

PROCESSING CIRCUIT AND METHOD FOR VARIABLE-LENGTH CODING AND DECODING

Technical Field:

- 5 The invention relates generally to image processing circuits and techniques, and more particularly to a processing circuit and method for the variable-length coding and encoding of data such as video data.

Background of the Invention:

- 10 Variable-length codes are used to encode many types of data. For example, the popular block-based Motion Picture Experts Group (MPEG) video compression standard encodes video data as variable-length symbols for transmission or storage. In addition, many types of variable-length codes, such as Huffman codes, are lossless.

- 15 Typically, variable-length encoded data is transmitted serially. Therefore, the transmission, reception, and decoding of such data are relatively time consuming as compared with data that can be transmitted, received, or decoded in parallel.

- To decrease the transmission, reception, and decoding times, circuit hardware has been developed to process such data. That is, the architecture of
20 such hardware is configured to efficiently implement the variable-length decoding or encoding process. A problem with such hardware, however, is that it is typically designed for a specific type of variable-length code. Therefore, hardware designed to encode or decode data according to one type of variable-length code may be inefficient or unable to encode or decode data according to another type of variable-
25 length code. But many bit streams such as some MPEG bit streams include bit segments that are respectively encoded according to different variable-length codes. Therefore, decoding hardware often must include multiple circuits each designed to decode bit segments according to a respective variable-length code. Unfortunately, this often increases the size, complexity, and cost of the decoding hardware.

- 30 Another alternative is to program a processor to perform the variable-length encoding or decoding. Therefore, for bit streams using more than one variable-

5

SUMMARY OF THE INVENTION

10

15

'01'

'110'

20

25

30

critical loops in processing variable length data and to save memory by using a compressed set of tables.

The general design intent was to fulfill the following requirements:

- 5 • Handle a High Definition Television (HDTV) MPEG stream at 19.4Mbits/sec into an 8MBit Video Buffering Verifier (VBV) buffer.
- Generate decimated coefficients to display HDTV at MP@ML resolutions
- Simultaneously handle encoding and decoding of Main Profile at Main Level (MP@ML) streams
- 10 • For a task such as the decoding of HDTV MPEG streams, the VLx processor might perform the following types of activities based on the program that it executes:
 - Preprocess an MPEG stream to build structures that define the content of the stream
- 15 • Decode Discrete Cosine Transform (DCT) coefficients
- Create an MPEG stream

The VLx processor is fed bitstreams by Map1000 tasks in one of two ways. It can process data that is placed in the Coprocessor Memory Bank, or it can take input bitstreams through I/O channels that are fed by the Map1000 Data
20 Streamer unit.

The resultant information, decimated bitstreams, or newly constructed streams are transferred back to the MAP1000 through memory transfers or as I/O output bitstreams.

The VLx processor consists of a simple processing engine, a set of
25 dedicated registers, a GetBits engine for handling bitstreams and I/O interactions, optimized access to the FFB for Coprocessor Memory 1 (CM1) access and a way to issue a DsContinue() operation to the Data Streamer.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is a block diagram of a media processing circuit that includes a variable-length encoder/decoder processor according to an embodiment of the invention.

5 Figure 2 is a block diagram of the variable-length decoder/encoder processor of Figure 1 and peripheral circuitry according to an embodiment of the invention.

DETAILED DESCRIPTION OF THE INVENTION

As the digital revolution takes hold and all forms of media such as film, audio,
10 and video become digital, the need for the acquisition, processing, display, storage, and communications of such media data has spurred rapid technology development. By taking advantage of these emerging technologies, many new applications have become possible and existing applications strengthened with improving
cost/performance ratios. Digital video, desktop video teleconferencing, machine
15 vision, digital cameras/camcorders and medical imaging are several such examples. Image and video computing algorithms and their fast implementations are some of the core enabling technologies for such applications. Because of the vast market potential, consumer products employing real-time digital video computing have been generating a great deal of excitement among both manufacturers and consumers.
20 The real-time aspect and consumer-level focus of such systems require high computational power at very low cost and optimal implementations of key algorithms. In prototyping and designing these systems, a programmable approach provides flexibility and adaptability to new applications and changing requirements, which is a definite advantage over specialized hardwired solutions.

25 Figure 1 is a block diagram of a media processor 10 that includes a Fixed Function Block (FFB) 11 and a variable-length coder/decoder (VLx) processor 12 according to an embodiment of the invention. In one embodiment, the processor 10 is a MAP 1000 processor produced by Equator Technologies of Seattle, Washington. Because many billions of operations per second are needed to
30 perform media processing in real time there has been great demand for processors capable of much higher computational power than previously available. The MAP

1000, a revolutionary new low-cost programmable single-chip solution which has been designed to meet the demanding compute requirements of digital media processing including:

- digital video: MPEG2 encoding and decoding, video pre/post filtering, and H.324 video conferencing
- digital audio compression: 5.1 channel AC3 and MPEG2 encoding and decoding
- imaging: Joint Photographic Experts Group (JPEG) encoding and decoding, wavelet transforms
- 3D graphics:
- 2D graphics
- telephony
- digital audio synthesis
- digital audio spatialization

The high degree of computational power available on the MAP 1000 makes it one of the most powerful processors in the world today. The MAP 1000 uses on-chip parallelism via a technique known as instruction-level parallelism to achieve such high computation rates. Instruction-level parallelism allows for multiple Central Processing Unit (CPU) operations to be initiated in a single clock cycle. This is done by having multiple on-chip execution units and/or by partitioning a particular execution unit into multiple smaller units (e.g., a 64-bit Arithmetic Logic Unit (ALU) is split into eight 8-bit ALUs).

The construction or interpretation of media bitstreams such as those used in the MPEG, JPEG, or Dolby AC3 however, is an inherently sequential process.

