediate clause. Examples of these cases, where the immediate clauses are the whole instruction, follow:

```
pc = 0x9876;
repeat = 0x12;
config = 0xDEAD;
if (a0gt != TRUE) pc = 0xFADE;
```

4.2.4.4 Pimm Clause

Some of the instructions operate using immediate data as part of a larger instruction. The part of such a complex instruction which is apropos to the immediate data is called a Pimm Clause and the syntactic form will always be that of an assignment of the immediate data to the **pimm** register. E.g.:

```
pimm = 0x1234, . . .
```

4.2.4.5 Configuration clause

The global operation of the various modules is controlled by the contents of the **config** register. When writing the program instruction to properly load the register, an immediate value is embedded in the program code. Such a collection of 1's and 0's is not very convenient, however. To remedy this, the syntax allows the programmer to use mnemonic tokens for the various configuration control constants. In the collecting of these tokens, the programmer's choice of plus '**+**' or bar '**|**' characters are for joining with a logical OR function. (There is merit in consistency.) The ampersand '**&**' character joins with a logical AND function. Precedence is strictly left to right. The resulting construct is called a "Configuration clause" in the grammar and elsewhere in this document. An example configuration clause is found after the equal sign in this instruction:

```
config = a0(sat) + m0(fract) | m1(int);
```

(See also 3.3.7 Configuration Register and 25 Configuration Register load instruction.)

4.2.4.6 DMA Configuration clause

The operation of the DMA function, getting data from the A/D module and putting data to the D/A module, is controlled by the contents of the **dmaconfig** register. When writing the program instruction to properly load the register, an immediate value is embedded in the program code. The syntax allows the programmer to use mnemonic tokens for the various control constants. In the collecting of these tokens, the programmer's choice of plus '**+**' or bar '**|**' characters are for joining with a logical OR function. (There is merit in consistency.) The ampersand '**&**' character joins with a logical AND function. Precedence is strictly left to right. The resulting construct is called a "DMA Configuration clause" in the grammar and elsewhere in this document. An example DMA configuration clause is found after the equal sign in this instruction:

```
dmaconfig = see_dma(on) + dab(1) + adb(1) + daen(on) + aden(on);
```

(See also 3.3.6_DMA Configuration Register and 26_DMA Configuration Register load instruction.)

4.2.4.7 Register-See Clause

Many of the DSP registers take their input from a selection of sources under program control. The configuration information for all of the registers in a specific module is grouped together in a Register-See clause. This clause is used to express the desired configuration of such input multiplexers: in effect to tell what each of the input registers of a module will see.

A Register-See clause consists of one or more items which specify the inputs for a register. These register item sub-clauses are separated by comma '**,**' characters. The order of the register items within a Register-See clause is unimportant.

Syntactically, the Register-See clause is made to look like a c-function call. It consists of the name of the register as the function, with the "address of" the desired data source appearing as a parameter to the function. The following is an example of a register-see clause for the ALU:

```
a0_x(&pimm), a0_y(&m0_r), a0_s(&a0), . . .
```

4.2.4.7.1 ALU0 Register-See Clause

The ALU0 Register-See clause may contain any of the following tokens in any order:

token	action
a0_x(&pimm)	Select immediate value (pimm register or DMA write address, selected by bit 0 of dmaconfig registe) as the input to the ALU0 x register
a0_x(&a0)	Select the output of ALU0 as the input to the ALU0 x register
a0_x(&b0_bus)	Select Data Memory Bus 0 as the input to the ALU0 x register
a0_x(&b1_bus)	Select Data Memory Bus 1 as the input to the ALU0 x register
a0_x(&b2_bus)	Select Data Memory Bus 2 as the input to the ALU0 x register
a0_x(&b3_bus)	Select Data Memory Bus 3 as the input to the ALU0 x register
a0_x(&m0_r)	Select the register bus from MAC0 as the input to the ALU0 x register
a0_x(&m1_r)	Select the register bus from MAC1 as the input to the ALU0 x register

Table 4.1 - Input Selection for ALU0 X Register

token	action
`a0_y(&pimm)`	Select immediate value (pimm register or DMA read address, selected by bit 0 of dmaconfig register) as the input to the ALU0 y register
`a0_y(&a0)`	Select the output of ALU0 as the input to the ALU0 y register
`a0_y(&b0_bus)`	Select Data Memory Bus 0 as the input to the ALU0 y register
`a0_y(&b1_bus)`	Select Data Memory Bus 1 as the input to the ALU0 y register
`a0_y(&b2_bus)`	Select Data Memory Bus 2 as the input to the ALU0 y register
`a0_y(&b3_bus)`	Select Data Memory Bus 3 as the input to the ALU0 y register
`a0_y(&m0_r)`	Select the register bus from MAC0 as the input to the ALU0 y register
`a0_y(&m1_r)`	Select the register bus from MAC1 as the input to the ALU0 y register

Table 4.2 - Input Selection for ALU0 Y Register

token	action
`a0_s(&pimm)`	Select PIMM as the input to the ALU0 s register
`a0_s(&(long)pimm)`	Select sign-extended PIMM as the input to the ALU0 s register
`a0_s(&b0_bus)`	Select b0_bus as the input to the ALU0 s register
`a0_s(&a0)`	Select the output of ALU0 as the input to the ALU0 s register

Table 4.3 - Input Selection for ALU0 S Register

token	action
a0_f(&pimm)	Select PIMM as the input to the ALU0 flags register
a0_f(&a0)	Select the output of ALU0 as the input to the ALU0 flags register

Table 4.4 - Input Selection for ALU0 Flags Register

token	action
a0_r(&a0_xh)	Select the ALU0 xh register as the source for the ALU0 register bus (a0_r)
a0_r(&a0_xl)	Select the ALU0 xl register as the source for the ALU0 register bus (a0_r)
a0_r(&a0_yh)	Select the ALU0 yh register as the source for the ALU0 register bus (a0_r)
a0_r(&a0_yl)	Select the ALU0 yl register as the source for the ALU0 register bus (a0_r)
a0_r(&a0_sh)	Select the ALU0 sh register as the source for the ALU0 register bus (a0_r)
a0_r(&a0_sl)	Select the ALU0 sl register as the source for the ALU0 register bus (a0_r)
a0_r(&a0_f)	Select the ALU0 flags register as the source for the ALU0 register bus (a0_r)

Table 4.5 - Input Selection for ALU0 Register Bus

4.2.4.7.2 MAC0 Register-See Clause

The MAC0 Register-See clause may contain any of the following tokens in any order:

token	action
m0_x(&pimm)	Select PIMM as the input to the MAC0 x register
m0_x(&b0_bus)	Select Data Memory Bus 0 as the input to the MAC0 x register
m0_x(&b1_bus)	Select Data Memory Bus 1 as the input to the MAC0 x register
m0_x(&b2_bus)	Select Data Memory Bus 2 as the input to the MAC0 x register
m0_x(&b3_bus)	Select Data Memory Bus 3 as the input to the MAC0 x register
m0_x(&m1_r)	Select the register bus from MAC1 as the input to the MAC0 x register
m0_x(&a0_r)	Select the register bus from ALU0 as the input to the MAC0 x register

Table 4.6 - Input Selection for MAC0 X Register

token	action
`m0_y(&pimm)`	Select PIMM as the input to the MAC0 y register
`m0_y(&b0_bus)`	Select Data Memory Bus 0 as the input to the MAC0 y register
`m0_y(&b1_bus)`	Select Data Memory Bus 1 as the input to the MAC0 y register
`m0_y(&b2_bus)`	Select Data Memory Bus 2 as the input to the MAC0 y register
`m0_y(&b3_bus)`	Select Data Memory Bus 3 as the input to the MAC0 y register
`m0_y(&m1_r)`	Select the register bus from MAC1 as the input to the MAC0 y register
`m0_y(&a0_r)`	Select the register bus from ALU0 as the input to the MAC0 y register

Table 4.7 - Input Selection for MAC0 Y Register

token	action
`m0_s(&(long)pimm)`	Select sign-extended PIMM as the input to the MAC0 s register
`m0_s(&b_bus)`	Select the memory bus catenation {b2_bus[7:0],b1_bus[15:0],b0_bus[15:0]} as the input to the MAC0 s register
`m0_s(&m0)`	Select output of MAC0 as the input to the MAC0 s register
`m0_s(&a0_r)`	Select the output of ALU0 register bus as the input to the MAC0 s register

Table 4.8 - Input Selection for MAC0 S Register

token	action
`m0_f(&pimm)`	Select PIMM as the input to the MAC0 flags register
`m0_f(&m0)`	Select the output of MAC0 as the input to the MAC0 flags register

Table 4.9 - Input Selection for MAC0 Flags Register

token	action
`m0_r(&m0_x)`	Select the MAC0 x register as the source for the MAC0 register bus (m0_r)
`m0_r(&m0_y)`	Select the MAC0 y register as the source for the MAC0 register bus (m0_r)
`m0_r(&m0_sh)`	Select the high half of the MAC0 s register as the source for the MAC0 register bus (m0_r)
`m0_r(&m0_sl)`	Select the low half of the MAC0 s register as the source for the MAC0 register bus (m0_r)
`m0_r(&m0_sg)`	Select the guard bits of the MAC0 s register as the source for the MAC0 register bus (m0_r)
`m0_r(&(int)m0_sg)`	Select the sign extended guard bits of the MAC0 s register as the source for the MAC0 register bus (m0_r)
`m0_r(&m0_f)`	Select the MAC0 flags register as the source for the MAC0 register bus (m0_r)

Table 4.10 - Input Selection for MAC0 Register Bus

4.2.4.7.3 MAC1 Register-See Clause

The MAC1 Register-See clause may contain any of the following tokens in any order:

token	action
`m1_x(&pimm)`	Select PIMM as the input to the MAC1 x register
`m1_x(&b0_bus)`	Select Data Memory Bus 0 as the input to the MAC1 x register
`m1_x(&b1_bus)`	Select Data Memory Bus 1 as the input to the MAC1 x register
`m1_x(&b2_bus)`	Select Data Memory Bus 2 as the input to the MAC1 x register
`m1_x(&b3_bus)`	Select Data Memory Bus 3 as the input to the MAC1 x register
`m1_x(&m0_r)`	Select the register bus from MAC0 as the input to the MAC1 x register
`m1_x(&a0_r)`	Select the register bus from ALU0 as the input to the MAC1 x register

Table 4.11 - Input Selection for MAC1 X Register

token	action
m1_y(&pimm)	Select PIMM as the input to the MAC1 y register
m1_y(&b0_bus)	Select Data Memory Bus 0 as the input to the MAC1 y register
m1_y(&b1_bus)	Select Data Memory Bus 1 as the input to the MAC1 y register
m1_y(&b2_bus)	Select Data Memory Bus 2 as the input to the MAC1 y register
m1_y(&b3_bus)	Select Data Memory Bus 3 as the input to the MAC1 y register
m1_y(&m0_r)	Select the register bus from MAC0 as the input to the MAC1 y register
m1_y(&a0_r)	Select the register bus from ALU0 as the input to the MAC1 y register

Table 4.12 - Input Selection for MAC1 Y Register

token	action
m1_s(&(long)pimm)	Select sign-extended PIMM as the input to the MAC1 s register
m1_s(&b_bus)	Select the memory bus catenation {b2_bus[7:0],b1_bus[15:0],b0_bus[15:0]} as the input to the MAC1 s register
m1_s(&m0)	Select output of MAC1 as the input to the MAC1 s register
m1_s(&a0_r)	Select the output of ALU0 register bus as the input to the MAC1 s register

Table 4.13 - Input Selection for MAC1 S Register

token	action
m1_f(&pimm)	Select PIMM as the input to the MAC1 flags register
m1_f(&m1)	Select the output of MAC1 as the input to the MAC1 flags register

Table 4.14 - Input Selection for MAC1 Flags Register

token	action
m1_r(&m1_x)	Select the MAC1 x register as the source for the MAC1 register bus (m1_r)
m1_r(&m1_y)	Select the MAC1 y register as the source for the MAC1 register bus (m1_r)
m1_r(&m1_sh)	Select the high half of the MAC1 s register as the source for the MAC1 register bus (m1_r)
m1_r(&m1_sl)	Select the low half of the MAC1 s register as the source for the MAC1 register bus (m1_r)
m1_r(&m1_sg)	Select the guard bits of the MAC1 s register as the source for the MAC1 register bus (m1_r)
m1_r(&(int)m1_sg)	Select the sign extended guard bits of the MAC1 s register as the source for the MAC1 register bus (m1_r)
m1_r(&m1_f)	Select the MAC1 flags register as the source for the MAC1 register bus (m1_r)

Table 4.15 - Input Selection for MAC1 Register Bus

4.2.4.7.4 SM Register-See Clause

The SM Register-See clause may contain allowable tokens in any order. Note that input selections are limited in several ways by hardware. The s, e, r, and w registers require an "all or none" selection of data from the SM. When the SM is not selected, the s and e register inputs are configured to select the pimm register. The r and w register inputs are selected by a single, common control, thus the source code selection must be consistent. The SM Register-See clause includes the SM B0123 Register-See Clause and uses the following additional tokens:

token	action
`sm_s(&pimm)`	Select PIMM as the input to the SM s register set
`sm_s(&sm)`	Select memory data from SM as the input to the SM s (and all other s/e/r/w) register

Table 4.16 - Input Selection for SM S Register

token	action
`sm_e(&pimm)`	Select PIMM as the input to the SM e register set
`sm_e(&sm)`	Select memory data from SM as the input to the SM e (and all other s/e/r/w) register

Table 4.17 - Input Selection for SM E Register

token	action
sm_r(&pimm)	Select PIMM as the input to the SM r register set
sm_r(&sm)	Select memory data from SM as the input to the SM r (and all other s/e/r/w) register
sm_r(&b0_r)	Select Data Memory Bus 0 ACU read pointer as the input to the SM r register
sm_r(&b1_r)	Select Data Memory Bus 1 ACU read pointer as the input to the SM r register
sm_r(&b2_r)	Select Data Memory Bus 2 ACU read pointer as the input to the SM r register
sm_r(&b3_r)	Select Data Memory Bus 3 ACU read pointer as the input to the SM r register

Table 4.18 - Input Selection for ACU0 R Register

token	action
sm_w(&pimm)	Select PIMM as the input to the SM w register set
sm_w(&sm)	Select memory data from SM as the input to the SM_w (and all other s/e/r/w) registers
sm_w(&b0_w)	Select Data Memory Bus 0 ACU write pointer as the input to the SM w register
sm_w(&b1_w)	Select Data Memory Bus 1 ACU write pointer as the input to the SM w register
sm_w(&b2_w)	Select Data Memory Bus 2 ACU write pointer as the input to the SM w register
sm_w(&b3_w)	Select Data Memory Bus 3 ACU write pointer as the input to the SM w register

