IV.3.4 THE IGC MICROCODE ASSEMBLER

The IGC is programmed using the IGC microcode asembler *asmpp5*. *Asmpp5* processes an input file of IGC microcode source, described below, and outputs three C header files. The first, *igc_microcode.h*, defines the array *static int igc_microcode[]*, which contains 32-bit words comprising the data for the microcde store. The second, *igc_opcodes.h*, contains macro definitions which generate opcodes (I words) and supplementary opcodes (P words) for specific instructions. The third, *igc_commands.h*, contains macros built on top of the *igc_opcodes.h* macros, which allow the user to specify an IGC command with a single macro. The file *igc_microcode.h* is included in application code which initializes the IGC; *igc_commands.h* and *igc_opcodes.h* are included in application code which generates IGC commands.

IV.3.4 — 1 Types of Instructions.

IGC instructions may be divided into 3 basic types:

1) those which use neither the quadratic expression evaluator (QEE) nor a scalar value
2) those which use the C coefficient as a integer scalar value
3) those which use the QEE result Q(x,y), in one of several ways:
   - a fixed number of bits of Q(x,y) is used (regardless of its magnitude)
   - only the sign-bit of Q(x,y) is used
   - all of Q(x,y), including the sign-bit, is used
   - all of Q(x,y), including a fixed minimum number of bits, and the sign-bit are used

Specific instances of QEE instructions may use the QEE in one of the following 3 modes:

constant mode	$Q(x,y) = C$
linear mode	$Q(x,y) = Ax + By + C$
quadratic mode	$Q(x,y) = Dx^2 + Exy + Fy^2 + Ax + By + C$

IV.3.4 — 2 Input File for Microcode Assembler.

The input file to *asmpp5* consists of specifications of (1) templates for computing P and I word opcodes for IGC instructions, and (2) microcode words. Lines beginning with "#" define opcodes, and all other lines, except comments, define microcode words, with a 1-1 correspondence between input lines and microcode store locations. Any portion of a line lying to the right of a semi-colon (';') (including lines beginning with a semi-colon) is treated as a comment. The opcode specifications and microcode word specifications are interlaced, with the placement of an opcode specification line relative to the microcode word specification lines defining the entry point for the instruction. Multiple opcode specifications may use the same entry point.

Opcode specification.

Opcode lines must begin with the special character '#'. They are of the form:

#NEM(arg1, arg2, ...) [DstAddr:dst] [SrcAddr:src] [AuxAddr:aux] [LpCnt1:lpcnt1] [LpCnt2:lpcnt2]
[FNI:fni] [NoLSB] [MBI] [SCA] [FBITS:fb]

NEM is a mnemonic name for the instruction, and 'arg1', 'arg2', etc. are a set of optional mnemonics for instruction arguments. The remaining fields are also optional. There must be no spaces in the list arg1, arg2, ...

'DstAddr', 'SrcAddr', and 'AuxAddr' specify starting values to be used for the three 8-bit pixel-memory address counters. Thus 3 separate pixel-memory operands (3 sequences of pixel-memory addresses) can be used in an instruction.

'LpCnt1' and 'LpCnt2' are starting values to be used for the two 7-bit loop counters used by the microcode sequencer. 'LpCnt1' may not be specified for an instruction which uses the QEE's; its value is implicitly equal to FNI - 1 (see description of FNI below).

The 'FNI' token indicates that the instruction is one which uses the quadratic expression evaluator output (QEE result); FNI is omitted for an instruction which does not use the QEE. The FNI argument should be set to 1 for instructions which only use the sign-bit of the QEE result, and to 2 for instructions which use the entire QEE result including sign-bit; for instructions which require some minimum fixed number of bits of the QEE result, the argument to FNI specifies this number of bits.

The token 'NoLSB' specifies that the instruction does *not* check for a TRR token representing the LSB of the QEE result; it is used for instructions which use only the sign-bit of the QEE result.

The token 'MBI' specifies that the instruction checks for a TRR token representing the sign-bit of the QEE result; it is used for any of the types of QEE instructions which use the sign-bit.

Together, the FNI, NoLSB, and FNI tokens specify how the QEE result is to be used for a specific instruction (see Section IV.3.4-1 above). However, note that none of these QEE-related tokens are related to the QEE mode (constant, linear, or quadratic), which is specified for separate *instances* of any instruction which uses the QEE, by the user or by a macro in *igc_commands.h.*

The presence of the 'SCA' token indicates that the instruction is one which uses a scalar coefficient, the C coefficient value interpreted as a signed integer. 'FNI' and 'SCA' are mutually exclusive, that is, no instruction can use both the QEE's and a scalar value.