Each symbol or structure within the bitstream takes its interpretation from the symbols that have preceded it. And the length of each symbol is either known *a priori* based on context or is encoded as a length value within the bitstream preceding the symbol value itself. This means that all of the parallelism of the MAP 1000 would go wasted when performing media bitstream encoding or decoding.

To make the encoding or decoding of media bitstreams vastly more efficient, Equator Technologies has developed the integrated VLx processor 12, which is disposed within the Map 1000 processor 10, to supplement parallel processing of Map 1000's core 14. The VLx processor 12 is dedicated to the processing needs of variable-length data such as that within a media stream. The VLx processor 12 allows flexibility in the processing of incoming bitstreams and in how that information about that bitstream is relayed back to the core 14 of the Map1000. Efficient processing has been achieved by designing the hardware to minimize critical loops in processing variable length data and to save memory by using a compressed set of tables.

The VLx processor 12 is thus essentially a 16-bit sequential RISC microprocessor with many special features to help with bit parsing. By using the VLx for the sequential bit-parsing algorithms, the parallel core 14 is free to run more parallel code efficiently and concurrently with the VLx processor 12. Because the VLx processor 12 is completely programmable, it can be used for other bit-serial conditional tasks such as the acceleration of Viterbi, Reed-Solomon, or JBIG processing. The VLx processor 12 is integrated with the rest of the MAP 1000 on-chip circuits as illustrated in Figure 1.

In one embodiment, the VLx processor 12 is designed meet the following requirements:

- Handle HDTV MPEG stream at 19.4Mbits/sec into an 8MBit VBV (buffer).
- Generate decimated coefficients to display HDTV at MP@ML resolutions.

Figure 2 is a block diagram of the VLx processor 12 of Figure 1 and portions of the FFU 11 of Figure 1 according to an embodiment of the invention. The VLx processor 12 includes a 16-bit CPU 16 and a GetBits (GB) processing engine 18, which communicates with the CPU 16 and which also functions as an input/output (I/O) device. The CPU 16 and GB engine 18 have access to a register file 20, which includes 32 general-purpose registers. The FFB 11 includes a coprocessor memory bank 1 (CM1), which the CPU 16 can access for both instruction prefetch operations

and load/store operations. The GB engine 18 can both consume and produce data on an I/O bus 22.

The CPU 16 operates on the 32 registers in the register file 20, and these registers, although not shown individually in Figure 2, are labeled r0 - r31. Of these 32 registers, the CPU 16 and the GB engine 18 share access to 13 of the registers, r0 - r12, for special table processing. There are also virtual registers (discussed below) which share addressing ports with 4 of the general-purpose registers r0-r3. These virtual registers are read-only, and that they are a view into some state of the machine that can change as a side-effect of instruction processing. Up to two registers in the register file 20 can be specified in an instruction. The first register denoted in the instruction is labeled R1. The second register, if specified, is denoted R2. The positioning of the register determines the interpretation of the virtual register specification.

The R1 register is the destination for result writebacks if the writeback version of the instruction is used. The instruction will also be available in the acc virtual register 24 on the cycle following the execution of the instruction.

The general registers r4 - r31 can be read from and written to by instructions. As stated above, the general registers r0 - r3 share their address port with the virtual registers. These four general registers can only be written by a RAM read (LD.W) instruction. Any attempt to read these register locations will result in a read to the virtual register value. The GB 18 is able to read general registers r0 - r3 as part of DCT processing.

In one embodiment, there is a one-cycle latency on register writeback. This means that the register contents for the modified register are available only for the next instruction in the acc virtual register 24.

As stated above, the virtual registers share addressing ports with four of the general-purpose registers r0 - r3. This means that if these registers are accessed by the CPU 16, the value used is going to come from some state of the machine and not the general register.

The virtual registers cannot be written to. Writeback to a virtual register will cause the general register that shares the virtual register port to be written. The VLx

assembler (not shown) recognizes the virtual registers by name and will flag write or writeback operation attempts to these virtual registers.

The GB 18 uses the registers r0 - r12 in a DCT processing mode. When the GB 18 is accessing these registers, access by the CPU 16 to the register file 20 will result in a register file conflict. Thus, the CPU 16 is not permitted to access the register file during a cycle in which the GB 18 is accessing the register file 20. The registers r0 - r12 are read by the GB 18 so that appropriate lookup tables for DCT-coefficient processing can be addressed. Example code for DCT processing is designed so as to avoid these register conflicts between the CPU 16 and GB 18.

The following table depicts a summary of register use.

TABLE I
Register Summary

Register	Virtual Register Mnemonic	Used by GB as table base for DCT processing	Value when in R1 position	Value when in R2 position
r0	<i>acc</i>	yes	last ALU result	last ALU result
r1	<i>dctSign, run</i>	yes	Gbsign ? 0 : (RF[425]<10:5>) << 1	value of bits 15:11 of last CM load value plus 1
r2	<i>symbol</i>	yes	first 16 bits of GB input buffer interpreted by GB mode	first 16 bits of GB input buffer interpreted by GB mode
r3	<i>isZero, nZero</i>	yes	0 for ALU instructions	GB count of leading 0 or 1 bits in <i>symbol</i>
r4..r12		yes	RF[register]	RF[register]
r13..r24			RF[register]	RF[register]
r25	<i>lev</i>		RF[register]	value of bits 10:5 of last CM load value if call.dct was used
r26..r31			RF[register]	RF[register]

The R1 and R2 positions are described on page 26, section 5 of the proposed data sheet for the processor 10 (Figure 1). This proposed data sheet is attached as Appendix A.