Table 4.19 - Input Selection for SM W Register

4.2.4.7.5 SM B0123 Register-See Clause

token	action
`sm_b0(&sm_serw[0])`	Select cached pointer set #0 to be returned from the SM to ACU0
`sm_b0(&sm_serw[1])`	Select cached pointer set #1 to be returned from the SM to ACU0
`sm_b0(&sm_serw[2])`	Select cached pointer set #2 to be returned from the SM to ACU0
`sm_b0(&sm_serw[3])`	Select cached pointer set #3 to be returned from the SM to ACU0
`sm_b0(&sm_serw[4])`	Select cached pointer set #4 to be returned from the SM to ACU0
`sm_b0(&sm_serw[5])`	Select cached pointer set #5 to be returned from the SM to ACU0
`sm_b0(&sm_serw[6])`	Select cached pointer set #6 to be returned from the SM to ACU0
`sm_b0(&sm_serw[7])`	Select cached pointer set #7 to be returned from the SM to ACU0

Table 4.20 - Pointer Selection for SM_B0 Return

token	action
`sm_b1(&sm_serw[0])`	Select cached pointer set #0 to be returned from the SM to ACU1
`sm_b1(&sm_serw[1])`	Select cached pointer set #1 to be returned from the SM to ACU1
`sm_b1(&sm_serw[2])`	Select cached pointer set #2 to be returned from the SM to ACU1
`sm_b1(&sm_serw[3])`	Select cached pointer set #3 to be returned from the SM to ACU1
`sm_b1(&sm_serw[4])`	Select cached pointer set #4 to be returned from the SM to ACU1
`sm_b1(&sm_serw[5])`	Select cached pointer set #5 to be returned from the SM to ACU1
`sm_b1(&sm_serw[6])`	Select cached pointer set #6 to be returned from the SM to ACU1
`sm_b1(&sm_serw[7])`	Select cached pointer set #7 to be returned from the SM to ACU1

Table 4.21 - Pointer Selection for SM_B1 Return

token	action
`sm_b2(&sm_serw[0])`	Select cached pointer set #0 to be returned from the SM to ACU2
`sm_b2(&sm_serw[1])`	Select cached pointer set #1 to be returned from the SM to ACU2
`sm_b2(&sm_serw[2])`	Select cached pointer set #2 to be returned from the SM to ACU2
`sm_b2(&sm_serw[3])`	Select cached pointer set #3 to be returned from the SM to ACU2
`sm_b2(&sm_serw[4])`	Select cached pointer set #4 to be returned from the SM to ACU2
`sm_b2(&sm_serw[5])`	Select cached pointer set #5 to be returned from the SM to ACU2
`sm_b2(&sm_serw[6])`	Select cached pointer set #6 to be returned from the SM to ACU2
`sm_b2(&sm_serw[7])`	Select cached pointer set #7 to be returned from the SM to ACU2

Table 4.22 - Pointer Selection for SM_B2 Return

token	action
`sm_b3(&sm_serw[0])`	Select cached pointer set #0 to be returned from the SM to ACU3
`sm_b3(&sm_serw[1])`	Select cached pointer set #1 to be returned from the SM to ACU3
`sm_b3(&sm_serw[2])`	Select cached pointer set #2 to be returned from the SM to ACU3
`sm_b3(&sm_serw[3])`	Select cached pointer set #3 to be returned from the SM to ACU3
`sm_b3(&sm_serw[4])`	Select cached pointer set #4 to be returned from the SM to ACU3
`sm_b3(&sm_serw[5])`	Select cached pointer set #5 to be returned from the SM to ACU3
`sm_b3(&sm_serw[6])`	Select cached pointer set #6 to be returned from the SM to ACU3
`sm_b3(&sm_serw[7])`	Select cached pointer set #7 to be returned from the SM to ACU3

Table 4.23 - Pointer Selection for SM_B3 Return

4.2.4.7.6 ACU 0 ACU Register-See Clause

The ACU0 Register-See clause includes the tokens in the ACU 0 Pointer Register-See Clause and the following tokens in any order:

token	action
b0_d(&pimm)	Select PIMM as the input to the ACU0 d register
b0_d(&a0_r)	Select ALU0 register bus as the input to the ACU0 d register
b0_d(&m0_r)	Select MAC0 register bus as the input to the ACU0 d register
b0_d(&m1_r)	Select MAC1 register bus as the input to the ACU0 d register
b0_d(&b0_p)	Select the modified pointer in ACU0 as the input to the ACU0 d register
b0_d(&b1_bus)	Select the Data Memory #1 bus as the input to the ACU0 d register
b0_d(&b2_bus)	Select the Data Memory #2 bus as the input to the ACU0 d register
b0_d(&b3_bus)	Select the Data Memory #3 bus as the input to the ACU0 d register

Table 4.24 - Input Selection for ACU0 D Register

4.2.4.7.7 ACU 0 Pointer Register-See Clause

The ACU0 Pointer Register-See clause may contain any (note the exception regarding r and w registers) of the following tokens in any order:

token	action
b0_c(&pimm)	Select PIMM as the input to the ACU0 c register set
b0_c(&sm)	Select cached pointer data from SM as the input to the ACU0 c register set

Table 4.25 - Input Selection for ACU0 C Register

token	action
b0_r(&pimm)	Select PIMM as the input to the ACU0 r register
b0_r(&sm)	Select cached pointer data from SM as the input to the ACU0 r register
b0_r(&a0_r)	Select ALU0 register bus as the input to the ACU0 r register
b0_r(&b0_p)	Select the modified pointer in ACU0 as the input to the ACU0 r register

Table 4.26 - Input Selection for ACU0 R Register

NB: though the update controls are independent, the inputs of the b0_r register and the b0_w register use the same selector, thus only one of the two types of input selection tokens is allowed in an ACU 0 Pointer Register-See Clause.

token	action
b0_w(&pimm)	Select PIMM as the input to the ACU0 w register
b0_w(&sm)	Select cached pointer data from SM as the input to the ACU0 w register
b0_w(&a0_r)	Select ALU0 register bus as the input to the ACU0 w register
b0_w(&b0_p)	Select the modified pointer in ACU0 as the input to the ACU0 w register

Table 4.27 - Input Selection for ACU0 W Register

NB: though the update controls are independent, the inputs of the b0_r register and the b0_w register use the same selector, thus only one of the two types of input selection tokens is allowed in an ACU 0 Pointer Register-See Clause.

4.2.4.7.8 ACU 1 Register-See Clause

The ACU1 Register-See clause includes the tokens in the ACU 1 Pointer Register-See Clause and the following tokens in any order:

token	action
b1_d(&pimm)	Select PIMM as the input to the ACU1 d register
b1_d(&a0_r)	Select ALU0 register bus as the input to the ACU1 d register
b1_d(&m0_r)	Select MAC0 register bus as the input to the ACU1 d register
b1_d(&m1_r)	Select MAC1 register bus as the input to the ACU1 d register
b1_d(&b1_p)	Select the modified pointer in ACU1 as the input to the ACU1 d register
b1_d(&b0_bus)	Select the Data Memory #0 bus as the input to the ACU1 d register
b1_d(&b2_bus)	Select the Data Memory #2 bus as the input to the ACU1 d register
b1_d(&b3_bus)	Select the Data Memory #3 bus as the input to the ACU1 d register

Table 4.28 - Input Selection for ACU1 D Register

4.2.4.7.9 ACU 1 Pointer Register-See Clause

The ACU1 Pointer Register-See clause may contain any (note the exception regarding r and w registers) of the following tokens in any order:

token	action
b1_c(&pimm)	Select PIMM as the input to the ACU1 c register set
b1_c(&sm)	Select cached pointer data from SM as the input to the ACU1 c register set

Table 4.29 - Input Selection for ACU1 C Register

token	action
b1_r(&pimm)	Select PIMM as the input to the ACU1 r register
b1_r(&sm)	Select cached pointer data from SM as the input to the ACU1 r register
b1_r(&a0_r)	Select ALU0 register bus as the input to the ACU1 r register
b1_r(&b1_p)	Select the modified pointer in ACU1 as the input to the ACU1 r register

Table 4.30 - Input Selection for ACU1 R Register

NB: though the update controls are independent, the inputs of the b1_r register and the b1_w register use the same selector, thus only one of the two types of input selection tokens is allowed in an ACU 1 Pointer Register-See Clause.

token	action
b1_w(&pimm)	Select PIMM as the input to the ACU1 w register
b1_w(&sm)	Select cached pointer data from SM as the input to the ACU1 w register
b1_w(&a0_r)	Select ALU0 register bus as the input to the ACU1 w register
b1_w(&b1_p)	Select the modified pointer in ACU1 as the input to the ACU1 w register

Table 4.31 - Input Selection for ACU1 W Register

NB: though the update controls are independent, the inputs of the b1_r register and the b1_w register use the same selector, thus only one of the two types of input selection tokens is allowed in an ACU 1 Pointer Register-See Clause.

4.2.4.7.10 ACU 2 Register-See Clause

The BUS2 Register-See clause includes the tokens in the ACU 2 Pointer Register-See Clause and the following tokens in any order:

token	action
`b2_d(&pimm)`	Select PIMM as the input to the ACU2 d register
`b2_d(&a0_r)`	Select ALU0 register bus as the input to the ACU2 d register
`b2_d(&m0_r)`	Select MAC0 register bus as the input to the ACU2 d register
`b2_d(&m1_r)`	Select MAC1 register bus as the input to the ACU2 d register
`b2_d(&b2_p)`	Select the modified pointer in ACU2 as the input to the ACU2 d register
`b2_d(&b0_bus)`	Select the Data Memory #0 bus as the input to the ACU2 d register
`b2_d(&b1_bus)`	Select the Data Memory #1 bus as the input to the ACU2 d register
`b2_d(&b3_bus)`	Select the Data Memory #3 bus as the input to the ACU2 d register

Table 4.32 - Input Selection for ACU2 D Register

4.2.4.7.11 ACU 2 Pointer Register-See Clause

The ACU2 Pointer Register-See clause may contain any (note the exception regarding r and w registers) of the following tokens in any order:

token	action
`b2_c(&pimm)`	Select PIMM as the input to the ACU2 c register set
`b2_c(&sm)`	Select cached pointer data from SM as the input to the ACU2 c register set

Table 4.33 - Input Selection for ACU2 C Register

token	action
b2_r(&pimm)	Select PIMM as the input to the ACU2 r register
b2_r(&sm)	Select cached pointer data from SM as the input to the ACU2 r register
b2_r(&a0_r)	Select ALU0 register bus as the input to the ACU2 r register
b2_r(&b2_p)	Select the modified pointer in ACU2 as the input to the ACU2 r register

Table 4.34 - Input Selection for ACU2 R Register

NB: though the update controls are independent, the inputs of the b2_r register and the b2_w register use the same selector, thus only one of the two types of input selection tokens is allowed in an ACU 2 Pointer Register-See Clause.

token	action
b2_w(&pimm)	Select PIMM as the input to the ACU2 w register
b2_w(&sm)	Select cached pointer data from SM as the input to the ACU2 w register
b2_w(&a0_r)	Select ALU0 register bus as the input to the ACU2 w register
b2_w(&b2_p)	Select the modified pointer in ACU2 as the input to the ACU2 w register

Table 4.35 - Input Selection for ACU2 W Register

NB: though the update controls are independent, the inputs of the b2_r register and the b2_w register use the same selector, thus only one of the two types of input selection tokens is allowed in an ACU 2 Pointer Register-See Clause.

4.2.4.7.12 ACU 3 Register-See Clause

The ACU3 Register-See clause includes the tokens in the ACU 3 Pointer Register-See Clause and the following tokens in any order:

token	action
`b3_d(&pimm)`	Select PIMM as the input to the ACU3 d register
`b3_d(&a0_r)`	Select ALU0 register bus as the input to the ACU3 d register
`b3_d(&m0_r)`	Select MAC0 register bus as the input to the ACU3 d register
`b3_d(&m1_r)`	Select MAC1 register bus as the input to the ACU3 d register
`b3_d(&b3_p)`	Select the modified pointer in ACU3 as the input to the ACU3 d register
`b3_d(&b0_bus)`	Select the Data Memory #0 bus as the input to the ACU3 d register
`b3_d(&b1_bus)`	Select the Data Memory #1 bus as the input to the ACU3 d register
`b3_d(&b2_bus)`	Select the Data Memory #2 bus as the input to the ACU3 d register

Table 4.36 - Input Selection for ACU3 D Register

4.2.4.7.13 ACU 3 Pointer Register-See Clause

The ACU3 Pointer Register-See clause may contain any (note the exception regarding r and w registers) of the following tokens in any order:

token	action
b3_c(&pimm)	Select PIMM as the input to the ACU3 c register set
b3_c(&sm)	Select cached pointer data from SM as the input to the ACU3 c register set

Table 4.37 - Input Selection for ACU3 C Register

token	action
b3_r(&pimm)	Select PIMM as the input to the ACU3 r register
b3_r(&sm)	Select cached pointer data from SM as the input to the ACU3 r register
b3_r(&a0_r)	Select ALU0 register bus as the input to the ACU3 r register
b3_r(&b3_p)	Select the modified pointer in ACU3 as the input to the ACU3 r register

Table 4.38 - Input Selection for ACU3 R Register

NB: though the update controls are independent, the inputs of the b3_r register and the b3_w register use the same selector, thus only one of the two types of input selection tokens is allowed in an ACU 3 Pointer Register-See Clause.

token	action
`b3_w(&pimm)`	Select PIMM as the input to the ACU3 w register
`b3_w(&sm)`	Select cached pointer data from SM as the input to the ACU3 w register
`b3_w(&a0_r)`	Select ALU0 register bus as the input to the ACU3 w register
`b3_w(&b3_p)`	Select the modified pointer in ACU3 as the input to the ACU3 w register

Table 4.39 - Input Selection for ACU3 W Register

NB: though the update controls are independent, the inputs of the b3_r register and the b3_w register use the same selector, thus only one of the two types of input selection tokens is allowed in an ACU 3 Pointer Register-See Clause.

4.2.4.8 Operation Clause

An Operation clause expresses the operation which is to be performed by a specific module. The clause is separated from other clauses by a comma ‘,’ character. An Operation clause is generally a very clear c-like expression that details the inputs and the desired action for the module. For example, an ALU-MAC instruction would contain two Operation clauses (one for each module) and might look like this:

```
. . . (a0_x & a0_y), . . . (m0_x * m0_y), . . .
```

Operations in ALU0 and MAC0 and MAC1 determine special result conditions which ***can be*** latched into the flag registers (a0_f, m0_f, and m1_f) of the respective units. These conditions come in two flavours: transient and sticky. Transient conditions are evaluated for truth/falsehood on a cycle by cycle basis. The sticky conditions are evaluated for truth/falsehood on a cycle by cycle basis and logically OR’d with the prior value. The flags are updated only if the register is indicated in the update clause of the instruction. Clearing a sticky bit in a flag register is accomplished by loading from the `pimm` register.

4.2.4.8.1 ALU0 Operation Clause

The operation to be performed by ALU0 is specified by this clause. Note that the **`nop()`** operation is a fiction of the assembler: the unit will perform a fixed operation but updates of the **`a0_f`** (flags) or **`a0_s`** (result) registers is not allowed.