'FBITS:fb' is used to change the value for the number of fractional bits of precision, FBITS; it may be specified only in instructions which do not use the QEE result.

FNI and LpCnt1 are mutually exclusive. If FNI is specified, the value for LpCnt1 is implicitly equal to FNI - 1. If this setting for LpCnt1 is unacceptable, 2 things can be done: (1) LpCnt2 can be used instead, if both loop counters are not needed for the instruction, or (2) the microcode can count LpCnt1 down explicitly, if the required starting count for Loop Counter 1 is less than the desired value of FNI - 1.

The arguments 'dst', 'src', 'aux', 'lpcnt1, 'lpcnt2', 'fni', and 'fb' must be constant expressions involving the 'args' which are interpretable by the C pre-processor, and should use parentheses liberally to avoid incorrect evaluation if the 'args' are complex expressions in invocations of the instructions. Valid ranges for the actual arguments passed in the opcode are as follows:

FIELD	ARGUMENT	MIN	MAX
DstAddr	dst	0	255
SrcAddr	src	0	255
AuxAddr	aux	0	255
LpCnt1	lpcnt1	0	128 †
LpCnt2	lpcnt2	0	128 †
FNI	fni	1††	(75-FBITS)
FBITS	fb	0	30

† range is 0-127 or 1-128 for any given instruction, depending on the how the microcode is written

†† care must be taken if this value is set to 1; see comments under "General Programming Considerations"

Since the specification for the arguments generally are expressions involving the 'args', the assembler cannot do error checking; rather, the code generating IGC commands must insure that valid values will result. Failure to detect errors can result in the IGC hanging.

For each opcode specification line, *asmpp5* places a macro definition in the output file *igc_opcodes.h* of the form

```
I_NAME(args) ...
```

This macro fully defines all the fields of the I register opcode except for the QEEMode and CoefMode fields. The QEEMode field specifies which QEE mode (constant, linear, or quadratic) is to be used; in the macro, it is set to 11 if the instruction uses the QEE's and 00 otherwise. These settings may be changed if it is desired to use the QEE's in constant or linear mode, either manually or by using the macros in *igc_commands.h*; for a non-QEE instruction, they must be left 0's. The CoefMode field is set to 00, and must be specified by the user; this field is ignored by the IGC, but rather is used to specify the number and ordering of coefficient arguments in an instruction stream for an off-chip "stream parser".

If any of the tokens LpCnt2, SrcAddr, or AuxAddr are used, bit 31 of the opcode is set by the I_NAME macro, and a supplementary opcode (the P register) is required. For this supplementary opcode, *asmpp5* generates a second macro of the form

P_NAME(args) ...

which defines the entire P register supplementary opcode. Bit 31 also is ignored by the IGC; it tells the off-chip "stream parser" that the supplementary opcode is expected in its instruction stream.

Microcode word specification.

All other lines in the input file, except blank lines and comment lines (beginning with a semicolon ";") are specifications of microcode words, one line to a microcode word. A microcode specification lines contains a number of optional but inter-dependent fields.They are divided into several groups:

(1) sequencer control - control the sequencer branching
(2) pixel-memory control (address and read/write)
(3) quadratic expression evaluator function
(4) pixel-ALU instruction (including direct register control, configuration and external operations)

Sequencer control is specified using one of the following tokens:

done	terminate this instruction (goto idle, or begin next instruction)
br:n	unconditionally branch to offset n (positive or negative)
TRR:n	increment if TRR set, offset branch otherwise
TC1:n	increment if Loop Counter 1 zero, offset branch otherwise
TC1B:n	increment if Loop Counter 1 non-zero, offset branch otherwise
TC2:n	increment if Loop Counter 2 zero, offset branch otherwise
ST1:n	increment if ST1HLPR input is High, offset branch otherwise
ST2:n	increment if ST2LLPR input is Low, offset branch otherwise
<default>	unconditionally increment (go to next address)

If none of these tokens is specified, the default sequencer control, an unconditional increment, is used.

The 'done' token is special. It specifies termination of the instruction; if another instruction is pending, control will jump to the starting microcode address for that instruction, otherwise control branches to 0, the idle address.

Two additional optional tokens control the loop counters (in module **LoopCount**):

cnt1	decrement loop counter 1 (ignored if 'done' is also specified)
cnt2	decrement loop counter 2 (ignored if 'done' is also specified)

Control of pixel-memory address is by the following tokens:

dst	use Destination address counter (default)
dst+	use Destination address counter, then increment it
dst-	use Destination address counter, then decrement it
src	use Source address counter
src+	use Source address counter, then increment it
src-	use Source address counter, then decrement it
aux+	use Auxiliary address counter, then increment it
aux	use Auxiliary address counter

The default for pixel-memory address control is 'dst'. The token 'aux-' (auxiliary address counter 1 with decrement) is not defined. If any increment or decrement token is specified in a microcode word containing a 'done' specification, the increment or decrement will not occur since the counters will be loaded with the respective starting addresses from the next instruction.