The VLx processor 12 uses coprocessor memory such as CM1, which is located in the FFB 11. In one embodiment, CM1 is a 4KB region of memory. When the FFU 3D2D control register 26 is in a VLD mode, the VLx processor 12 has a one-cycle turnaround to memory requests (either 64-bit instruction fetches or 16-bit loads/stores) and the memory bank CM1 cannot be used by any other component of the FFB 11, such as the 3D accelerator (Figure 1).

The memory region CM1 is only accessible to circuits and components external to the FFB 11, such as the Data Streamer (Figure 1) or PIO controller 28 when the FFU 3D2D control register 26 is in Variable Length Decode (VLD) mode. Thus, requests from the VLx processor 12 to access CM1 take priority over requests from circuit components external to the FFB 11 to use the memory CM1.

The VLx processor 12 addresses the memory CM1 with 16-bit addresses, where 0000 specifies the first 16-data-bit location in CM1 and FFFF specifies the last 16-data-bit location in CM1.

The CPU 16 is now discussed in more detail. The CPU 16 is a 16-bit processing unit that supports simple arithmetic operations (adds, subtracts, shifts) on the 32 16-bit registers in the register file 20. The CPU 16 can also initiate loads/stores from/into CM1. Special instructions control the GB 18. The clock (not shown in Figure 2) for the CPU 16 can be stopped/started via the VLx PIO register (not shown in Figure 2) setting.

The CPU 16 continually processes instructions that have been prefetched into its instruction buffer 30. The instruction buffer 30 holds 8 16-bit instructions in 2 4-instruction subregisters 32 and 34. The CPU 16 initiates instruction prefetch of 4 16-bit instructions (64 bits total) in time to prevent the stalling of the instruction pipeline except in the case of a branch or when three consecutive load/store operations prevent instruction prefetch from happening in time for use. A special mode of the CPU 16 called fastbranch allows the CPU 16 to loop executing the instructions in the instruction buffer 30 without performing any instruction prefetch.

The CPU 16 begins executing instructions at the beginning memory location of CM1 (offset 0 from whatever base address of CM1 is chosen) in response to a PIO reset, and the VLx processor 12 clock (not shown in Figure 2) is enabled at the same time. The CPU 16 decodes and then executes one instruction per cycle, with the results of the instruction execution for arithmetic and logical operations being available in the acc virtual register 24 in the next cycle. Optional register writebacks are done in the following cycle. The VLx CPU pipeline has three stages, as depicted in Table 2.

TABLE II
VLx CPU Pipeline

R/D	Instruction decode, register read
EX	Operation execution
WB	[optional] writeback of results

The instruction prefetch is ongoing, happening once within every 4 cycles unless a branch occurs. In the case of a branch, instruction prefetch for a target instruction and ones following to fill the instruction buffer 30 will occur potentially in 2 consecutive cycles. During a branch, the CPU pipeline will suspend processing for 3 cycles as the branch target is calculated, read from CM1, and then decoded.

The VLx instruction set is described in detail in section 5 of Appendix A. The instructions fall into the following categories:

- Arithmetic operations, logical, and shifting operations such as add, subtract, and, or, xor, shift left, shift rights, and endian swap
- Branch operations, goto, branch on condition, fastbranching
- GB control operations
- CM1 load/store operations

Still referring to Figure 2, the GB 18 is now described. The GB 18 functions both as a large bitshifter and an I/O device. It is designed to help in the encoding or decoding of variable-length code words in a bitstream. In addition, it has special

logic for interpretation of parts of an MPEG2 bitstream so that it assists in DCT coefficient extraction or construction of a motion vector.

The GB 18 has the following capabilities:

- 5 • Perform a per byte bit reversal on incoming bits from the I/O input bitstream (received on the I/O bus 22 via the Data Streamer of Figure 1). This keeps MPEG2 bits handled by the GB 18 in a contiguous order so that the GB 18 can function as a giant shifter.
- 10 • Perform a per byte bit reversal on outgoing bits on the I/O output bitstream (sent to the Data Streamer of Figure 1). This preserves the appropriate endianness for interaction with a Very Long Instruction Word (VLIW) program.
- 15 • Shift in data from the I/O input stream and make the first 16 bits of this data available in a *symbol* virtual register 36 for use by the program, an optionally causing shifted bits to be placed in the output stream.
- 20 • Endian swap the view of the value stored in the *symbol* virtual register 36.
- Use the current value stored in the *symbol* virtual register 36 to lookup the result value in a limited set of hardware encoded MPEG2 tables, and then return the table value to the *symbol* virtual register 36.
- Use the current value stored in the *symbol* virtual register 36 in conjunction with general-purpose registers r0 - r12 in the register file 20 to return an address of CM1 to the *symbol* virtual register 36.
- Splice data into the output stream.
- Save and restore bitstream I/O state information so that other I/O bitstream pairs can be handled.
- 25 The GB 18 is controlled by the CPU 16. The CPU 16 controls the GB 18 by specifying an operation and up to two operands. The operation is specified as the

K1 operand in the VLx instruction. The first GB operand comes directly from the GB instruction in the VLx instruction at location K1. The second GB operand, optional based on operation type, is taken from the K2 or R2 operand value of the instruction. (The K2 and R2 operands are discussed in more detail in the attached Appendix A).

- 5 Some GB 18 operations require additional information. This information may come from the last CPU 16 result state made visible in the *acc* virtual register 24 of the VLx instruction immediately following the one containing the GB operation.

The GB pipeline varies according to the GB operation. There are a couple variants on the pipeline. The first pipeline is one when bits are being shifted out of the input buffer 42. A second pipeline is for splicing bits into the output buffer 44. A third pipeline is for saving/writing the GB configuration. A fourth pipeline is for hardware-accelerated table lookup values. These are not true pipelines in that each state of the pipeline is not manipulating unique resources. Instead, the GB operation should be viewed as being 2 to 4 cycles in latency. A summary of the GB pipelines is shown in Table 3.