The special result conditions which can be captured to a0_f are:

condition	name	**`symbol`**
Overflow on bit 16 or bit 31, depending on operation	Overflow	**`ov`**
Sticky overflow on bit 16 or bit 31, depending on operation. Can be reset by only by software.	Sticky overflow	**`sov`**
The result is zero.	Zero	**`z`**
Sticky zero condition.	Sticky zero	**`sz`**
The result is greater than or equal to zero.	Greater or equal	**`ge`**
Sticky greater than or equal to condition.	Sticky greater or equal	**`sge`**
The result is greater than zero.	Greater than	**`gt`**
Sticky greater than condition.	Sticky greater than	**`sgt`**
The result generated a carry out from bit 15 or 31, depending on operation	Carry	**`c`**
Sticky carry condition	Sticky carry	**`sc`**

Table 4.40 - ALU0 Result Conditions

Not all operations show these results: only the **`z`** (and **`sz`**) conditions are produced by all ALU0 operations. Hence, those special result conditions apropos to each operation are indicated with the individual operation description, appearing as a vector, e.g. **`{gt,z,c}`**. The “sticky” condition is implied by the normal form of the condition, so this form is not explicitly noted. Some ALU0 operations always result in a special condition of a fixed value. For such cases, the fixed value will be noted in the vector, appearing as an assign: e.g. **`{gt,z,c=0}`**.

These are the operations which may be performed by ALU0:

token	action
nop()	No operation is specified
(a0_xl + a0_yl)	Add the contents of the ALU0 xl and yl registers (short). **{ov,z,ge,gt,c}**
(a0_xl - a0_yl)	Subtract the contents of the ALU0 yl from the xl register (short). **{ov,z,ge,gt,c}**
(a0_x + a0_y)	Add the contents of the ALU0 x and y registers (long). **{ov,z,ge,gt,c}**
(a0_x - a0_y)	Subtract the contents of the ALU0 y from the x register (long). **{ov,z,ge,gt,c}**
(a0_s + a0_xl)	Add the contents of the ALU0 xl and s registers (long). **{ov,z,ge,gt,c}**
(a0_s & 0)	Set the ALU0 result to zero (long). **{ov,z,ge,gt,c}**
divp()	Perform division partial primitive (long). **{ov,z,ge,gt,c}**

Table 4.41 - Operation Selection for ALU0 (Arithmatic)

token	action
`(a0_xl & a0_yl)`	AND the contents of the ALU0 xl and yl registers (short). `{ov=0,z,ge,gt,c=0}`
`(a0_xl | a0_yl)`	OR the contents of the ALU0 xl and yl registers (short). `{ov=0,z,ge,gt,c=0}`
`(a0_xl ^ a0_yl)`	XOR the contents of the ALU0 xl and yl registers (short). `{ov=0,z,ge,gt,c=0}`
`(~ a0_s)`	Invert (1's complement) the contents of the ALU0 s register (long). `{ov,z,ge,gt,c=0}`
`abs(a0_xl)`	Take the absolute value of the contents of the ALU0 xl register (short). `{ov,z,ge,gt,c=0}`

Table 4.42 - Operation Selection for ALU0 (Logical)

token	action
`(arith)(a0_xl << a0_y[3:0])`	Perform an arithmetic left shift on the contents of the ALU0 xl register, with the shift amount specified by the low 4 bits of the ALU0 y register. `{ov,z,ge,gt,c=0}`
`(logic)(a0_xl << a0_y[3:0])`	Perform a logical left shift on the contents of the ALU0 xl register, with the shift amount specified by the low 4 bits of the ALU0 y register. `{ov,z,ge,gt,c=0}`
`(arith)(a0_xl >> a0_y[3:0])`	Perform an arithmetic right shift on the contents of the ALU0 xl register, with the shift amount specified by the low 4 bits of the ALU0 y register. `{ov=0,z,ge,gt,c=0}`
`(logic)(a0_xl >> a0_y[3:0])`	Perform a logical right shift on the contents of the ALU0 xl register, with the shift amount specified by the low 4 bits of the ALU0 y register. `{ov=0,z,ge,gt,c=0}`
`(a0_x[a0_y[3:0]])`	Test the bit of the ALU0 x register that is specified by the low 4 bits of the ALU0 y register (0 selects LSB) and set the flags as appropriate. `{ov=0,z,ge=1,gt,c=0}`

Table 4.43 - Operation Selection for ALU0 (Bits & Short Shifts)

NB: the a0_sl register is the nominal target for short shift instructions. I.e. the short shift instructions generate results apropos to the a0_sl register. If it is desired to update a0_sh as well, a long shift instruction should be used.

token	action
`(arith)(a0_x << a0_y[3:0])`	Perform an arithmetic left shift on the contents of the ALU0 x register, with the shift amount specified by the low 4 bits of the ALU0 y register. `{ov=0,z,ge,gt,c=0}`
`(logic)(a0_x << a0_y[3:0])`	Perform a logical left shift on the contents of the ALU0 x register, with the shift amount specified by the low 4 bits of the ALU0 y register. `{ov=0,z,ge,gt,c=0}`
`(arith)(a0_x >> a0_y[3:0])`	Perform an arithmetic right shift on the contents of the ALU0 x register, with the shift amount specified by the low 4 bits of the ALU0 y register. `{ov=0,z,ge,gt,c=0}`
`(logic)(a0_x >> a0_y[3:0])`	Perform a logical right shift on the contents of the ALU0 x register, with the shift amount specified by the low 4 bits of the ALU0 y register. `{ov=0,z,ge,gt,c=0}`

Table 4.44 - Operation Selection for ALU0 (Long Shifts)

4.2.4.8.2 MAC0 Operation Clause

The operation to be performed by MAC0 is specified by this clause. Note that the `nop()` operation is a fiction of the assembler: the unit will perform a fixed operation but updates of the `m0_f` (flags) or `m0_f` (result) registers is not allowed.

The special result conditions which can be captured to m0_f are:

condition	name	`symbol`
Overflow on bit 31	Overflow	`ov`
Sticky overflow on bit 31. Can be reset by only by software.	Sticky overflow	`sov`
40 bit Overflow on bit 39	Overflow	`eov`
Sticky 40 bit overflow on bit 39. Can be reset by only by software.	Sticky overflow	`seov`

Table 4.45 - MAC0 Result Conditions

Special result conditions apropos to each operation are indicated with the individual operation description, appearing as a vector, e.g. **{eov,ov}**. The "sticky" condition is implied by the normal form of the condition, so this form is not explicitly noted.

token	action
nop()	No operation is specified
(m0_x * m0_y)	Multiply the contents of the MAC0 x and y registers. **{ov}**
-(m0_x * m0_y)	Multiply the contents of the MAC0 x and y registers and change the sign of the result. **{ov}**
m0_s[39:0] + (m0_x * m0_y)	Multiply the contents of the MAC0 x and y registers and add the 40 bit contents of the MAC0 s register. **{eov,ov}**
m0_s[39:0] - (m0_x * m0_y)	Multiply the contents of the MAC0 x and y registers and subtract the result from the 40 bit contents of the MAC0 s register. **{eov,ov}**

Table 4.46 - Operation Selection for MAC0

4.2.4.8.3 MAC1 Operation Clause

The operation to be performed by MAC1 is specified by this clause. Note that the **nop()** operation is a fiction of the assembler: the unit will perform a fixed operation but updates of the **m1_f** (flags) or **m1_s** (result) registers is not allowed.

The special result conditions which can be captured to m1_f are:

condition	name	**symbol**
Overflow on bit 31	Overflow	**ov**
Sticky overflow on bit 31. Can be reset by only by software.	Sticky overflow	**sov**
40 bit Overflow on bit 39	Overflow	**eov**
Sticky 40 bit overflow on bit 39. Can be reset by only by software.	Sticky overflow	**seov**

Table 4.47 – MAC1 Result Conditions

Special result conditions apropos to each operation are indicated with the individual operation description, appearing as a vector, e.g. `{eov,ov}`. The “sticky” condition is implied by the normal form of the condition, so this form is not explicitly noted.

token	Action
`nop()`	No operation is specified
`(m1_x * m1_y)`	Multiply the contents of the MAC1 x and y registers. `{ov}`
`-(m1_x * m1_y)`	Multiply the contents of the MAC1 x and y registers and change the sign of the result. `{ov}`
`m1_s[39:0] + (m1_x * m1_y)`	Multiply the contents of the MAC1 x and y registers and add the 40 bit contents of the MAC1 s register. `{eov,ov}`
`m1_s[39:0] - (m1_x * m1_y)`	Multiply the contents of the MAC1 x and y registers and subtract the result from the 40 bit contents of the MAC1 s register. `{eov,ov}`

Table 4.48 - Operation Selection for MAC1

4.2.4.8.4 SM Operation Clause

The operation to be performed by the SM is specified by this clause.

token	action
sm_i = *{idx}*, **sm_a = pimm**	Load the SM I register with the value *{idx}* (restricted to 0-7) and load the SM a register from the **pimm** register; no memory access occurs
sm_i = *{idx}*, **sm = *sm_a++**	Load the SM I register with the value *{idx}* (restricted to 0-7) and **read** the SM RAM at the address in the a register; post-increment the a register.
sm_i = *{idx}*, **sm = *sm_a--**	Load the SM I register with the value *{idx}* (restricted to 0-7) and **read** the SM RAM at the address in the a register; post-decrement the a register.
sm_i = *{idx}*, ***sm_a++ = sm**	Load the SM I register with the value *{idx}* (restricted to 0-7) and **write** the sm_b2 data to the SM RAM at the address in the a register; post-increment the a register.
sm_i = *{idx}*,***sm_a-- = sm**	Load the SM I register with the value *{idx}* (restricted to 0-7) and **write** the sm_b2 data to the SM RAM at the address in the a register; post-decrement the a register.

Table 4.49 - Operation Selection for SM

4.2.4.8.5 ACU0 Operation Clause

The operation to be performed by Data Memory Bus 0 ACU is specified by this clause.

token	action
`nop()`	No operation
`nop(), b0_p = b0_w + 1`	Perform no memory operation, but calculate a modified pointer by adding 1 to the value of b0_w
`*b0_w = b0_d, b0_p = b0_w + 1`	Write the b0_d data to the memory addressed by b0_w, calculate a modified pointer by adding 1 to the value of b0_w
`*b0_w = b0_d, b0_p = b0_w + 2`	Write the b0_d data to the memory addressed by b0_w, calculate a modified pointer by adding 2 to the value of b0_w
`*b0_w = b0_d, b0_p = b0_w + pimm`	Write the b0_d data to the memory addressed by b0_w, calculate a modified pointer by adding the contents of the **pimm** register to the value of b0_w
`b0_bus = *b0_r, b0_p = b0_r + 1`	read the data memory addressed by b0_r to the b0_bus, calculate a modified pointer by adding 1 to the value of b0_r
`b0_bus = *b0_r, b0_p = b0_r + 2`	read the data memory addressed by b0_r to the b0_bus, calculate a modified pointer by adding 2 to the value of b0_r
`b0_bus = *b0_r, b0_p = b0_r + pimm`	read the data memory addressed by b0_r to the b0_bus, calculate a modified pointer by adding the contents of the **pimm** register to the value of b0_r

Table 4.50 - Operation Selection for ACU0

4.2.4.8.6 ACU1 Operation Clause

The operation to be performed by Data Memory Bus 1 ACU is specified by this clause.

token	action
`nop()`	No operation
`nop(), b1_p = b1_w + 1`	Perform no memory operation, but calculate a modified pointer by adding 1 to the value of b1_w
`*b1_w = b1_d, b1_p = b1_w + 1`	Write the b1_d data to the memory addressed by b1_w, calculate a modified pointer by adding 1 to the value of b1_w
`*b1_w = b1_d, b1_p = b1_w + 2`	Write the b1_d data to the memory addressed by b1_w, calculate a modified pointer by adding 2 to the value of b1_w
`*b1_w = b1_d, b1_p = b1_w + pimm`	Write the b1_d data to the memory addressed by b1_w, calculate a modified pointer by adding the contents of the **pimm** register to the value of b1_w
`b1_bus = *b1_r, b1_p = b1_r + 1`	read the data memory addressed by b1_r to the b1_bus, calculate a modified pointer by adding 1 to the value of b1_r
`b1_bus = *b1_r, b1_p = b1_r + 2`	read the data memory addressed by b1_r to the b1_bus, calculate a modified pointer by adding 2 to the value of b1_r
`b1_bus = *b1_r, b1_p = b1_r + pimm`	read the data memory addressed by b1_r to the b1_bus, calculate a modified pointer by adding the contents of the **pimm** register to the value of b1_r

Table 4.51 - Operation Selection for ACU1

4.2.4.8.7 ACU2 Operation Clause

The operation to be performed by Data Memory Bus 2 ACU is specified by this clause.

token	action
`nop()`	No operation
`nop(), b2_p = b2_w + 1`	Perform no memory operation, but calculate a modified pointer by adding 1 to the value of b2_w
`*b2_w = b2_d, b2_p = b2_w + 1`	Write the b2_d data to the memory addressed by b2_w, calculate a modified pointer by adding 1 to the value of b2_w
`*b2_w = b2_d, b2_p = b2_w + 2`	Write the b2_d data to the memory addressed by b2_w, calculate a modified pointer by adding 2 to the value of b2_w
`*b2_w = b2_d, b2_p = b2_w + pimm`	Write the b2_d data to the memory addressed by b2_w, calculate a modified pointer by adding the contents of the **pimm** register to the value of b2_w
`b2_bus = *b2_r, b2_p = b2_r + 1`	read the data memory addressed by b2_r to the b2_bus, calculate a modified pointer by adding 1 to the value of b2_r
`b2_bus = *b2_r, b2_p = b2_r + 2`	read the data memory addressed by b2_r to the b2_bus, calculate a modified pointer by adding 2 to the value of b2_r
`b2_bus = *b2_r, b2_p = b2_r + pimm`	read the data memory addressed by b2_r to the b2_bus, calculate a modified pointer by adding the contents of the **pimm** register to the value of b2_r

Table 4.52 - Operation Selection for ACU2

4.2.4.8.8 ACU3 Operation Clause

The operation to be performed by Data Memory Bus 3 ACU is specified by this clause.

token	action
`nop()`	No operation
`nop(), b3_p = b3_w + 1`	Perform no memory operation, but calculate a modified pointer by adding 1 to the value of b3_w
`*b3_w = b3_d, b3_p = b3_w + 1`	Write the b3_d data to the memory addressed by b3_w, calculate a modified pointer by adding 1 to the value of b3_w
`*b3_w = b3_d, b3_p = b3_w + 2`	Write the b3_d data to the memory addressed by b3_w, calculate a modified pointer by adding 2 to the value of b3_w
`*b3_w = b3_d, b3_p = b3_w + pimm`	Write the b3_d data to the memory addressed by b3_w, calculate a modified pointer by adding the contents of the **pimm** register to the value of b3_w
`b3_bus = *b3_r, b3_p = b3_r + 1`	read the data memory addressed by b3_r to the b3_bus, calculate a modified pointer by adding 1 to the value of b3_r
`b3_bus = *b3_r, b3_p = b3_r + 2`	read the data memory addressed by b3_r to the b3_bus, calculate a modified pointer by adding 2 to the value of b3_r
`b3_bus = *b3_r, b3_p = b3_r + pimm`	read the data memory addressed by b3_r to the b3_bus, calculate a modified pointer by adding the contents of the **pimm** register to the value of b3_r

Table 4.53 - Operation Selection for ACU3

4.2.4.9 Update Clause

Update clauses exist to individually and independently specify which registers in the associated active module are to be updated during the current instruction cycle. An Update clause consists of zero or more comma separated items, each of which specifies a specific register to be updated. The various register sub-clauses are optional and may appear in any order.