Control of pixel-memory read/write is by the token 'WRT'. If it is not given, the specified pixel-memory address is read on this cycle. If it is given, the value on the 'sum' output of the pixel-ALU is written to the specified pixel-memory location provided either (1) the pixel -ALU Enable register is set, or (2) the ALU instruction specified FrcEn on the previous microcycle (see below). Note that pixel-memory is always read, even on a write cycle.

Quadratic expression evaluator function is controlled by two optional but mutually exclusive tokens:

TrH	do not shift the QEE and coefficient shifters on this cycle
TrF	unconditionally shift the QEE and coefficient shifters on this cycle

The default is to conditionally shift the QEE and coefficient shifters on a given cycle, that is, assert ShiftHSP1 if and only if a TRR token is not being asserted on the "QEE result ready" signal TRRHSP1.

The pixel-ALU instruction is specified by several sets of tokens. To understand use of these tokens, recall that the EMC's pixel-ALU is based on a one-bit full adder, with input multiplexers on the 'a', 'b', and 'c' inputs. The 'sum' output can feed back to the 'a' input, and also becomes the 'write data' bit for pixel-memory; the 'carry' output can be saved in either the Enable register (which qualifies writes into pixel-memory) or the Carry register. Both of these registers also feed back to the adder inputs.

The Carry and Enable registers are controlled using these optional but mutually esclusive tokens:

LdEn	load the Carry register with the 'carry' output of the pixel-ALU on this cycle (not previous cycle); default is to save the old Carry
CRY	load the Enable register with the 'carry' output of the pixel-ALU on this clock cycle; default is to save the old Enable
FrcEn	save the Enable and Carry register contents, but enable writes to pixel-memory during the next cycle (WRT must also be specified), regardless of the contents of the Enable register

The inputs to the adder are specified using the complex token:

alu:a+b+c

where 'a', 'b', and 'c'represent the corresponding inputs to the pixel-ALU.

Selections for 'a' are:

1	a constant 1
0	a constant 0
sum	the 'sum' output of the adder from the previous cycle
sumbar	the inverse of the 'sum' output
tree	the local value of the quadratic expression evaluator (QEE) result
treebar	the inverse of the QEE result
sumtree	the logical-AND of the QEE result and the adder 'sum' output
sumtreebar	the logical-NAND of the QEE result and the adder 'sum' output

Selections for 'b' are:

1	a constant 1
0	a constant 0
enable	the previous cycle contents of the Enable register
enablebar	the inverse of the Enable register
rddat	the value read from pixel-memory on the previous cycle
rddatbar	the inverse of the value read from pixel-memory
ear	the logical-AND of the Enable register and the pixel-memory read
earbar	the logical-NAND of the Enable register and the pixel-memory read

Selections for 'c' are:

1	a constant 1
0	a constant 0
cry	the previous cycle contents of the Carry register
crybar	the inverse of the previous cycle contents of the Carry register

The default is 'alu:sum+0+0'.

Any specification of the form 'alu:a+1+0' is automatically encoded as the functionally equivalent form 'alu:a+0+1'. The configuration of the ALU control bits which would be specified by the 'alu:a+1+0' form is said to be *redundant*, since identical function is

specified by the 'alu:a+0+1' form.

This redundant configuration is used for special functions, and is invoked by using, instead of the form 'alu:a+b+c', one of the tokens:

alu:extN, for N = 0,1,...,5
alu:cfg
alu:cfgbar.

These forms set the 'b' and 'c' inputs to the redundant form (b=1, c=0), and the 'a' inputs are encoded as

N = 4*AGtsTHSP1 + 2*AGtsSHSP1 + ACmpHSP1

with N=6 and N=7 corresponding to 'cfg' and 'cfgbar' respectively. For the 'extN' form, the ExtOp*N*HSPR output of the IGC is asserted for one cycle and is used to specify one of 6 external commands; these are designed for controlling off-chip hardware on the Renderer board.

The setting of the ALU control bits specified by 'cfg' and 'cfgbar' is reserved for activation of the configuration registers on the EMC's.