TABLE III

GB Bit Shift Operation Pipeline

Decode	Receive and decode VLx CPU directive
Get GBSign	Based on the bitshift length, determine the Gbsign which is used in determining <i>dctSign</i>
Shift and Count	For GB shifting operations, shift the appropriate number of bits from the input buffer; will yield an updated <i>symbol</i> value; count the number of leading one or zero bits depending on how GB is configured; will yield an updated <i>nZero</i> value. This may optionally splice consumed bits into the output buffer 44.
DCT lookup	If in DCT mode, read <i>tbase[nZero]</i> register and calculated table index; yields an updated <i>symbol</i> value. Se "DCT Processing" on page 24.

- 20 Additional details of the VLx processor 12 are discussed below and in Appendix A.

A summary of the processor 12 follows, followed by a description of the operation of the processor 12 including the operation of the CPU 16 and the GB 18.

Features

The VLx processor 12 consists of a simple processing engine (the CPU 16), a set of dedicated registers (the register file 20), a GetBits engine (the GB 18) for handling bitstreams and I/O interactions, optimized access to the FFB 11 for CM1 access and a way to issue a DsContinue() operation to the Data Streamer (Figure 1).

The VLx processor 12 components are summarized in the following table and sections:

10 **Table IV**

Component	Function
16-bit 200MHz CPU 16	processes VLx instructions read from FFB11, CM1
instruction buffer 30	holds 8 16-bit instructions
Registers 20	32 16-bit registers r0 – r31 (4 port); some special purpose; some general purpose
Memory CM1	Optimized access to FFU 11 CM1 RAM; requires that no other FFU unit use CM1 while the VLx is operating.
GetBits engine 18	optimized processing of data received from and sent out over I/O bus 22; sends and receives data 4 bytes at a time

VLx Processor General Operational Overview

The VLx CPU 16 executes instructions that are read from the CM1 RAM into the VLx instruction buffer 30. These instructions set VLx register values,

perform simple arithmetic operations on the contents of registers, read and write information from CM1, allow flow of control in the instruction stream to be modified and control the I/O that occurs through the GB engine 18.

The VLx processor 12 can receive signals from the PIO controller 28.

- 5 These signals allow the VLx clock (not shown in Figure 2) to be turned on and off, the VLx program counter (not shown in Figure 2) to be reset to the first instruction, and I/O input to the GB engine 18 to be enabled or disabled.

Communication with the core block 14 (Figure 1) is via the FFB CM1 connection to the data transfer switch (not shown).

- 10 The VLx GB engine 18 is connected to the I/O Bus 22. The GB engine 18 can interact with up to two Data Streamer buffers (Figure 1) at any one time - one of which supplies input streams, the other of which takes output.

VLx Register Description

- 15 The VLx CPU 16 operates on 32 16-bit registers in the register file 20. These registers are *regular* readable/writeable registers. Some of these registers share addressing ports with the *virtual* registers that can only be read since they are a way of examining state in the system that occurs as a result of execution of other instructions.

- 20 For regular registers, there is a 1 cycle latency on register writeback. This means that the register contents are not available for the next instruction following an instruction that modifies the register. If the value is needed, the following instruction can use the accumulator value in the virtual register *acc* 24 that is always set with the results of a mathematical operation. For example

- 25 # Writeback the results to *zzin* of incrementing *zzin* by 1
 ADD_K(W, *zzin*, 1);
 # *zzin* value not available to the following instruction. Use *acc*
 SHL_K(0, *acc*, 2);
 # *zzin* value is now available.

- 30 There are several *virtual* registers, the most notable of which is the accumulator *acc* virtual register 24. The following table describes these special

registers. Note that the register symbolic names, and not the register index values, should be used within the VLx programs since the actual index values may still be subject to change. Also note that several of these virtual registers have values that are specific to the MPEG2 decoding processing and the

5 interaction with the GB engine 18 for this purpose.

TABLE V

Actual Values Used When Instruction
Operand RI1 References a Virtual Register

Register Index	Register Kind	Assembler - Symbolic Name for Operand	Value Used	Indirect Source
0	virtual	acc	Last result	accumulator
1	virtual	reserved	reserved	GB engine 18
1	virtual	dctsign	GBsign ? 0 (from iszeroR1) : (bits <10:5> of RF[RI2]) << 1	GB engine 18
2	virtual	symbol	GBSymbol	GB engine 18
3	virtual	iszeroR1	0	-
4 ... 12	real	tbase*	RF[RI1]	-
14 ... 31	real		RF[RI1]	-

TABLE VI

Actual Values Used When Instruction
Operand RI2 References a Virtual Register

Register Index	Register Kind	Assembler - Symbolic Name for Operand	Value Used	Indirect Source
0	Virtual	acc	Last result	accumulator

1	Virtual	reserved		
1	virtual	reserved		
2	virtual	symbol	GBSymbol	GB engine 18
3	virtual	nzero	GBnzero	GB engine 18
4 ... 12	real	tbase*	RF[R12]	-
25	virtual	lev	bits <10:5> of RF[25]	-
14 ... 31	real		RF[R12]	-

FFB CM1 Description

The VLx processor 12 requires a prioritized access path to CM1. While the VLx processor 12 is using CM1, CM1 cannot be used by any other FFB 11 unit, such as the 3D accelerator (Figure 1). CM1 is allocated for use by the VLx processor 12 by setting the 3D2D Control Register 26 so that it specifies allocating by the VLx processor 12.

CM1 load and store operations typically have a 1-cycle latency from VLx processor time of issue. However, arbitration to CM1 within the FFB 11 will allow other components of the MAP1000 processor 10 (Figure 1) to access CM1, and this can cause increased latency that is dependent on the size of the external request. The arbitration algorithm always gives preference to the VLx processor 12 over external requests such as made by the Data Streamer (Figure 1). Other components that may have a need to access CM1 while the VLx processor 12 is executing include the Data Streamer which may be pulling or pushing data into or from CM1 as needed by the VLx application.