Syntactically, the Update clause is made to look like a c-function call. It consists of the name of the register as the function, with the literal token “next” appearing as a parameter to the function. The following is an example of an Update clause which affects all the registers of MAC1:

```
m1_x(next), m1_y(next), m1_s(next), m1_f(next),
```

4.2.4.9.1 ALU0 Update Clause

The ALU0 Update clause may contain the following tokens in any order:

token	action
a0_xl(next)	Update low half of ALU0 x register
a0_xh(next)	Update high half of ALU0 x register
a0_x(next)	Update both halves of ALU0 x register
a0_yl(next)	Update low half of ALU0 y register
a0_yh(next)	Update high half of ALU0 y register
a0_y(next)	Update both halves of ALU0 y register
a0_sl(next)	Update low half of ALU0 s register
a0_sh(next)	Update high half of ALU0 s register
a0_s(next)	Update both halves of ALU0 s register
a0_f(next)	Update the ALU0 flags register

Table 4.54 - ALU0 Register Update

4.2.4.9.2 MAC0 Update Clause

The MAC0 Update clause may contain the following tokens in any order:

token	action
`m0_x(next)`	Update the MAC0 x register
`m0_y(next)`	Update the MAC0 y register
`m0_s(next)`	Update the MAC0 s register
`m0_f(next)`	Update the MAC0 flags register

Table 4.55 - MAC0 Register Update

4.2.4.9.3 MAC1 Update Clause

The MAC1 Update clause may contain the following tokens in any order:

token	action
`m1_x(next)`	Update the MAC1 x register
`m1_y(next)`	Update the MAC1 y register
`m1_s(next)`	Update the MAC1 s register
`m1_f(next)`	Update the MAC1 flags register

Table 4.56 - MAC1 Register Update

4.2.4.9.4 SM Update Clause

The SM Update clause may contain the following tokens in any order:

token	action
sm_s(next)	Update the SM s (circular start) register
sm_e(next)	Update the SM e (circular end) register
sm_r(next)	Update the SM r register
sm_w(next)	Update the SM w register
sm_a(next)	Update the SM a register

Table 4.57 - SM Register Update

4.2.4.9.5 ACU0 Update Clause

The ACU0 Update clause may contain the following tokens in any order:

token	action
b0_c(next)	Update the ACU0 c register set
b0_r(next)	Update the ACU0 r register
b0_w(next)	Update the ACU0 w register
b0_d(next)	Update the ACU0 d register

Table 4.58 – ACU0 Register Update

4.2.4.9.6 ACU1 Update Clause

The ACU1 Update clause may contain the following tokens in any order:

token	action
`b1_c(next)`	Update the ACU1 c register set
`b1_r(next)`	Update the ACU1 r register
`b1_w(next)`	Update the ACU1 w register
`b1_d(next)`	Update the ACU0 d register

Table 4.59 - ACU0 Register Update

4.2.4.9.7 ACU2 Update Clause

The ACU2 Update clause may contain the following tokens in any order:

token	action
`b2_c(next)`	Update the ACU2 c register set
`b2_r(next)`	Update the ACU2 r register
`b2_w(next)`	Update the ACU2 w register
`b2_d(next)`	Update the ACU2 d register

Table 4.60 - ACU2 Register Update

4.2.4.9.8 ACU3 Update Clause

The ACU3 Update clause may contain the following tokens in any order:

token	action
`b3_c(next)`	Update the ACU3 c register set
`b3_r(next)`	Update the ACU3 r register
`b3_w(next)`	Update the ACU3 w register
`b3_d(next)`	Update the ACU3 d register

Table 4.61 - ACU3 Register Update

5 Instruction Reference

5.1 Introduction

The DSP instructions fall into one of two main categories: they are either Control Instructions or Compute Instructions. The instruction set is introduced and generally described in this section. The sections which follow are each dedicated to the details of the individual instructions.

Control Instructions are those which are used to configure the global operational characteristics of the machine (called Configuration Instructions), or which control the possible flows of data through the datapath (called Data Flow Control Instructions), or which control the flow of program execution (called Program Flow Instructions).

Compute Instructions are those which direct the computational processes in the various compute modules. The operations may be limited to the ALU (ALU Instructions) or to one or both MAC units (MAC Instructions) or they may direct simultaneous MAC and ALU operations (ALU-MAC Instructions).

5.2 Control Instructions

5.2.1 Configuration Instructions

Configuration Register load instruction

DMA Configuration Register load instruction

ACU configuration instruction

5.2.2 Program Flow Instructions

Branch instruction

Call instruction

If (conditional execution) instruction

Iterate instruction

Nop instruction

Repeat instruction

Return instruction

5.2.3 Data Flow Control Instructions

ACU Data Memory 0/1 with immediate instruction

ACU Data Memory 0/2 with immediate instruction

ACU Data Memory 0/3 with immediate instruction

ACU Data Memory 1/0 with immediate instruction

ACU Data Memory 1/2 with immediate instruction

ACU Data Memory 1/3 with immediate instruction

ACU Data Memory 2/0 with immediate instruction

ACU Data Memory 2/1 with immediate instruction

ACU Data Memory 2/3 with immediate instruction

ACU Data Memory 3/0 with immediate instruction

ACU Data Memory 3/1 with immediate instruction

ACU Data Memory 3/2 with immediate instruction

Scratch Memory/Pointer Cache with immediate instruction

5.3 Compute Instructions

5.3.1 ALU Instructions

ALU-ACU instruction

ALU0 with Immediate instruction

5.3.2 ALU-MAC Instructions

ALU0-MAC0 instruction

ALU0-MAC1 instruction

5.3.3 MAC Instructions

MAC0 with Immediate instruction

MAC1 with Immediate instruction

MAC0-MAC1 instruction

MAC0-MAC1-ACU Instruction

6 ACU configuration instruction

6.1 syntax

```
{b0_ptr_see_clause}{b0_update_clause}
{b1_ptr_see_clause}{b1_update_clause}
{b2_ptr_see_clause}{b2_update_clause}
{b3_ptr_see_clause}{b3_update_clause}
{sm_b0123_see_clause};
```

6.2 description

- The input multiplexers for each of the Data Memory Bus 0 ACU pointer registers are configured to select the indicated sources (the absence of a specified source is still a source and often (but not always) selects the **pimm** register); (see 4.2.4.7.7).
- The input multiplexers of the Data Memory Bus 1 ACU pointer registers are configured; (4.2.4.7.9).
- The input multiplexers of the Data Memory Bus 2 ACU pointer registers are configured; (4.2.4.7.11).
- The input multiplexers of the Data Memory Bus 3 ACU pointer registers are configured; (4.2.4.7.13).
- The cached pointer sets are selected according to the SM B0123 Register-See Clause.
- The Data Memory Bus 0 ACU registers which are specifically indicated in the update clause will be written; (see 4.2.4.9.5).
- The Data Memory Bus 1 ACU registers in the update clause are written; (see 4.2.4.9.6).
- The Data Memory Bus 2 ACU registers in the update clause will be written; (see 4.2.4.9.7).
- The Data Memory Bus 3 ACU registers in the update clause will be written; (see 4.2.4.9.8).

6.3 flags affected

none.

6.4 examples

```
b0_c(&sm), b0_r(&b0_p), b0_d(&b0_p), b0_c(next), b0_r(next),
       b1_c(&sm), b1_r(&sm), b1_d(&b1_p), b1_c(next),
       b1_w(next), b2_c(&pimm), b2_r(&a0_r), b2_c(next)
       b3_c(&pimm), b3_r(&a0_r), b3_c(next),
       sm_b0(&sm_serw[7]), sm_b1(&sm_serw[4]),
       sm_b2(&sm_serw[4]), sm_b3(&sm_serw[0]);
```

7 ACU Data Memory 0/1 with immediate instruction

7.1 syntax

```
{pimm_clause}
{b0_see_clause} {b0_op_clause} {b0_update_clause}
{b1_see_clause} {b1_op_clause} {b1_update_clause};
```

7.2 description

- The **pimm** register is loaded with the specified value; (see 4.2.4.4).
- The input multiplexers for each of the Data Memory Bus 0 ACU registers are configured to select the indicated sources (the absence of a specified source is still a source and often (but not always) selects the **pimm** register).
- The input multiplexers for each of the Data Memory Bus 1 ACU registers are configured as specified.
- The memory read or write specified by the ACU0 Operation Clause is performed, in the DMEM pipeline stage.
- The memory read or write specified by the ACU1 Operation Clause is performed, in the DMEM pipeline stage.
- The registers specified by the ACU0 Update Clause are updated.
- The registers specified by the ACU1 Update Clause are updated.

7.3 flags affected

none.

7.4 examples

```
pimm = 0xBABE, b0_c(&sm), b0_r(&b0_p), b0_d(&b0_p),
       *b0_w = b0_d, b0_p = b0_w + 1, b0_w(next), b0_c(next),
       b1_c(&pimm), b1_r(&a0_r), nop(), b1_c(next);

pimm = 0x1234, b0_c(&sm), b0_r(&sm), b0_d(&b0_p),
       b0_bus = *b0_r, b0_p = b0_w + pimm,
       b0_c(next), b0_w(next),
       b1_c(&pimm), b1_r(&a0_r),
       b1_bus = *b1_r, b1_p = b1_w + 2,
       b1_c(next);
```

8 ACU Data Memory 0/2 with immediate instruction

8.1 syntax

```
{pimm_clause}
{b0_see_clause} {b0_op_clause} {b0_update_clause}
{b2_see_clause} {b2_op_clause} {b2_update_clause};
```

8.2 description

- The **pimm** register is loaded with the specified value; (see 4.2.4.4).
- The input multiplexers for each of the Data Memory Bus 0 ACU registers are configured to select the indicated sources (the absence of a specified source is still a source and often (but not always) selects the **pimm** register).
- The input multiplexers for each of the Data Memory Bus 2 ACU registers are configured as specified.
- The memory read or write specified by the ACU0 Operation Clause is performed, in the DMEM pipeline stage.
- The memory read or write specified by the ACU2 Operation Clause is performed, in the DMEM pipeline stage.
- The registers specified by the ACU0 Update Clause are updated.
- The registers specified by the ACU2 Update Clause are updated.

8.3 flags affected

none.

8.4 examples

```
pimm = 0xBABE, b0_c(&sm), b0_r(&b0_p), b0_d(&b0_p),
       *b0_w = b0_d, b0_p = b0_w + 1, b0_w(next), b0_c(next),
       b2_c(&pimm), b2_r(&a0_r), nop(), b2_c(next);

pimm = 0x1234, b0_c(&sm), b0_r(&sm), b0_d(&b0_p),
       b0_bus = *b0_r, b0_p = b0_w + pimm,
       b0_c(next), b0_w(next),
       b2_c(&pimm), b2_r(&a0_r),
       b2_bus = *b2_r, b2_p = b2_w + 2,
       b2_c(next);
```

9 ACU Data Memory 0/3 with immediate instruction

9.1 syntax

```
{pimm_clause}
{b0_see_clause} {b0_op_clause} {b0_update_clause}
{b3_see_clause} {b3_op_clause} {b3_update_clause};
```

9.2 description

- The **pimm** register is loaded with the specified value; (see 4.2.4.4).
- The input multiplexers for each of the Data Memory Bus 0 ACU registers are configured to select the indicated sources (the absence of a specified source is still a source and often (but not always) selects the **pimm** register).
- The input multiplexers for each of the Data Memory Bus 3 ACU registers are configured as specified.
- The memory read or write specified by the ACU0 Operation Clause is performed, in the DMEM pipeline stage.
- The memory read or write specified by the ACU3 Operation Clause is performed, in the DMEM pipeline stage.
- The registers specified by the ACU0 Update Clause are updated.
- The registers specified by the ACU3 Update Clause are updated.

9.3 flags affected

none.

9.4 examples

```
pimm = 0xBABE, b0_c(&sm), b0_r(&b0_p), b0_d(&b0_p),
       *b0_w = b0_d, b0_p = b0_w + 1, b0_w(next), b0_c(next),
       b3_c(&pimm), b3_r(&a0_r), nop(), b3_c(next);

pimm = 0x1234, b0_c(&sm), b0_r(&sm), b0_d(&b0_p),
       b0_bus = *b0_r, b0_p = b0_w + pimm, b0_c(next),
       b0_w(next), b3_c(&pimm), b3_r(&a0_r), b3_bus = *b3_r,
       b3_p = b3_w + 2, b3_c(next);
```

10 ACU Data Memory 1/0 with immediate instruction

10.1 syntax

{pimm_clause}
{b1_see_clause} {b1_op_clause} {b1_update_clause}
{b0_see_clause} {b0_op_clause} {b0_update_clause}

10.2 description

- The **pimm** register is loaded with the specified value; (see 4.2.4.4).
- The input multiplexers for each of the Data Memory Bus 1 ACU registers are configured to select the indicated sources (the absence of a specified source is still a source and often (but not always) selects the **pimm** register).
- The input multiplexers for each of the Data Memory Bus 0 ACU registers are configured as specified.
- The memory read or write specified by the ACU1 Operation Clause is performed, in the DMEM pipeline stage.
- The memory read or write specified by the ACU0 Operation Clause is performed, in the DMEM pipeline stage.
- The registers specified by the ACU1 Update Clause are updated.
- The registers specified by the ACU0 Update Clause are updated.

10.3 flags affected

none.

10.4 examples

```
pimm = 0xBABE, b1_c(&sm), b1_r(&b1_p), b1_d(&b1_p),
       *b1_w = b1_d, b1_p = b1_w + 1, b1_w(next), b1_c(next),
       b0_c(&pimm), b0_r(&a0_r), nop(), b0_c(next);

pimm = 0x1234, b1_c(&sm), b1_r(&sm), b1_d(&b1_p),
       b1_bus = *b1_r, b1_p = b1_w + pimm,
       b1_c(next), b1_w(next),
       b0_c(&pimm), b0_r(&a0_r),
       b0_bus = *b0_r, b0_p = b0_w + 2,
       b0_c(next);
```

11 ACU Data Memory 1/2 with immediate instruction

11.1 syntax

```
{pimm_clause}
{b1_see_clause} {b1_op_clause} {b1_update_clause}
{b2_see_clause} {b2_op_clause} {b2_update_clause};
```

11.2 description

- The **pimm** register is loaded with the specified value; (see 4.2.4.4).
- The input multiplexers for each of the Data Memory Bus 1 ACU registers are configured to select the indicated sources (the absence of a specified source is still a source and often (but not always) selects the **pimm** register).
- The input multiplexers for each of the Data Memory Bus 2 ACU registers are configured as specified.
- The memory read or write specified by the ACU1 Operation Clause is performed, in the DMEM pipeline stage.
- The memory read or write specified by the ACU2 Operation Clause is performed, in the DMEM pipeline stage.
- The registers specified by the ACU1 Update Clause are updated.
- The registers specified by the ACU2 Update Clause are updated.