The forms which cause **ACmpHSP1** to be asserted ('ext1', 'ext3', 'ext5', and 'cfgbar') enable the paired address lines. This means that instead of the IGC outputs **AddrL<0:7>HSPR** and **AddrH<0:7>HSPR** being identical, **AddrH<0:7>HSPR** are the inverses of **AddrL<0:7>HSPR.** Thus each EMC gets a different pixel-memory address.

It is not intended that the pixel-ALU be used in instructions which use these special redundant forms, since the values of the ALU control signals are somewhat arbitrarily defined; however, the pixel-ALU could be used on these cycles if the desired control signals could be resolved with the ones needed to implement a specific external operation or configuration operation.

One additional token

dir

may be specified; this token enables the direct-to-ALU register, used for the 'scalar' type instruction which uses the C coefficient as an integer value. This feature is used to pass integer data directly to the pixel-ALU's without using the QEE. A 32-bit integer value is written to the C register with an instruction. When the 'dir' token is used, the output of the direct-to-ALU register (or its complement) defines the **ACmp** output of the IGC, and the direct-to-ALU register is shifted to the right, so that the next most significant bit of the C coefficient is at the output. The complement of the direct-to-ALU value is used if the **Sequencer** output **ACmpHSP1** is asserted; this occurs if the ALU token, contains the 'bar' suffix, either in the 'alu:ext*bar' or 'alu:cfgbar' forms, or in the 'a' portion of the 'alu:a+b+c' form.

i.e: alu:1+x+y dir passes user 'scalar' bit as 'a' input

IV.3.4 — 3 General programming considerations.

Some important considerations when writing microcode for the IGC:

Microcode word 0 must consist of the single token 'done'.

Loop counters. When implementing a loop using the token TC1:N (or TC2:N), where N is zero or negative, some microcode word in the loop must contain the token 'cnt1' (or 'cnt2'); otherwise, the IGC will hang in the loop, unless the Loop Counter was at 0 upon entering the loop.

It makes no sense to specify either of the loop counter tokens 'cnt1' or 'cnt2' in a 'done' microcode word, since the executing the 'done' word ends the instruction and causes the loop counters to be loaded with the initial count values for the next instruction. If either 'cnt1' or 'cnt2' is specified in the same microcode word as 'done', *asmpp5* prints a warning and ignores the 'cnt' token.

The loop counts may also be used as flags, to control conditional execution of portions of the microcode sequence. That is, the same microcode section may be used to implement more than one instruction, by specifying zero or non-zero values for LpCnt1 or LpCnt2 in the opcode specification, and using 'TC1', 'TC2', and 'br' branches. For example:

```
#INSTRUCTION_A()    LpCnt1:1
#INSTRUCTION_B()    LpCnt1:0
TC1:2 <other microcode tokens>          ; executed for both instructions
br:2 <other microcode tokens>           ; exceuted only for instruction B
<microcode tokens>                      ; executed only for instruction A
<microcode tokens>                      ; executed for both instructions
......
```

Pixel-memory address counters. It makes no sense to specify any of the post-decrement or post-increment pixel-memory address tokens in a 'done' microcode word, since the executing the 'done' word ends the instruction and causes the address counters to be loaded with the pixel-memory addresses for the next instruction. If any post-increment of post-decrement of pixel-memory address is specified in the same microcode word as 'done', *asmpp5* prints a warning and ignores the post-increment or post-decrement.

Control of pixel-ALUs. When none of the pixel-ALU control tokens 'alu:', 'LdEn', or 'CRY' are specified for a microcode word, the pixel-ALU's are effectively in a "no-op" state, since the Carry and Enable registers are both saved, and the Sum register is re-cycled (since the default 'alu' setting is 'alu:sum+0+0').

TRR Tokens in QEE instructions. Every instruction which uses the QEE will cause a TRR token to be generated when the LSB of the tree result reaches the pixel-ALU, unless NoLSB is specified, and if MBI is specified, another TRR token will be generated when the sign-bit of the QEE result reaches the pixel-ALU. It is critical that the instruction test for the corrrect number of TRR tokens (using the TRR:N sequencer control).The TRR test must be used once for a QEE instruction which specifies neither MBI nor NoLSB, or which specifies both MBI and NoLSB, and twice for an instruction which specifies MBI but not NoLSB. It is non-sensical for an instruction to specify NoLSB but not MBI, since no token at all would be generated, so the sequencer cannot be synchronized with the QEE. Instructions which do not use the QEE result must never test for TRR tokens.

Setting FNI for QEE instructions. The setting of the FNI field in a QEE instruction which uses the sign-bit is critical. It must always be at least 1; however, if FNI=1, and all the coefficients are zero or out-of-range, then the two TRR tokens will appear on successive shift cycles, and the sign-bit token is likely to be missed. So for instructions which generate both TRR tokens, the minimum for FNI generally should be 2.