The VLx processor 12 can issue one memory operation per cycle. Memory operations include instruction prefetch, as well as memory load or store operations triggered as a result of executing VLx instructions that affect memory. For instruction prefetch, the VLx requests memory reads of eight bytes (four 16-bit VLx instructions). For instruction load/store operations, the VLx requests memory operations on 16-bit quantities.

If there are external requests to CM1 required, the VLx application should take this into account and be written such that there are cycles in which no

memory accesses are made to CM1 so that external requests can obtain servicing time, preventing starvation.

VLx CPU Description

5 The VLx CPU 16 executes instructions described later in this chapter.

 The processor operates on the 32 16-bit registers in the register file 20, which are described above. The CPU 16 also can read and write values into the CM1 memory space as described above. Execution of special instructions controls the GB engine 18 and other instructions allow the VLx application to
10 issue a DsContinue() type operation to a descriptor program executing in the Data Streamer (Figure 1).

 Significant elements in the CPU 16 are:

- Program counter (not shown in Figure 2)
- Instruction prefetch buffer 30
- 15 • Instruction execution logic (not shown)
- Accumulator (although the accumulator itself is not shown in Figure 2, the value stored in the accumulator can be read by reading the virtual register *acc* 24)

 The CPU 16 continually processes the instructions that have been
20 prefetched into the instruction prefetch buffer 30. The instruction prefetch buffer 30 holds 8 16-bit VLX instructions. The CPU 16 initiates the instruction buffer prefetch from CM1 in enough time to prevent stalling the instruction pipeline except in the case of branching or execution of more than 2 instructions that operate on CM1 per 4 cycles. Each instruction prefetch loads four valid VLx
25 instructions into the prefetch buffer 30.

 The VLx processor 12 starts executing instructions at the beginning of CM1 (offset 0 from CM1). Instructions are decoded and then executed, one per cycle, with the results of the instruction execution for arithmetic and logical operations being available in the accumulator in the next cycle, and optionally
30 written back to a result register on the subsequent cycle.

 The VLx instructions fall into the following categories:

- Arithmetic operations such as ADD, SUB, AND, OR, XOR, Shift Left Endian swap, Shift Right Endian swap
- Branch operations such as >, ==, Goto, branch on condition, indirect branching, and a fastbranch mechanism
- GetBits control operations
- Memory load store operations.

The CPU's 16-bit instruction words consist of a 5-bit opcode together with several operands. The instruction set includes several classes of operations, and also includes special opcodes for control of and interaction with the GB engine.

The IOIF Fastbranch operation is included to speed looping under certain conditions. It allows the VLx processor's CPU 16 to iterate in place using its internal instruction buffer 30 only. This both eliminates calls to CM1 for instructions and provides a no-delay branch to the top of the instruction loop, which improves speed.

PIOs to VLx Processor

The VLx processor 12 has one 32-bit PIO readable/writeable value with values read or written according to the bits below:

TABLE VII

Bit(s)	Name	Description
0	Run	If set to 1, turns on VLx clock; otherwise turn off VLx clock
1	Step	Step 1 clock cycle
2	ResetPC	Sets PC to 0; marks contents of instruction buffer as invalid (forces out any fastbranch state)
3	Debug	Sets a debug breakpoint. If this bit is set, bits 7-15 specify the breakpoint address.
4	GBCooldown	Turns off request of I/O input on input stream

5	SetRFAddr	If set, use bit 10 to determine whether to read or write the register specified by bits 11-15
7-15	Breakpoint or RAM Address	Breakpoint address if Bit 3 is set; RAM address if neither bit 3 nor bit 5 are set
11-15	RF Addr	Index into register file
7	GBPending	Set if there is I/O outstanding
10	WriteRF	If bit 5 set, if this bit is set, write the value at bits 16-31 to the register specified by bits 11-15; otherwise read the register specified by bits 11-15 and put the value in bits 16-31
16-31	RFDData	Data to write to RF at index specified by bits 11-15 if WriteRF and ForceRF bits are set; otherwise data is *R11.

VLx Processor Pipe Stages

The VLx processor's main stages are as follows:

- Prefetch: 64-bit prefetch into the instruction prefetch buffer 30
- 5 • Decode: 16-bit instruction decode by CPU 16 from buffer and register file address set up
- Execution: instruction execution
- Writeback: write back of results to register in register file 20

Prefetch: The 64-bit prefetch is designed to fetch instruction data into the
 10 instruction prefetch buffer 30 on the assumption that the CPU 16 is executing in-line code. Each new 64-bit word is timed to arrive as the last instruction is loaded into the CPU 16. The CPU 16 can run with no stalls due to missing instructions on in-line code. Note that in one embodiment, the instruction prefetch buffer 30 size is 128 bits (8 instructions * 16 bits per instruction) and 2 prefetch operations
 15 are required to fill the buffer 30.

Decode: The decode stage sets the register file addresses so that data from the register file 20 is ready when the execution stage arrives. Note that the

execution of some instructions, such as READ_GB_x instructions, rely on the register-file addresses being set appropriately by instructions immediately following the READ_GB_x.

1st Stage Execution: The execution uses the data from the register file 20 or supplied constants as operands. The instruction is executed and results are available in the *acc* virtual register 24 in time for use by the next instruction's execution.

Writeback: Though the CPU 16 automatically places the results of arithmetic and logical instructions in the accumulator, if the programmer wishes 10 to have the result be copied to another register, that action must be indicated through the WB field in the instruction. It takes one cycle to write back the results of the execution stage to a register. If a programmer wishes to use the results of an operation in the next cycle the *acc* virtual register 24 is used.