11.3 flags affected

none.

11.4 examples

```
pimm = 0xBABE, b1_c(&sm), b1_r(&b1_p), b1_d(&b1_p),
       *b1_w = b1_d, b1_p = b1_w + 1, b1_w(next), b1_c(next),
       b2_c(&pimm), b2_r(&a0_r), nop(), b2_c(next);

pimm = 0x1234, b1_c(&sm), b1_r(&sm), b1_d(&b1_p),
       b1_bus = *b1_r, b1_p = b1_w + pimm, b1_c(next),
       b1_w(next), b2_c(&pimm), b2_r(&a0_r), b2_bus = *b2_r,
       b2_p = b2_w + 2, b2_c(next);
```

12 ACU Data Memory 1/3 with immediate instruction

12.1 syntax

```
{pimm_clause}
{b1_see_clause} {b1_op_clause} {b1_update_clause}
{b3_see_clause} {b3_op_clause} {b3_update_clause};
```

12.2 description

- The **pimm** register is loaded with the specified value; (see 4.2.4.4).
- The input multiplexers for each of the Data Memory Bus 1 ACU registers are configured to select the indicated sources (the absence of a specified source is still a source and often (but not always) selects the **pimm** register).
- The input multiplexers for each of the Data Memory Bus 3 ACU registers are configured as specified.
- The memory read or write specified by the ACU1 Operation Clause is performed, in the DMEM pipeline stage.
- The memory read or write specified by the ACU3 Operation Clause is performed, in the DMEM pipeline stage.
- The registers specified by the ACU1 Update Clause are updated.
- The registers specified by the ACU3 Update Clause are updated.

12.3 flags affected

none.

12.4 examples

```
pimm = 0xBABE, b1_c(&sm), b1_r(&b1_p), b1_d(&b1_p),
       *b1_w = b1_d, b1_p = b1_w + 1, b1_w(next), b1_c(next),
       b3_c(&pimm), b3_r(&a0_r), nop(), b3_c(next);

pimm = 0x1234, b1_c(&sm), b1_r(&sm), b1_d(&b1_p),
       b1_bus = *b1_r, b1_p = b1_w + pimm,
       b1_c(next), b1_w(next),
       b3_c(&pimm), b3_r(&a0_r),
       b3_bus = *b3_r, b3_p = b3_w + 2, b3_c(next);
```

13 ACU Data Memory 2/0 with immediate instruction

13.1 syntax

```
{pimm_clause}
{b2_see_clause} {b2_op_clause} {b2_update_clause};
{b0_see_clause} {b0_op_clause} {b0_update_clause}
```

13.2 description

- The **pimm** register is loaded with the specified value; (see 4.2.4.4).
- The input multiplexers for each of the Data Memory Bus 2 ACU registers are configured to select the indicated sources (the absence of a specified source is still a source and often (but not always) selects the **pimm** register).
- The input multiplexers for each of the Data Memory Bus 0 ACU registers are configured as specified.
- The memory read or write specified by the ACU2 Operation Clause is performed, in the DMEM pipeline stage.
- The memory read or write specified by the ACU0 Operation Clause is performed, in the DMEM pipeline stage.
- The registers specified by the ACU2 Update Clause are updated.
- The registers specified by the ACU0 Update Clause are updated.

13.3 flags affected

none.

13.4 examples

```
pimm = 0xBABE, b2_c(&sm), b2_r(&b2_p), b2_d(&b2_p),
       *b2_w = b2_d, b2_p = b2_w + 1, b2_w(next), b2_c(next),
       b0_c(&pimm), b0_r(&a0_r), nop(), b0_c(next);

pimm = 0x1234, b2_c(&sm), b2_r(&sm), b2_d(&b2_p),
       b2_bus = *b2_r, b2_p = b2_w + pimm,
       b2_c(next), b2_w(next),
       b0_c(&pimm), b0_r(&a0_r),
       b0_bus = *b0_r, b0_p = b0_w + 2,
       b0_c(next);
```

14 ACU Data Memory 2/1 with immediate instruction

14.1 syntax

```
{pimm_clause}
{b2_see_clause} {b2_op_clause} {b2_update_clause};
{b1_see_clause} {b1_op_clause} {b1_update_clause};
```

14.2 description

- The **pimm** register is loaded with the specified value; (see 4.2.4.4).
- The input multiplexers for each of the Data Memory Bus 2 ACU registers are configured to select the indicated sources (the absence of a specified source is still a source and often (but not always) selects the **pimm** register).
- The input multiplexers for each of the Data Memory Bus 1 ACU registers are configured as specified.
- The memory read or write specified by the ACU2 Operation Clause is performed, in the DMEM pipeline stage.
- The memory read or write specified by the ACU1 Operation Clause is performed, in the DMEM pipeline stage.
- The registers specified by the ACU2 Update Clause are updated.
- The registers specified by the ACU1 Update Clause are updated.

14.3 flags affected

none.

14.4 examples

```
pimm = 0xBABE, b2_c(&sm), b2_r(&b2_p), b2_d(&b2_p),
       *b2_w = b2_d, b2_p = b2_w + 1, b2_w(next), b2_c(next),
       b1_c(&pimm), b1_r(&a0_r), nop(), b1_c(next);

pimm = 0x1234, b2_c(&sm), b2_r(&sm), b2_d(&b2_p),
       b2_bus = *b2_r, b2_p = b2_w + pimm,
       b2_c(next), b2_w(next),
       b1_c(&pimm), b1_r(&a0_r),
       b1_bus = *b1_r, b1_p = b1_w + 2,
       b1_c(next);
```

15 ACU Data Memory 2/3 with immediate instruction

15.1 syntax

```
{pimm_clause}
{b2_see_clause} {b2_op_clause} {b2_update_clause};
{b3_see_clause} {b3_op_clause} {b3_update_clause};
```

15.2 description

- The **pimm** register is loaded with the specified value; (see 4.2.4.4).
- The input multiplexers for each of the Data Memory Bus 2 ACU registers are configured to select the indicated sources (the absence of a specified source is still a source and often (but not always) selects the **pimm** register).
- The input multiplexers for each of the Data Memory Bus 3 ACU registers are configured as specified.
- The memory read or write specified by the ACU2 Operation Clause is performed, in the DMEM pipeline stage.
- The memory read or write specified by the ACU3 Operation Clause is performed, in the DMEM pipeline stage.
- The registers specified by the ACU2 Update Clause are updated.
- The registers specified by the ACU3 Update Clause are updated.

15.3 flags affected

none.

15.4 examples

```
pimm = 0xBABE, b2_c(&sm), b2_r(&b2_p), b2_d(&b2_p),
       *b2_w = b2_d, b2_p = b2_w + 1, b2_w(next), b2_c(next),
       b3_c(&pimm), b3_r(&a0_r), nop(), b3_c(next);

pimm = 0x1234, b2_c(&sm), b2_r(&sm), b2_d(&b2_p),
       b2_bus = *b2_r, b2_p = b2_w + pimm,
       b2_c(next), b2_w(next),
       b3_c(&pimm), b3_r(&a0_r),
       b3_bus = *b3_r, b3_p = b3_w + 2,
       b3_c(next);
```

16 ACU Data Memory 3/0 with immediate instruction

16.1 syntax

```
{pimm_clause}
{b3_see_clause} {b3_op_clause} {b3_update_clause};
{b0_see_clause} {b0_op_clause} {b0_update_clause}
```

16.2 description

- The **pimm** register is loaded with the specified value; (see 4.2.4.4).
- The input multiplexers for each of the Data Memory Bus 3 ACU registers are configured to select the indicated sources (the absence of a specified source is still a source and often (but not always) selects the **pimm** register).
- The input multiplexers for each of the Data Memory Bus 0 ACU registers are configured as specified.
- The memory read or write specified by the ACU3 Operation Clause is performed, in the DMEM pipeline stage.
- The memory read or write specified by the ACU0 Operation Clause is performed, in the DMEM pipeline stage.
- The registers specified by the ACU3 Update Clause are updated.
- The registers specified by the ACU0 Update Clause are updated.

16.3 flags affected

none.

16.4 examples

```
pimm = 0xBABE, b3_c(&sm), b3_r(&32_p), b3_d(&b3_p),
       *b3_w = b3_d, b3_p = b3_w + 1, b3_w(next), b3_c(next),
       b0_c(&pimm), b0_r(&a0_r), nop(), b0_c(next);

pimm = 0x1234, b3_c(&sm), b3_r(&sm), b3_d(&b3_p),
       b3_bus = *b3_r, b3_p = b3_w + pimm,
       b3_c(next), b3_w(next),
       b0_c(&pimm), b0_r(&a0_r),
       b0_bus = *b0_r, b0_p = b0_w + 2,
       b0_c(next);
```

17 ACU Data Memory 3/1 with immediate instruction

17.1 syntax

```
{pimm_clause}
{b3_see_clause} {b3_op_clause} {b3_update_clause};
{b1_see_clause} {b1_op_clause} {b1_update_clause};
```

17.2 description

- The **pimm** register is loaded with the specified value; (see 4.2.4.4).
- The input multiplexers for each of the Data Memory Bus 3 ACU registers are configured to select the indicated sources (the absence of a specified source is still a source and often (but not always) selects the **pimm** register).
- The input multiplexers for each of the Data Memory Bus 1 ACU registers are configured as specified.
- The memory read or write specified by the ACU3 Operation Clause is performed, in the DMEM pipeline stage.
- The memory read or write specified by the ACU1 Operation Clause is performed, in the DMEM pipeline stage.
- The registers specified by the ACU3 Update Clause are updated.
- The registers specified by the ACU1 Update Clause are updated.

17.3 flags affected

none.

17.4 examples

```
pimm = 0xBABE, b3_c(&sm), b3_r(&b3_p), b3_d(&b3_p),
       *b3_w = b3_d, b3_p = b3_w + 1, b3_w(next), b3_c(next),
       b1_c(&pimm), b1_r(&a0_r), nop(), b1_c(next);

pimm = 0x1234, b3_c(&sm), b3_r(&sm), b3_d(&b3_p),
       b3_bus = *b3_r, b3_p = b3_w + pimm,
       b3_c(next), b3_w(next),
       b1_c(&pimm), b1_r(&a0_r),
       b1_bus = *b1_r, b1_p = b1_w + 2,
       b1_c(next);
```

18 ACU Data Memory 3/2 with immediate instruction

18.1 syntax

```
{pimm_clause}
{b3_see_clause} {b3_op_clause} {b3_update_clause};
{b2_see_clause} {b2_op_clause} {b2_update_clause};
```

18.2 description

- The **pimm** register is loaded with the specified value; (see 4.2.4.4).
- The input multiplexers for each of the Data Memory Bus 3 ACU registers are configured to select the indicated sources (the absence of a specified source is still a source and often (but not always) selects the **pimm** register).
- The input multiplexers for each of the Data Memory Bus 2 ACU registers are configured as specified.
- The memory read or write specified by the ACU3 Operation Clause is performed, in the DMEM pipeline stage.
- The memory read or write specified by the ACU2 Operation Clause is performed, in the DMEM pipeline stage.
- The registers specified by the ACU3 Update Clause are updated.
- The registers specified by the ACU2 Update Clause are updated.

18.3 flags affected

none.

18.4 examples

```
pimm = 0xBABE, b3_c(&sm), b3_r(&b3_p), b3_d(&b3_p),
       *b3_w = b3_d, b3_p = b3_w + 1, b3_w(next), b3_c(next),
       b2_c(&pimm), b2_r(&a0_r), nop(), b2_c(next);

pimm = 0x1234, b3_c(&sm), b3_r(&sm), b3_d(&b3_p),
       b3_bus = *b3_r, b3_p = b3_w + pimm,
       b3_c(next), b3_w(next),
       b2_c(&pimm), b2_r(&a0_r),
       b2_bus = *b2_r, b2_p = b2_w + 2,
       b2_c(next);
```

19 ALU-ACU instruction

19.1 syntax

```
{alu0_op_clause}{alu0_update_clause}

{bus0_op_clause}{bus0_update_clause}

{bus1_op_clause}{bus1_update_clause}

(bus2_op_clause){bus2_update_clause}

{bus3_op_clause}{bus3_update_clause} ;
```

19.2 description

- The specified ALU0 operation is performed; (see 4.2.4.8.1).
- The specified ACU0 operation is performed; (see 4.2.4.8.5).
- The specified ACU1 operation is performed; (see 4.2.4.8.6).
- The specified ACU2 operation is performed; (see 4.2.4.8.7).
- The specified ACU3 operation is performed; (see 4.2.4.8.8).
- The ALU0 registers which are specifically indicated in the update clause will be written; (see 4.2.4.9.1).
- The ACU0 registers which are specifically indicated in the update clause will be written; (see 4.2.4.9.5).
- The ACU1 registers which are specifically indicated in the update clause will be written; (see 4.2.4.9.6).
- The ACU2 registers which are specifically indicated in the update clause will be written; (see 4.2.4.9.7).
- The ACU3 registers which are specifically indicated in the update clause will be written; (see 4.2.4.9.8).

19.3 flags affected

ALU0 flags, according to the operation and if **a0_f(next)** is specified in the ALU0 update clause.

19.4 examples

```
(a0_x + a0_y), a0_s(next),
      *b0_w = b0_d, b0_p = b0_w + 1, b0_w(next), b0_c(next),
      b1_c(&pimm), b1_r(&a0_r), nop(),
      b2_c(&pimm), b2_r(&a0_r), nop(),
      b3_c(&pimm), b3_r(&a0_r), nop();

 (logic)(a0_x << a0_y[3:0]),
      a0_x(next), a0_y(next), a0_s(next), a0_f(next),
      b0_c(&sm), b0_r(&b0_p), b0_d(&b0_p),
      *b0_w = b0_d, b0_p = b0_w + 1,
      b0_w(next), b0_c(next),
      b1_c(&sm), b1_r(&sm), b1_d(&b1_p),
      b1_bus = *b1_r, b1_p = b1_w + pimm,
      b1_c(next), b1_w(next),
      b2_c(&pimm), b2_r(&a0_r),
      b2_bus = *b2_r, b2_p = b2_w + 2,
      b2_c(next),
      b3_c(&pimm), b3_r(&a0_r),
      nop(),
      b3_c(next);
// a comment for that instruction seems in order here.
```

20 ALU0 with Immediate instruction

20.1 syntax

{pimm_clause}

{alu0_see_clause}{alu0_op_clause}{alu0_update_clause} **;**

20.2 description

- The **pimm** register is loaded with the specified value; (see 4.2.4.4).
- The input multiplexers for each of the ALU0 registers are configured to select the indicated sources (the absence of a specified source is still a source and often (but not always) selects the **pimm** register); (see 4.2.4.7.1).
- The specified ALU0 operation is performed; (see 4.2.4.8.1).
- The ALU0 registers which are specifically indicated in the update clause will be written; (see 4.2.4.9.1).