NO, because tree is halted automatically

QEE control tokens. Any loop which contains the TRR:N test must also contain a TrF ("tree force") token. However, in other situations, care must be used when specifying TrF. It is important to realize that when a TRR token appears, that the token will remain for only one shift clock cycle (cycle for which the QEE operates). Normally, the QEE will halt when a token appears, and the token will remain; however, if TrF is asserted the token will disappear after one cycle, and the token will probably be *missed*; in particular, the microcode word which specifies ALU controls, etc for the clock cycle when the MSB of the tree result (or the last bit of the tree result to be used) is at the ALU, must not have TrF set. TRRHSP1 is asserted 2 clock cycles prior to the cycle at which the LSB of the integer part of the tree result appears at the pixel-ALU, and, on an MBI instruction, it is asserted again 1 clock cycle prior to the cycle at which the MSB (sign-bit) of the QEE result appears at the pixel-ALU. If FBITS = 0, the LSB of the whole part of the tree result can be just 2 clock cycles behind the MSB of the previous tree result. Thus, the two tokens, representing the MSB of one instruction and the LSB of the next, can appear on successive shift cycles. (This is seen in the microcode for the 'LOAD' instruction, in which neither of the last two microcode words have TrF specified. If the next to the last word had TrF specified (rather than the default "conditional treestep"), and if FBITS=0, then TRRHSP1 could be asserted on the clock cycle on which the most significant bit of the tree result to be used is at the input to the ALU, which is under the control of the next to last microcode word, Shift would not be inhibited, and the IGC would hang.) In general, TrF should never be asserted except when the TRR:N sequencer control is used. In general, TrF and TrH should be used only for instructions which use the QEE; the default conditional treestep should be used for other instructions.

Using multiple conditional branch conditions. The following events may be used to control conditional branches in an instruction.

(a) occurence of TRR token marking LSB of QEE result
(b) LoopCounter 1 reaching zero
(c) Loop Counter 2 reaching zero
(d) occurence of TRR token marking sign-bit of QEE result

Any instruction that generates more than one of these events must insure that they occur in a known sequence, since only one can be tested for on a given microcycle; it is recommended that the order of enumeration above be used.

Example: An instruction might munge the *entire* QEE result (including sign-bit) with an

operand in pixel-memory; this instruction would be the type that uses the entire QEE result, including a fixed minimum number of bits (equal to the length of the pixel memory operand), plus the sign-bit. A loop might be used perform the QEE/memory operation, and when this loop counts out the memory input to pixel-ALU would be set to 0 and the computation proceed until the sign-bit of the QEE result appears. FNI must be selected to insure that the TRR token marking the sign-bit occurs after the loop is exited; so FNI would probably be set to 2 more than the length of the pixel-memory operand.

IV.3.5 IGC FUNCTIONAL SIMULATOR

The simulator for the IGC is *IGCtst*. It consists of a main routine, which simulates the Stream Parser, and calls the procedure *IGC*, which exercises an instance of *struct IGCtype*.

Each line of input represents one input word and contains 2 fields specified as follows:

(1) the number of clock cycles to wait in between asserting WrEn
(2) a 32 bit integer representing the data inputs D<0:31>HTPR (D0 is LSB)

Since the main routine models the Stream Parser, no register address information is given in the input stream.

Output is placed on standard output and consists of lines of 0's and 1's representing the IGC outputs, including the EMC control signals, as well as some additional diagnostic and status signals. The form of this output is compatible with the input format of the simulator for an array of EMC's, *emcs*.

Normally, output is not produced until FBITS is modified (a P register write occurs with D30 = 1); this suppresses output during the initial microcode loading sequence. However, if any command line argument is specified, *IGCtst* produces output on all cycles.

IGCtst normally is used with the preprocessor *prep*. *Prep* is compiled with the include files *igc_commands.h, igc_opcodes.h* and *igc_microcode.h,* which are generated by the IGC microcode asembler, *asmpp5*. First, *prep* sends input to *IGCtst* to load the microcode store, and sends an FB_FBITS command to set the number of fractional bits for the fixed-point bit-streams to a default value of 12. Then *prep* reads lines from standard input, one IGC instruction per line, generates the correct opcodes (and supplementary opcodes) from the macros in igc_microcode.h, sets the QEE mode and stream parser control bits, and sends the correct stream of command data to *IGCtst* via standard output.

Each input line to *prep* consists of an IGC_ macro (as described in Section III.4.2.3.1). The pointer (first argument of each IGC macro) should be 'p'.