15 Branching

Branches take 3 cycles to execute. All branches are of an absolute nature. These can take an immediate operand (11 bits) or a register value. The CPU 16 does not execute any instructions after a branch op is received until the target is loaded.

20

FAST BRANCH

The fastbranch instruction (IOIF Fastbranch) defines a mode of operation where the CPU 16 halts instruction fetch from CM1 and executes code only within the instruction buffer 30. The advantage of this is that loops can be 25 executed more quickly within code in the buffer 30 than when using code requiring fetches from CM1, since the branch to a user-specified offset occurs without delay in the buffer 30. This also frees up cycles for CM1 to be accessed by the Data Streamer (Figure 1).

The instruction buffer 30 can hold up to eight instructions, and code in the 30 buffer 30 must be aligned on an 8 word boundary. Using the fastbranch

capability requires that the programmer predefine the number of instructions in the body of the loop. This number, whose range is 0 to 3, defines the address within the buffer 30 that the CPU wraps back to once the last instruction (8th in the buffer 30) is reached. This means that 5 to 8 instructions can be included in this loop.

GetBits Engine

The GB engine 18 in the VLx processor 12 is designed to assist the VLx processor 12 in decoding variable-length-coded symbols. GB 18 is designed to perform bit extraction and insertion operations on data received from the I/O Bus 22, and to assist in coefficient extraction. It is responsible for assembling the input data and doing part of the computation necessary to find the correct lookup table.

The GB engine 18 has the following basic capabilities:

- Does a per byte bit reversal on incoming bits from the I/O input stream. This keeps the bits handled by the GB engine 18 contiguous in the GetBits processing.
- Does a per byte bit reversal on outgoing bits on the I/O stream. This preserves the appropriate endian values for interaction with the VLIW core 14 (Figure 1).
- Shifts in data from the I/O input stream and make this data available in the *symbol* virtual register 36, and optionally causing this data to also be placed on the output stream.
- Endian swap the view of the bits in the I/O input stream
- Counts the number of leading zero or one bits starting from the bitstream Most Significant Bit position (first bit received after per byte bit reversal) and makes this count available in the virtual register *nzero* (not shown in Figure 2).
- Can optionally use the current *symbol* value to lookup the resulting value in a limited set of hardware encoded MPEG2 tables, and then return the table value as the *symbol* value.

- Can splice data into the output stream.

The GB engine 18 has two primary interfaces with the VLx CPU 16:

1. Via the register file 20
2. Simple control bits

5 The control bits are:

- Run/Halt (from the VLx CPU 16 to GB 18)
- Done/Busy (from GB 18 to the CPU 16)

Execution of VLx GB instructions by the CPU 16 configures and controls the GetBits operations. In order to perform appropriate configuration,
10 information such as appropriate Data Streamer buffer and channel information must be passed to the VLx processor 12 so that it can configure the GB engine 18.

The GB engine 18 interacts with the I/O bus 22 using two of its slots on the I/O bus 22 at any given time. One of these slots 38 is used for an input
15 stream. The other slot 40 is used for an output stream.

When the GB engine 18 is processing input, input data is moved into an input buffer 42. As GetBits operations are executed, the bits in the input buffer 42 are processed. The GB engine 18 makes read requests on the I/O bus 22 at the I/O bus rate to keep this input buffer 42 supplied with data. The I/O transfer
20 size for input is 4 bytes.

The GetBits engine 18 can only process input data if it has been configured to read input, and if it has a sufficient quantity of unprocessed bits in its input buffer 42. A VLx program configures the GB engine 18 for input, and is responsible for checking to see if there is sufficient bits to process by using a
25 special branch conditional instruction (BRACIND(...C_gbstall)).

Output is generated to an output buffer 44 either through explicit placement of data (for example, through the G_splice) or as a side effect of shifting through symbols in the input buffer 42.

The GB engine 18 can only generate output if it has been configured to
30 write output to the I/O bus 22, and if it has sufficient space in its output buffer 44 to write more bits out. When 4 bytes worth of valid bits have been written to the

output buffer 44, the GB engine 18 will initiate a 4 byte I/O write transfer to the I/O bus 22.

The input buffer 42 size is 112 bits. The output buffer 44 size is 92 bits.

Sufficient input data for GetBits processing exists in the input buffer 42 when

there are at least 40 bits in the buffer 42. The output buffer 44 must have room for at least 21 bits for GetBits processing that affects output buffer state.

VLx Instructions

The CPU 16 is programmed in VLx assembly language. The instruction set is listed below:

Destination Control for Results

For instructions except BRACIND, the following holds true:

If the WB field of the instruction = 0, then the results of the instruction are available from the acc virtual register 24 for arithmetic operations..

If the WB field = 1, then the results of an instruction are copied the register addressed by the RI1 field of the instruction.

For the instruction BRACIND only, the effect of WB is different. For discussion, see description of BRACIND.

Instruction Format

The CPU's 16-bit instruction words have the following structure:

TABLE VIII

VLx Instruction Format

Type	Field 1 - Opcode	Field 2 - Writeback Control	Field 3 - Operand (RI1)	Field 4 - Operand (RI2)
1	5-bit opcode	1-bit WB flag	5-bit operand (register address)	5-bit operand (register address)
2	5-bit opcode	1-bit WB flag	5-bit operand (register	5-bit operand (constant)

			address)	
3	5-bit opcode	1-bit WB flag	5-bit operand (constant)	5-bit operand (register address)
4	5-bit opcode	1-bit WB flag	5-bit operand (constant)	5-bit operand (constant)
5	5-bit opcode	1-bit WB flag	11-bit operand (address or constant)	--