20.3 flags affected

ALU0 flags, according to the operation and if **a0_f(next)** is specified in the update clause.

20.4 examples

```
pimm = 0x1234, a0_y(&pimm), a0_xl & a0_yl, a0_f(next);

pimm = 0xFADE, a0_x(&m0), a0_y(&m1), a0_s(&a0), a0_xl + a0_yl,
       a0_x(next), a0_y(next), a0_s(next), a0_f(next);
```

21 ALU0-MAC0 instruction

21.1 syntax

{alu0_see_clause}{alu0_op_clause}{alu0_update_clause}

{mac0_see_clause}{mac0_op_clause}{mac0_update_clause} **;**

21.2 description

- The input multiplexers for each of the ALU0 registers are configured to select the indicated sources (the absence of a specified source is still a source and often (but not always) selects the **pimm** register); (see 4.2.4.7.1).
- The input multiplexers for each of the MAC0 registers are configured to select the indicated sources (the absence of a specified source is still a source and often (but not always) selects the **pimm** register); (see 4.2.4.7.2).
- The specified ALU0 operation is performed; (see 4.2.4.8.1).
- The specified MAC0 operation is performed; (see 4.2.4.8.2).
- The ALU0 registers which are specifically indicated in the update clause will be written; (see 4.2.4.9.1).
- The MAC0 registers which are specifically indicated in the update clause will be written; (see 4.2.4.9.2).

21.3 flags affected

ALU0 flags, according to the operation and if **a0_f(next)** is specified in the ALU0 update clause.

MAC0 flags, according to the operation and if **m0_f(next)** is specified in the MAC0 update clause.

21.4 examples

```
a0_y(&pimm), a0_xl & a0_yl, a0_f(next),
      m0_x(&a0), (m0_x * m0_y), m0_f(next);

a0_x(&m0), a0_y(&m1), a0_s(&a0), a0_xl + a0_yl , a0_x(next),
      a0_y(next), a0_s(next), a0_f(next),
      m0_x(&a0), m0_y(&m1), m0_s(&m0), m0_f(&m0),
      m0_s[39:0] + (m0_x * m0_y),
      m0_x(next), m0_y(next), m0_s(next), m0_f(next);
```

22 ALU0-MAC1 instruction

22.1 syntax

{alu0_see_clause}{alu0_op_clause}{alu0_update_clause}

{mac1_see_clause}{mac1_op_clause}{mac1_update_clause} **;**

22.2 description

- The input multiplexers for each of the ALU0 registers are configured to select the indicated sources (the absence of a specified source is still a source and often (but not always) selects the **pimm** register); (see 4.2.4.7.1).
- The input multiplexers for each of the MAC1 registers are configured to select the indicated sources (the absence of a specified source is still a source and often (but not always) selects the **pimm** register); (see 4.2.4.7.3).
- The specified ALU0 operation is performed; (see 4.2.4.8.1).
- The specified MAC1 operation is performed; (see 4.2.4.8.3).
- The ALU0 registers which are specifically indicated in the update clause will be written; (see 4.2.4.9.1).
- The MAC1 registers which are specifically indicated in the update clause will be written; (see 4.2.4.9.3).

22.3 flags affected

ALU0 flags, according to the operation and if **a0_f(next)** is specified in the ALU0 update clause.

MAC1 flags, according to the operation and if **m0_f(next)** is specified in the MAC0 update clause.

22.4 examples

```
a0_y(&pimm), a0_xl & a0_yl, a0_f(next),
      m1_x(&a0), (m1_x * m1_y), m1_f(next);

a0_x(&m0), a0_y(&m1), a0_s(&a0), a0_xl + a0_yl,
      a0_x(next), a0_y(next), a0_s(next), a0_f(next),
      m1_x(&a0), m1_y(&m0), m1_s(&m1), m1_f(&m1),
      m1_s[39:0] + (m1_x * m1_y),
      m1_x(next), m1_y(next), m1_s(next), m1_f(next);
```

23 Branch instruction

23.1 syntax

```
pc = {hexadecimal_target_address} , flush();

pc = {hexadecimal_target_address} ;
```

23.2 description

The Program Counter (PC) register is set to the specified address and the machine begins fetching and executing the sequence of instructions starting at that point.

This DSP is a pipelined machine with a branch penalty of 3 cycles. The **branch** instruction has two forms which allow the selection of the desired pipeline operation during the branch.

In the first form of the instruction, the instruction pipeline is flushed once the **branch** instruction is executed. The flush operation will eliminate the instructions already in the prefetch, fetch, and decode stages of the pipeline. The resultant 3 stage bubble in the pipeline is the branch penalty. This case is roughly equivalent to following the branch with 3 **nop()** instructions and not flushing the pipeline.

In the second form of the instruction, the pipeline is not flushed. The 3 instructions immediately after the **branch** instruction will already be in the pipeline by the time the branch instruction is executed. These instructions will be in the decode, fetch, and prefetch stages, respectively, of the pipeline. The 3 instructions are run to completion while the DSP begins fetching instruction from the new address into the pipeline. In this form of the instruction there is no bubble introduced into the pipeline and no performance penalty exists.

23.3 flags affected

none.

23.4 examples

```
pc = 0x1234, flush();

pc = 0x1234; nop(); nop(); nop(); // same as with flush

pc = 0x1234;
   pimm = 0x0001, a0_y(&pimm), a0_y(next); // load ptr inc.
   a0_s(&a0), a0_r(&a0_s), (a0_x + a0_y), a0_s(next); // add
   b0_d(&a0_r), *b0_w = b0_d, b0_p = b0_w+1, b3_d(&pimm), nop();
// pointer now written back.
// instruction at 0x1234 now follows immediately in pipeline.
```

24 Call instruction

24.1 syntax

```
call ( {hexadecimal_target_address} ) , flush();

call ( {hexadecimal_target_address} );
```

24.2 description

The program counter (PC) register value is incremented, and pushed onto the stack and program execution jumps to the specified address in and the machine begins fetching and executing the sequence of instructions starting at that point.

This DSP is a pipelined machine with a branch penalty of 3 cycles. The **call** instruction has two forms which allow the selection of the desired pipeline operation during the branch.

In the first form of the instruction, the instruction pipeline is flushed once the **call** instruction is executed. The flush operation will eliminate the instructions already in the prefetch, fetch, and decode stages of the pipeline. The resultant 3 stage bubble in the pipeline is the branch penalty. This case is roughly equivalent to following the **call** with 3 **nop()** instructions and not flushing the pipeline.

In the second form of the instruction, the pipeline is not flushed. The 3 instructions immediately after the **call** instruction will already be in the pipeline by the time the branch instruction is executed. These instructions will be in the decode, fetch, and prefetch stages, respectively, of the pipeline. The 3 instructions are run to completion while the DSP begins fetching instruction from the new address into the pipeline. In this form of the instruction there is no bubble introduced into the pipeline and no performance penalty exists.

24.3 flags affected

none.

24.4 examples

```
call(0x1234), flush();

call(0x1234); nop(); nop(); nop(); // same as with flush

call(0x1234);
   pimm = 0x0001, a0_y(&pimm), a0_y(next); // load ptr inc.
   a0_s(&a0), a0_r(&a0_s), (a0_x + a0_y), a0_s(next); // add
   b0_d(&a0_r), *b0_w = b0_d, b0_p = b0_w+1, b3_d(&pimm), nop();
// pointer now written back.
// instruction at 0x1234 now follows immediately in pipeline.
```

25 Configuration Register load instruction

25.1 syntax

config = *{configuration_clause}* **;**

25.2 description

the 'config' register is loaded with the specified value.

The tokens in the configuration clause are joined with any mix of plus '**+**' or bar '**|**' characters (which perform a logical OR operation) or the ampersand '**&**' character (which performs a logical AND operation.)

The precedence of the OR and AND operations are strictly left to right.

See also 3.3.7 Configuration Register and 4.2.4.5 Configuration clause.

25.3 flags affected

none.

25.4 examples

```
config = 0xDEAD; // magic numbers are bad.
Config = m1(fract) | m1(round) & 0x42; // magicnums still bad.
config = a0(sat) + m0(fract), m1(int) | m1(round) + b0(lin) +
     b3(lin) + b2(circ) + b3(circ);
```

26 DMA Configuration Register load instruction

26.1 syntax

dmaconfig = *{dmaconfiguration_clause}* **;**

26.2 description

The dmaconfig register is loaded with the specified values.

The tokens in the configuration clause are joined with any mix of plus '**+**' or bar '**|**' characters (which perform a logical OR operation) or the ampersand '**&**' character (which performs a logical AND operation.)

The precedence of the OR and AND operations are strictly left to right.

See also 3.3.6 DMA Configuration Register and 4.2.4.6 DMA Configuration clause.

26.3 flags affected

none.

26.4 examples

```
dmaconfig = 0xDEAD; // magic numbers are bad.
dmaconfig = see_dma(on) + dab(1) + adb(1) + daen(on) +
      aden(on);
```

27 If (conditional execution) instruction

27.1 syntax

if(*{conditional_test}* **)** *{permissible_instruction}***;**

27.2 description

The test of a machine flag against a specified condition is performed. If the test evaluates to FALSE, the program counter (pc) is simply incremented. If the test evaluates to TRUE, the specified "permissible" instruction is executed.

A *permissible_instruction* is one of: **branch, call, return**.

The *conditional_test* has three forms. The form is one of:

{testable_flag}{equality_test}{logic_value}

or

{testable_acu_pointer}{equality_test}{acu_end_pointer}

or

*{iteration_register}***--** *{equality_test}* **0**

The *testable_flag* is any one of the flags of ALU0 (see 3.4.4), MAC0 (see 3.5.4), MAC1 (see 3.6.4) or 'XF' the externally testable hardware flag.

The *equality_test* is either of "**==**" or "**!=**" as desired by the programmer.

The *logic_value* is any of "0" or "false" or "reset" to syntactically select logical zero. To select logical one, any of "1" or "true" or "set" may be used.

The second form of the *conditional_test* is:

{testable_acu_pointer}{equality_test}{acu_end_pointer}

The *testable_acu_pointer* is any of the read or write registers in one of the four ACUs (see 3.8.2 or 3.8.3 for ACU0 registers, other sections as appropriate for the other ACUs).

The *equality_test* is either of "**==**" or "**!=**" as desired by the programmer.

The *acu_end_pointer* is the end (circular buffer) pointer register (noted as one of **b0_e**, **b1_e**, **b2_e**, **b3_e**, as appropriate) in the same ACU as the specified *testable_acu_pointer*.

The third form form of the instruction is used to test and decrement one of the two interation counter registers: **iterate0** or **iterate1**. The test is whether or not the register is equal to zero ("**== 0**") or not ("**!= 0**"), and the register is always decremented by the execution of this instruction. Note the "**--**" suffixed to the register name.

DSP pipeline is designed so that conditionals are treated as “branch not taken” cases: in the event that the conditional evaluates to TRUE the branch penalty is 3 cycles.

27.3 flags affected

none.

27.4 examples

```
if (a0z == set) call(0x1234);
if (m0eov == 0) return();
if (a0gt != TRUE) pc = 0x1234;

if (b0_r != b0_e) pc = 0x1234;
if (b3_w == b3_e) return();

if (iterate0-- != 0) pc = myloop;
if (iterate1-- == 0) call(sub_sample);
```

28 Iterate instruction

28.1 syntax

{iteration counter register} **=** *{count}* **;**

28.2 description

the iteration counter register (**iterate0** or **iterate1**) is loaded with the specified count. The count may be any value from 0 to 65535 (0xffff).

28.3 flags affected

none.

28.4 examples

```
iterate0 = 0x1234;
iterate1 = 42;
```

29 MAC0 with Immediate instruction

29.1 syntax

{pimm_clause}

{mac0_see_clause}{mac0_op_clause}{mac0_update_clause} **;**

29.2 description

- The **pimm** register is loaded with the specified value; (see 4.2.4.4).
- The input multiplexers for each of the MAC0 registers are configured to select the indicated sources (the absence of a specified source is still a source and often (but not always) selects the **pimm** register); (see 4.2.4.7.2).
- The specified MAC0 operation is performed; (see 4.2.4.8.2).
- The registers which are specifically indicated in the update clause will be written; (see 4.2.4.9.2).

29.3 flags affected

MAC0 flags, according to the operation and if **m0_f(next)** is specified in the update clause.

29.4 examples

```
pimm = 0x1234, m0_y(&pimm), (m0_x * m0_y), m0_f(next);

pimm = 0xFADE, m0_x(&a0), m0_y(&m1), m0_s(&m0), m0_f(&m0),
       m0_s[39:0] + (m0_x * m0_y) , m0_x(next), m0_y(next),
       m0_s(next), m0_f(next);
```

30 MAC0-MAC1 instruction

30.1 syntax

{mac0_see_clause}{mac0_op_clause}{mac0_update_clause}

{mac1_see_clause}{mac1_op_clause}{mac1_update_clause} **;**

30.2 description

- The input multiplexers for each of the MAC0 registers are configured to select the indicated sources (the absence of a specified source is still a source and often (but not always) selects the **pimm** register); (see 4.2.4.7.2).
- The input multiplexers for each of the MAC1 registers are configured to select the indicated sources (the absence of a specified source is still a source and often (but not always) selects the **pimm** register); (see 4.2.4.7.3).
- The specified MAC0 operation is performed; (see 4.2.4.8.2).
- The specified MAC1 operation is performed; (see 4.2.4.8.3).
- The MAC0 registers which are specifically indicated in the update clause will be written; (see 4.2.4.9.2).
- The MAC1 registers which are specifically indicated in the update clause will be written; (see 4.2.4.9.3).

30.3 flags affected

MAC0 flags, according to the operation and if **m0_f(next)** is specified in the MAC0 update clause.

MAC1 flags, according to the operation and if **m1_f(next)** is specified in the MAC1 update clause.

30.4 examples

```
m0_x(&a0), (m0_x*m0_y), m0_f(next), m1_f(&m1), (m1_x*m1_y);

m0_x(&a0), m0_y(&m1), m0_s(&m0), m0_f(&m0), m0_s[39:0] + (m0_x
      * m0_y) , m0_x(next), m0_y(next), m0_s(next),
      m0_f(next), m1_x(&a0), m1_y(&m0), m1_s(&m1), m1_f(&m1),
      m1_s[39:0] - (m1_x * m1_y) , m1_x(next), m1_y(next),
      m1_s(next), m1_f(next);
```

31 MAC0-MAC1-ACU Instruction

31.1 syntax

```
{mac0_op_clause}{mac0_update_clause}

{mac1_op_clause}{mac1_update_clause}

{bus0_op_clause}{bus0_update_clause}

{bus1_op_clause}{bus1_update_clause}

(bus2_op_clause){bus2_update_clause}

{bus3_op_clause}{bus3_update_clause} ;
```

31.2 description

- The specified MAC0 operation is performed; (see 4.2.4.8.2).
- The specified MAC1 operation is performed; (see 4.2.4.8.3).
- The specified ACU0 operation is performed; (see 4.2.4.8.5).
- The specified ACU1 operation is performed; (see 4.2.4.8.6).
- The specified ACU2 operation is performed; (see 4.2.4.8.7).
- The specified ACU3 operation is performed; (see 4.2.4.8.8).
- The MAC0 registers which are specifically indicated in the update clause will be written; (see 4.2.4.9.2).
- The MAC1 registers which are specifically indicated in the update clause will be written; (see 4.2.4.9.3).
- The ACU0 registers which are specifically indicated in the update clause will be written; (see 4.2.4.9.5).
- The ACU1 registers which are specifically indicated in the update clause will be written; (see 4.2.4.9.6).
- The ACU2 registers which are specifically indicated in the update clause will be written; (see 4.2.4.9.7).
- The ACU3 registers which are specifically indicated in the update clause will be written; (see 4.2.4.9.8).

31.3 flags affected

MAC0 flags, according to the operation and if **m0_f(next)** is specified in the MAC0 update clause.

MAC1 flags, according to the operation and if **m1_f(next)** is specified in the MAC1 update clause.

31.4 examples

```
(m0_x*m0_y), m0_f(next), (m1_x*m1_y),
      b0_c(&sm), b0_r(&b0_p), b0_d(&b0_p),
      *b0_w = b0_d, b0_p = b0_w + 1, b0_w(next), b0_c(next),
      b1_c(&pimm), b1_r(&a0_r), nop(),
      b2_c(&pimm), b2_r(&a0_r), nop(),
      b3_c(&pimm), b3_r(&a0_r), nop();

m0_s[39:0] + (m0_x * m0_y),
      m0_x(next), m0_y(next), m0_s(next), m0_f(next),
      m1_s[39:0] - (m1_x * m1_y),
      m1_x(next), m1_y(next), m1_s(next), m1_f(next),
      b0_c(&sm), b0_r(&b0_p), b0_d(&b0_p),
      *b0_w = b0_d, b0_p = b0_w + 1,
      b0_w(next), b0_c(next),
      b1_c(&sm), b1_r(&sm), b1_d(&b1_p),
      b1_bus = *b1_r, b1_p = b1_w + pimm,
      b1_c(next), b1_w(next),
      b2_c(&pimm), b2_r(&a0_r),
      b2_bus = *b2_r, b2_p = b2_w + 2,
      b2_c(next),
      b3_c(&pimm), b3_r(&a0_r),
      nop(),
      b3_c(next);
// a comment for that instruction seems in order here.
```

32 MAC1 with Immediate instruction

32.1 syntax

{pimm_clause}

{mac1_see_clause}{mac1_op_clause}{mac1_update_clause} **;**

32.2 description

- The **pimm** register is loaded with the specified value; (see 4.2.4.4).
- The input multiplexers for each of the MAC1 registers are configured to select the indicated sources (the absence of a specified source is still a source and often (but not always) selects the **pimm** register); (see 4.2.4.7.3).
- The specified MAC1 operation is performed; (see 4.2.4.8.3).
- The registers which are specifically indicated in the update clause will be written; (see 4.2.4.9.3).

32.3 flags affected

MAC1 flags, according to the operation and if **m1_f(next)** is specified in the update clause.

32.4 examples

```
pimm = 0x1234, m1_y(&pimm), (m1_x * m1_y), m1_f(next);

pimm = 0xBABE, m1_x(&a0), m1_y(&m0), m1_s(&m1), m1_f(&m1),
       m1_s[39:0] + (m1_x * m1_y) , m1_x(next), m1_y(next),
       m1_s(next), m1_f(next);
```

33 Nop instruction

33.1 syntax

```
nop();
pc++;
pc += 1;
```

33.2 description

no operation is performed. The program counter (pc) is incremented.

33.3 flags affected

none.

33.4 examples

```
nop();
pc++;
pc += 1;
```

34 Repeat instruction

34.1 syntax

```
repeat = {hexadecimal_repeat_count} ;
```

34.2 description

the repeat counter register is loaded and the instruction immediately following in the program memory is fetched and executed the specified number of times.

NB: The hardware does one more than the number which is actually loaded into the register, however the assembler makes an adjustment in the machine level instruction coding so that the register is actually loaded with one less than the source code specified value, resulting in performance as indicated by the programmer.

NB: The minimum repeat count at the source code level is 2, the maximum is 65536. These map to the machine code level constants 1 and 65535 (0xffff) respectively.

34.3 flags affected

the **repeat** instruction affects none, but the repeated instruction might do so.

34.4 examples

```
repeat = 0x1234;
   nop();
```

35 Return instruction

35.1 syntax

```
return(), flush();

return();
```

35.2 description

The program counter (PC) register value is loaded from the stack and program execution jumps to the new address in the program memory and the machine begins fetching and executing the sequence of instructions starting at that point.

This DSP is a pipelined machine with a branch penalty of 3 cycles. The **`return`** instruction has two forms which allow the selection of the desired pipeline operation during the branch.

In the first form of the instruction, the instruction pipeline is flushed once the **`return`** instruction is executed. The flush operation will eliminate the instructions already in the prefetch, fetch, and decode stages of the pipeline. The resultant 3 stage bubble in the pipeline is the branch penalty. This case is roughly equivalent to following the **`return`** with 3 **nop()** instructions and not flushing the pipeline.

In the second form of the instruction, the pipeline is not flushed. The 3 instructions immediately after the **`return`** instruction will already be in the pipeline by the time the branch instruction is executed. These instructions will be in the decode, fetch, and prefetch stages, respectively, of the pipeline. The 3 instructions are run to completion while the DSP begins fetching instruction from the new address into the pipeline. In this form of the instruction there is no bubble introduced into the pipeline and no performance penalty exists.

35.3 flags affected

none.

35.4 examples

```
return(), flush();

return(); nop(); nop(); nop(); // same as return(), flush();

return();
   pimm = 0x0001, a0_y(&pimm), a0_y(next); // load ptr inc.
   a0_s(&a0), a0_r(&a0_s), (a0_x + a0_y), a0_s(next); // add
   b0_d(&a0_r), *b0_w = b0_d, b0_p = b0_w+1, b3_d(&pimm), nop();
// pointer now written back.
// instruction after original call now follows immediately
//    in the pipeline.
```

36 Scratch Memory/Pointer Cache with immediate instruction

36.1 syntax

{pimm_clause}

{sm_see_clause}{sm_op_clause}{sm_update_clause} **;**

36.2 description

the **pimm** register is loaded according to the Pimm Clause and the "serw" cache pointer set for the data bus ACU's are selected by the SM Register-See Clause. In the SM Operation Clause the cache pointer write index register (**sm_i**) is set and the **sm** memory operation (if any) is performed and the new value for the address register (**sm_a**) is calculated. In the SM Update Clause the selected registers are updated.

36.3 flags affected

none.

36.4 examples

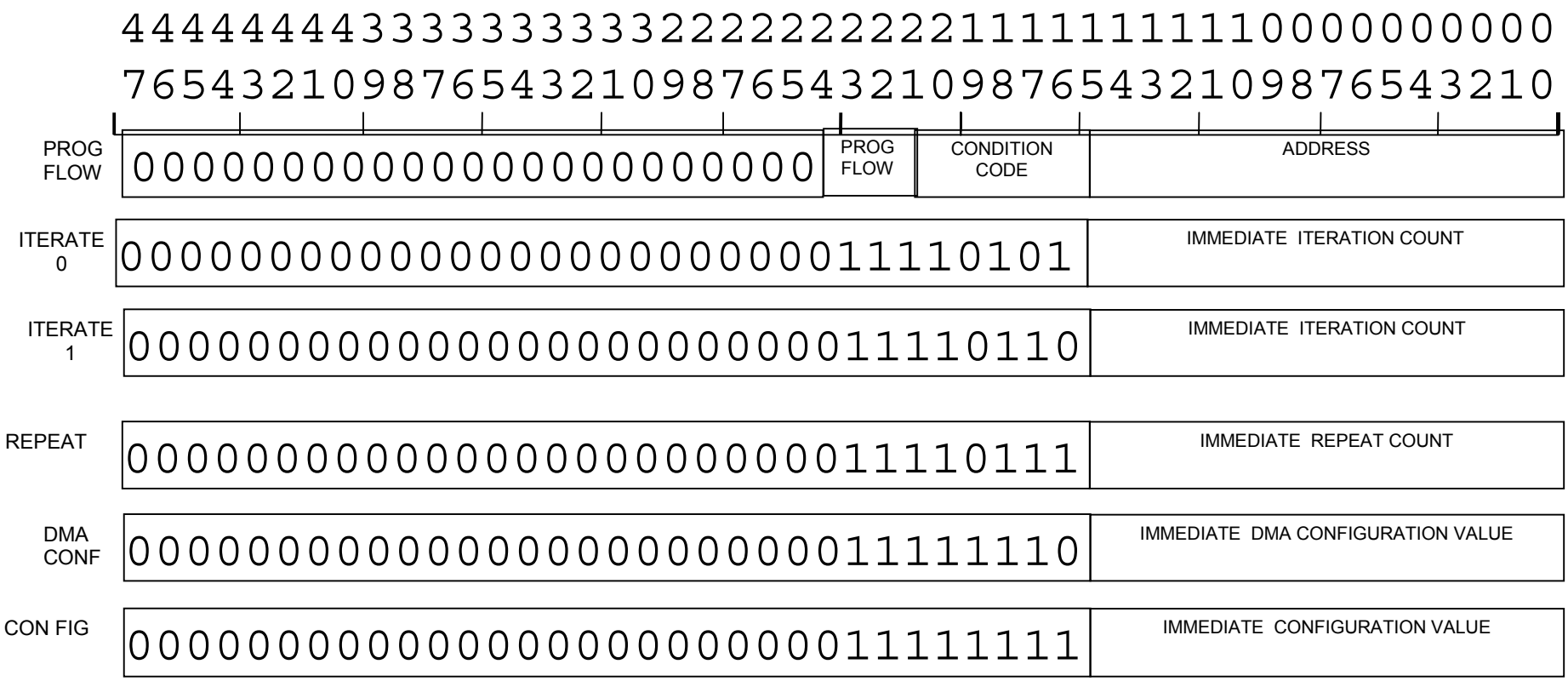
```
pimm = 0x1234, sm_s(&pimm), sm_b0(&sm_serw[1]),
       sm_b1(&sm_serw[6]), sm_i = 4, sm_a = pimm, sm_s(next);

pimm = 0xDEAD, sm_s(&sm), sm_r(&sm),
       sm_b0(&sm_serw[7]), sm_b1(&sm_serw[4]),
       sm_b2(&sm_serw[4]), sm_b3(&sm_serw[0]),
       sm_i = 0, sm = *sm_a++,
       sm_s(next), sm_e(next), sm_r(next), sm_w(next);
       /* fetch new pointers to cache set 0 (sm_i=0) */
// note that address register is _not_ updated!

// write back pointer set 5 (sm_b2 see clause) to
//    location "my_ptr" in sm (sm_i is ignored).
// assume that sm_a already contains "my_ptr" address.
pimm = useful_value, sm_b2(&sm_serw[5]), sm_i=0,
       *sm_a = sm;
```

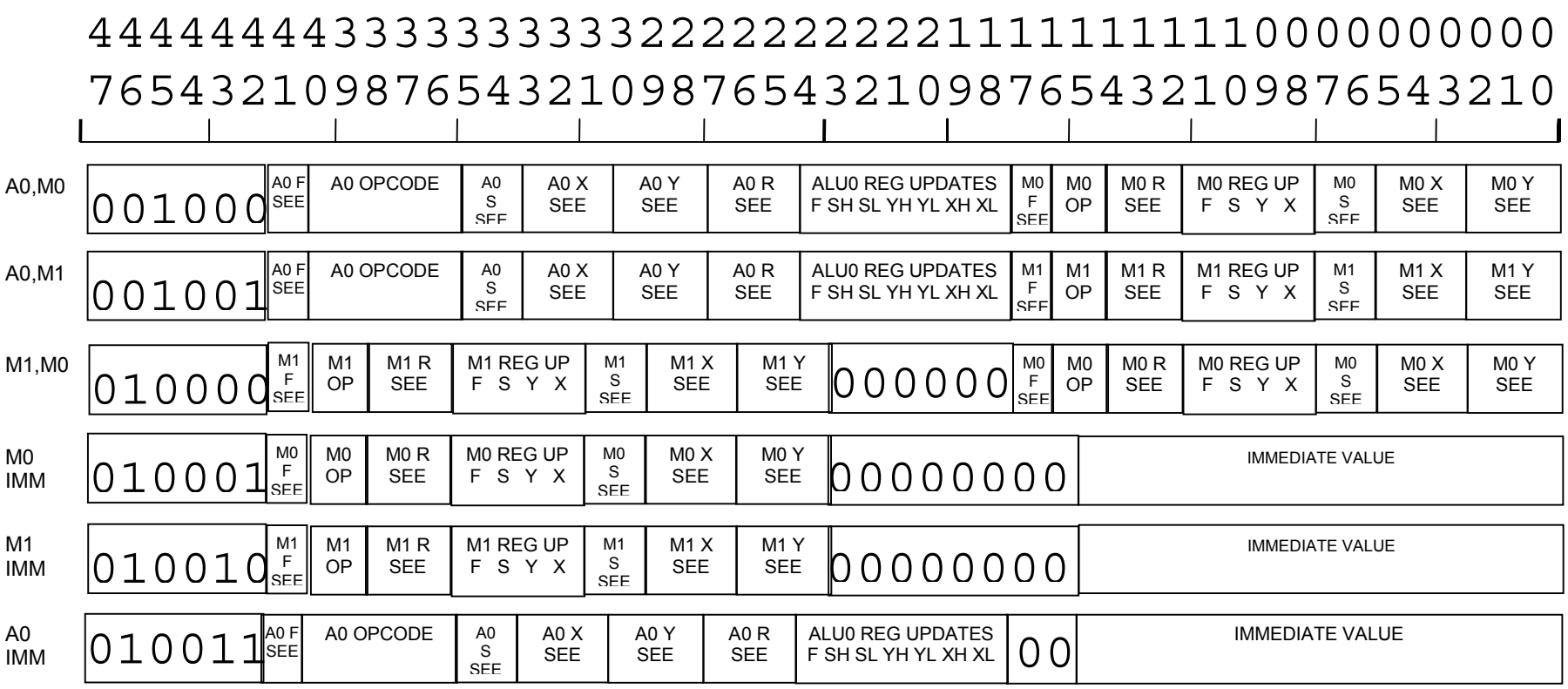
37 Appendix – Instructions in Numerical Order