TABLE IX

Operands Used In Instructions

Operand	Function
WB	The Write Bit has two uses: For all opcodes except Bracind, WB controls whether results of the operation are copied to a register in addition to the accumulator: 0 = operation results placed in the accumulator only. 1 = operation results are copied into the register addressed in the RI1 field. For the Bracind opcode, WB functions as follows: 0 = no change to RF[RI1] contents 1 = causes RF[RI1] to be set to the program counter value
RI1	Register Index 1: a 5-bit register address indexing one of the 32 registers in the register file or one of the virtual registers.
RI2	Register Index 2: a 5-bit register address indexing one of the 32 registers in the register file or one of the virtual registers
K5	a 5-bit value
M11	an 11-bit Coprocessor Memory 1 (CM1) memory address
K11	an 11-bit value

Instruction Descriptions

- 5 The format for instruction descriptions is shown below. MNEMONIC is the assembly language mnemonic for the instruction. WB is the value of the WB field. Arg1 and Arg2 are operands as described in Table 6 above. All cycle counts are in VLx clock cycles for a 200 MHz VLx clock.

ADD_K(WB,RI1,K5)

Cycles: 1 for results to virtual register *acc* 24;
2 for results to any other available register (typically in the register file 20).

Function: Add constant K5 and the contents of register RI1.

Example: ADD_K(0,tbase12,0)

Description: The example sums the value 0 and the contents of register *tbase12*. The result of the operation is available in the *acc* virtual register 24 in the following cycle.

ADD_R(WB,RI1,RI2)

Cycles: 1 for results to virtual register *acc* 24;
2 for results to any other register.

Function: Add the contents of register RI2 and the contents of register RI1.

Example: ADD_R(1,pctype,acc)

Description: The example sums the contents of the virtual register *acc* 24 with the contents of the register symbolically addressed as *pctype* and makes the results available in the virtual register *acc* 24 in the subsequent cycle, and in the *pctype* register for use in 2 cycles.

SUB_K(WB,RI1,K5)

Cycles: 1 for results to virtual register *acc*;
2 for results to any other register.

Function: Subtract constant K5 from the contents of register RI1.

Note that SUB may be use to negate a value in one step by using the *iszero* virtual register to supply the constant zero allows one step negation using SUB_K(0,iszero,reg).

Example: SUB_K(0,acc,1)

Description: Subtract 1 from the virtual register *acc 24* and make the results available in *acc 24* for the next instruction.

SUB_R(WB,R11,R12)

- 5 **Cycles:** 1 for results to virtual register *acc 24*;
 2 for results to any other register.
- Function:** Subtract the contents of register *R12* from contents of register *R11*.
- Example:** SUB_R(0,5,3)

SHL_K(WB,R11,K5)

- 10 **Cycles:** 1 for results to virtual register *acc 24*;
 2 for results to any other register.
- 15 **Function:** Shift contents of register *R11* left by a number of bits equal to the value at bit positions 0:3 of *K5*. All shift operations result in zeros being shifted in to fill vacant bits. Note that the value of bit 4 of *K5* controls whether endian swapping is done before the shift operation, as follows:
 Value of *K5[4]* =
 1 : endian swap then shift left by number of bits = *K5[3:0]*
 0 : only shift left by number of bits = *K5[3:0]*.

SHL_R(WB,R11,R12)

- 20 **Cycles:** 1 for results to virtual register *acc 24*;
 2 for results to any other register.
- 25 **Function:** Shift contents of register at index *R11* left by a number of bits equal to the value at bit positions 0:3 in register *R12*. (That is, shifts may be from 0 to 15 bits.) All shift operations result in zeros being shifted in to fill vacant bits. Note that the value of bit 4 of the contents of register *R12* controls whether endian swapping is done before the shift operation:
30 If value of bit 4 of *R12* contents =
 1 : endian swap then shift right by amount = *R12[3:0]*
 0 : only shift right by amount = *R12[3:0]*.

SHR_K(WB,R11,K5)

Cycles: 1 for results to virtual register acc 24;
2 for results to any other register.

Function: Shift right the contents of register R11 by K5 bits. All shift operations result in zeros being shifted in to fill vacant bits. Note that bit 4 of K5 controls whether endian swapping is done before the shift operation, as follows:
Value of K5[4] =
1 : endian swap then shift left by amount = K5[3:0]
0 : only shift left by number of bits = K5[3:0].

Example: See below:

TABLE X
Examples of SHR_K Instruction

OPERATION	DATA	RESULT
	1110 0000 0000 1011	
SHR_K(0,acc,0b10000)	1101 0000 0000 0111	Simple endian swap
SHR_K(0,acc,0b11000)	0000 0000 1101 0000	Swap then shift right 8 bits
SHR_K(0,acc,0b01000)	0000 0000 1110 0000	Simple shift right 8 bits.

SHR_R(WB,R11,R12)

Cycles: 1 for results to virtual register acc 24;
2 for results to any other register.

Function: Shift right the contents of register at index R11 by number of bits equal to the value in the register at R12. All shift operations result in zeros being shifted in to fill vacant bits. Note that the value of bit 4 of the contents of register R12 controls whether endian swapping is done before the shift operation, as follows:
Value of bit 4 of R12 contents =
1 : endian swap then shift right by amount = R12[3:0]
0 : only shift right by amount = R12[3:0].

AND_K(WB,RI1,K5)

Cycles: 1 for results to virtual register *acc* 24;
2 for results to any other register.

Function: Logical AND the contents of register RI1 with the constant K5.

5

AND_R(WB,RI1,RI2)

Cycles: 1 for results to virtual register *acc* 24;
2 for results to any other register.

Function: Logical AND the contents of register RI1 with the contents of register RI2.

10

OR_K(WB,RI1,K5)

Cycles: 1 for results to virtual register *acc* 24;
2 for results to any other register.

Function: Logical OR the contents of register RI1 with the constant K5.