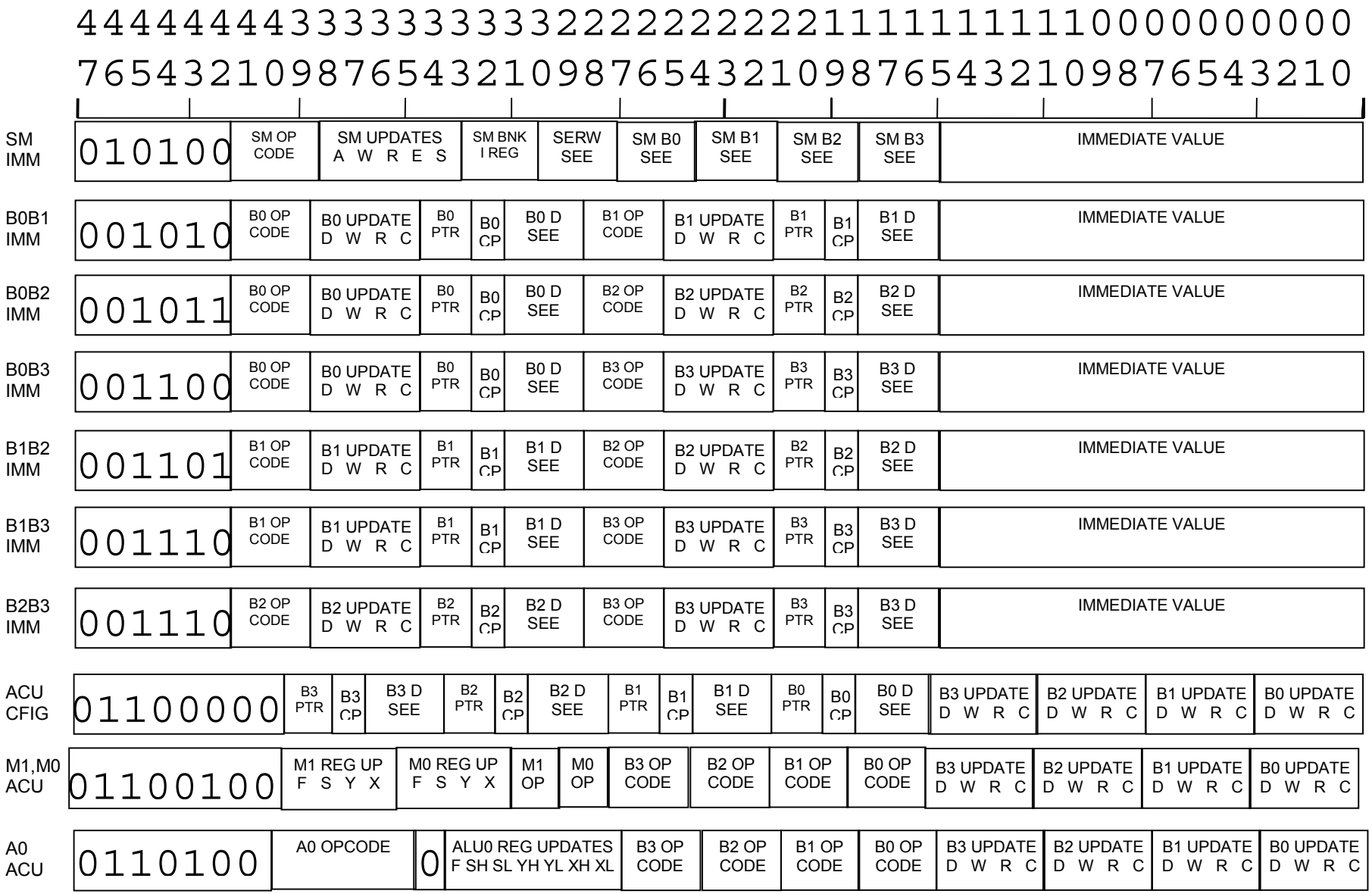
Instruction word [47:x]	Instruction type
0000 0000 0000 0000 0000 0000 00	Branch instruction
0000 0000 0000 0000 0000 0000 01	Call instruction
0000 0000 0000 0000 0000 0000 10	Return instruction
0000 0000 0000 0000 0000 0000 1111 0101	Iterate instruction (register 0)
0000 0000 0000 0000 0000 0000 1111 0110	Iterate instruction (register 1)
0000 0000 0000 0000 0000 0000 1111 0111	Repeat instruction
0000 0000 0000 0000 0000 0000 1111 1110	DMA Configuration Register load instruction
0000 0000 0000 0000 0000 0000 1111 1111	Configuration Register load instruction
0000 0000 0000 0000 0000 0001 00	Delayed Branch instruction (no pipeline flush)
0000 0000 0000 0000 0000 0001 01	Delayed Call instruction (no pipeline flush)
0000 0000 0000 0000 0000 0001 10	Delayed Return instruction (no pipeline flush)
0010 00	ALU0-MAC0 instruction
0010 01	ALU0-MAC1 instruction
0010 10	ACU Data Memory 0/1 with immediate instruction
0010 11	ACU Data Memory 0/2 with immediate instruction
0011 00	ACU Data Memory 0/3 with immediate instruction
0011 01	ACU Data Memory 1/2 with immediate instruction
0011 10	ACU Data Memory 1/3 with immediate instruction
0011 11	ACU Data Memory 2/3 with immediate instruction
0100 00	MAC0-MAC1 instruction
0100 01	MAC0 with Immediate instruction
0100 10	MAC1 with Immediate instruction

0100 11	ALU0 with Immediate instruction
0101 00	Scratch Memory/Pointer Cache with immediate instruction
0110 00	ACU configuration instruction
0110 01	MAC0-MAC1-ACU Instruction
011010	ALU-ACU instruction

Figure 37.1 - Instructions in Numerical Order


4444444433333333322222222211111111100000000000
76543210987654321098765432109876543210987654321O
PROG FLOW
00000000000000000000000
PROG FLOW
CONDITION CODE
ADDRESS
ITERATE 0
00000000000000000000000011110101
IMMEDIATE ITERATION COUNT
ITERATE 1
00000000000000000000000011110110
IMMEDIATE ITERATION COUNT
REPEAT
00000000000000000000000011110111
IMMEDIATE REPEAT COUNT
DMA CONF
00000000000000000000000011111110
IMMEDIATE DMA CONFIGURATION VALUE
CON FIG
00000000000000000000000011111111
IMMEDIATE CONFIGURATION VALUE


44444444333333333222222222111111110000000000
76543210987654321098765432109876543210987654321O
A0,M0
001000
A0 F SEE
A0 OPCODE
A0 S SEE
A0 X SEE
A0 Y SEE
A0 R SEE
ALU0 REG UPDATES F SH SL YH YL XH XL
M0 F SEE
M0 OP
M0 R SEE
M0 REG UP F S Y X
M0 S SEE
M0 X SEE
M0 Y SEE
A0,M1
001001
A0 F SEE
A0 OPCODE
A0 S SEE
A0 X SEE
A0 Y SEE
A0 R SEE
ALU0 REG UPDATES F SH SL YH YL XH XL
M1 F SEE
M1 OP
M1 R SEE
M1 REG UP F S Y X
M1 S SEE
M1 X SEE
M1 Y SEE
M1,M0
010000
M1 F SEE
M1 OP
M1 R SEE
M1 REG UP F S Y X
M1 S SEE
M1 X SEE
M1 Y SEE
000000
M0 F SEE
M0 OP
M0 R SEE
M0 REG UP F S Y X
M0 S SEE
M0 X SEE
M0 Y SEE
M0 IMM
010001
M0 F SEE
M0 OP
M0 R SEE
M0 REG UP F S Y X
M0 S SEE
M0 X SEE
M0 Y SEE
00000000
IMMEDIATE VALUE
M1 IMM
010010
M1 F SEE
M1 OP
M1 R SEE
M1 REG UP F S Y X
M1 S SEE
M1 X SEE
M1 Y SEE
00000000
IMMEDIATE VALUE
A0 IMM
010011
A0 F SEE
A0 OPCODE
A0 S SEE
A0 X SEE
A0 Y SEE
A0 R SEE
ALU0 REG UPDATES F SH SL YH YL XH XL
00
IMMEDIATE VALUE


44444444333333333222222222111111111000000000
76543210987654321098765432109876543210987654321 0
SM IMM
010100
SM OP CODE
SM UPDATES A W R E S
SM BNK I REG
SERW SEE
SM B0 SEE
SM B1 SEE
SM B2 SEE
SM B3 SEE
IMMEDIATE VALUE
B0B1 IMM
001010
B0 OP CODE
B0 UPDATE D W R C
B0 PTR
B0 CP
B0 D SEE
B1 OP CODE
B1 UPDATE D W R C
B1 PTR
B1 CP
B1 D SEE
IMMEDIATE VALUE
B0B2 IMM
001011
B0 OP CODE
B0 UPDATE D W R C
B0 PTR
B0 CP
B0 D SEE
B2 OP CODE
B2 UPDATE D W R C
B2 PTR
B2 CP
B2 D SEE
IMMEDIATE VALUE
B0B3 IMM
001100
B0 OP CODE
B0 UPDATE D W R C
B0 PTR
B0 CP
B0 D SEE
B3 OP CODE
B3 UPDATE D W R C
B3 PTR
B3 CP
B3 D SEE
IMMEDIATE VALUE
B1B2 IMM
001101
B1 OP CODE
B1 UPDATE D W R C
B1 PTR
B1 CP
B1 D SEE
B2 OP CODE
B2 UPDATE D W R C
B2 PTR
B2 CP
B2 D SEE
IMMEDIATE VALUE
B1B3 IMM
001110
B1 OP CODE
B1 UPDATE D W R C
B1 PTR
B1 CP
B1 D SEE
B3 OP CODE
B3 UPDATE D W R C
B3 PTR
B3 CP
B3 D SEE
IMMEDIATE VALUE
B2B3 IMM
001110
B2 OP CODE
B2 UPDATE D W R C
B2 PTR
B2 CP
B2 D SEE
B3 OP CODE
B3 UPDATE D W R C
B3 PTR
B3 CP
B3 D SEE
IMMEDIATE VALUE
ACU CFIG
01100000
B3 PTR
B3 CP
B3 D SEE
B2 PTR
B2 CP
B2 D SEE
B1 PTR
B1 CP
B1 D SEE
B0 PTR
B0 CP
B0 D SEE
B3 UPDATE D W R C
B2 UPDATE D W R C
B1 UPDATE D W R C
B0 UPDATE D W R C
M1,M0 ACU
01100100
M1 REG UP F S Y X
M0 REG UP F S Y X
M1 OP
M0 OP
B3 OP CODE
B2 OP CODE
B1 OP CODE
B0 OP CODE
B3 UPDATE D W R C
B2 UPDATE D W R C
B1 UPDATE D W R C
B0 UPDATE D W R C
A0 ACU
0110100
A0 OPCODE
0
ALU0 REG UPDATES F SH SL YH YL XH XL
B3 OP CODE
B2 OP CODE
B1 OP CODE
B0 OP CODE
B3 UPDATE D W R C
B2 UPDATE D W R C
B1 UPDATE D W R C
B0 UPDATE D W R C

38 Appendix – Using the Asm85 Assembler

38.1 Requirements

Asm85 is strictly text based, written in PERL, and best run in a unix-type environment. (Windows native versions of PERL exist but have not been tested for this program.) Thus:

38.1.1 Operating Environment

Cygwin must be installed.

Don't worry: it's free. (Cygwin/XFree86 is a port of XFree86 to Cygwin; Cygwin provides a UNIX-like API on the Win32 platform. As of 2003-01-01 the supported Win32 platforms are Windows 95, Windows 98, Windows Me, Windows NT 4.0, Windows 2000, and Windows XP.) Visit http://www.cygwin.com/ for directions and download information. It's easy!

The PERL language must be installed - usually it is /usr/bin/perl - make sure that /usr/bin/ is in your path.

The Parse::RecDescent and Data::Dumper packages also must be installed.

In all likelihood, you'll get the current version of PERL and support packages when you download the cygwin package. At the very least you will get an acceptable version of what you need.

38.2 Invocation

Open a Cygwin bash shell window.

Change directory to where you keep your source code and a copy of the Asm85.pl file.

To assemble your source file named *src_file* you type the command:

```
perl asm85.pl -i src_file
```

Asm85 will print the instruction words according to your source code.

38.2.1 Program options

The program has some options which may be used. The syntax for invoking Asm85 is

```
perl asm85.pl -i src_file [-l] [-d] [-h] [-s]
```

The parameters in [brackets] are optional. They have the following function:

parameter	function
-d	Ignore the source file specified with the –i option and run Asm85 using the internal test code instead.

-h	Print the help banner and exit. This option overrides all other options.
-l	List the source code and address information along with the instruction words produced.
-s	Print the source code file first, then process it.

38.2.2 Normal output example

The normal output from Asm85 is just one instruction word per line, printed in hex (no 0x prefix). It might look like this:

```
$ perl asm85.pl -i src_file

2c0f00000000
207001fc0f00



end of source at LINE 27
$
```

38.2.3 Listing mode output example

The list mode output from Asm85 is *verbose*. Before the instruction word is printed in hex (no 0x prefix), the address at which the instruction resides is printed in [braces]. The line number and each line of source code is also printed. Note that in the current version of Asm85, the line(s) of source code which produce a machine instruction may be printed immediately before or after the instruction. (sorry.)

Using the same source file as in the previous example, but with some blank lines removed from the output, it might look like what you find on the next page.

```
$ perl asm85.pl -i src_file -l


                                0:
                                1: //
                                2: // my sample code with comments to init
                                3: // the alu and mac regs to zero
                                4: //
                                5:
                                6: // zero the pimm reg and all the mac1 registers
                                7:
[0000]  2c0f00000000

                                8: pimm = 0x0000,
                                9:   m1_x(&pimm), m1_y(&pimm), m1_f(&pimm), m1_s(&(long)pimm),
                               10:     (m1_x*m1_y),
                               11:       m1_x(next), m1_y(next), m1_s(next), m1_f(next);
                               12:
                               13: // now do all the alu regs and all the mac0 registers
                               14:
[0001]  207001fc0f00

                               15: a0_x(&pimm), a0_y(&pimm), a0_s(&pimm), a0_f(&pimm),
                               16:   (a0_xl & a0_yl),
                               17:     a0_x(next), a0_y(next), a0_s(next), a0_f(next),
                               18:   m0_x(&pimm), m0_y(&pimm), m0_s(&(long)pimm), m0_f(&pimm),
                               19:     (m0_x * m0_y),
                               20:       m0_x(next), m0_y(next), m0_s(next), m0_f(next);
                               21:
                               22:
                               23: /* that's all folks.... */
                               24:


end of source at LINE 27
$
```