15

OR_R(WB,RI1,RI2)

Cycles: 1 for results to virtual register *acc* 24;
2 for results to any other register.

Function: Logical OR the contents of register RI1 with the contents of register RI2.

20

EXOR_K(WB,RI1,K5)

Cycles: 1 for results to virtual register *acc* 24;
2 for results to any other register.

Function: Exclusive-OR the contents of register RI1 with constant K5.

25

EXOR_R(WB,RI1,RI2)

Cycles: 1 for results to virtual register acc 24;
2 for results to any other register.

Function: Exclusive-OR the contents of registers RI1 and RI2 and place results in the accumulator.

5 COPY OPERATIONS

SETREG(WB,RI1,acc)

Cycles: 1 for results to virtual register acc 24;
2 for results to any other register.

Function: Copy accumulator contents to register RI1.

10 **Example:** SETREG(1,5,acc)

Description: The example copies the contents of the virtual register acc 24 to register 5. Note that WB = 0 causes no action and is functionally equivalent to a NOP.

SET_K(K11)

15 **Cycles:** 1 for results to virtual register acc 24;
2 for results to any other register.

Function: Copy the 11-bit constant K11 into the virtual register acc 24.

Example: SET_K(0b0000001000)

20 **Description:** Set virtual register acc 24 to value 0b0000001000.

READ_R(WB,RI1,RI2)

Cycles: 1 for results to virtual register acc 24;
2 for results to any other register.

25 **Function:** Read value into register RI1 from CM1 location addressed by the contents of RI2.

Example: READ_R(W,resa,acc)

Description: Set value in register *resa* = value in RAM[*acc*]

WRITE_R(WB,RI1,RI2)

5

Cycles: 1 for results to virtual register *acc* 24;
2 for results to any other register.

Function: Write value of RI1 into CM1 at location addressed by the contents of RI2.

BEQ0(K11)

10

Cycles: 3 NOPs until target is loaded.

Function: Branch to location K11 if virtual register *acc* 24 is 0.

Example: BEQ0(0b000011110101)

GOTO(K11)

15

Cycles: 3 NOPs until target is loaded.

Function: Goto location K11.

Example: GOTO(0b00000000011)

Description: Sets program counter to 0b00000000011.

BGT0

20

Cycles: 3 NOPs until target is loaded.

Function: If virtual register *acc* > 0 then branch to location K11.

Example: BGT0(jumploc1)

BRACIND(WB,RI1,K5)

Cycles: 3 NOPs until target is loaded.

Function: The Branch Conditional Indirect instruction provides branch control. The user mask K5 used to determine which conditions to test for the conditional branch. There are four mutually-exclusive groups or conditions. Within a particular condition group, the BRACIND instruction test for up to three conditions simultaneously. All of the simultaneously tested conditions must be true in order for the branch to be taken. The destination will be the address whose value is held in the register file R11. The current program counter value will be written to R11 if WB is set.

WB =0 If branch condition is not detected, R11 content is unchanged.

WB =1 If branch condition is detected, put the current PC value in R11. (This allows setup for return from a subroutine call.)

K5: Mask value comprised of four mutually-exclusive groups. Each group can test for up to three conditions simultaneously.

The test conditions are as follows:

TABLE XI

Cond. Group	Cond. Name	K5 Value	Cond. Description
Group 1	C_gbpend	00100	Whether any IO Bus event is pending
	C_gbloopp	00010	Reserved – special function
	C_dts	00001	Whether a DTS continue is allowed
Group 2	C_gbstall	01100	Whether the GetBits engine is stalled
	C_eq0	01010	Whether current results = 0
	C_4eq0	01001	Whether rightmost 4 bits of current results = 0
Group 3	C_always	10100	Always take branch
	C_8eq0	10010	Whether rightmost 8 bits of current result = 0
	C_lsb	10001	Whether register file input is != 0
Group 4	C_gbstall2	11100	same as Gbstall
	C_lt0	11010	Whether current results < 0
	C_gt0	11001	Whether current results > 0

The following operations can be tested simultaneously:

- C_gbpend, C_gbloop, C_dts
- C_gbstall, C_eq0, C_4eq0
- C_always, C_8eq0, C_lsb
- C_gbstall2, C_lt0, C_gt0.

The way to specify simultaneous testing requires that the corresponding bits be set appropriately. For example, C_neq0 is also defined. This is the or'ing of bits for C_lt0 and C_gt0.

Description: Some examples of BRACIND use are:

- 5 Checking for status of GB. If GB is not ready call stall handler routine.
- Returning from a subroutine call (see routine in line above).
- 10 Precomputing a destination address and using this to control the program flow.

READ_GB_K(WB,K1,K2)

- 15 Cycles: 4 cycles, depending upon K1 value. K1 value of G_hwmot will cause results not to be available until up to 8 cycles later.

Function: Send value of K1 and K2 to the GB engine 18.

READ_GB_R(0,K1,R12)

- Cycles: 4-8 cycles depending on value of K1
- 20 All programmer-controlled configuration and control of the GB engine 18 is done by means of the CPU 16 writing a 16-bit control word directly to the GB engine 18. It uses this value to:

- Set the GB 18 operating mode
- Cause a new symbol to be processed

- 25 The GB engine 18 results are accessed by the CPU 16 through the virtual registers. Types of data that are accessible in this way include:

- symbol buffer contents (16 bits) in virtual register *symbol* 36
- number of leading zeros /ones in virtual register *nzero* (not shown in Figure 2)

GetBits Command Arguments

As mentioned previously, the GB engine 18 is controlled using two instructions, READ_GB_K and READ_GB_R. All GetBits activity is controlled by a constant as the first argument plus a secondary argument of a type as needed. The table below lists commands.

TABLE XII
GetBits Commands

Assembler symbolic reference (K1 value)	K2 or R12 Value	Meaning
G_dct	0	DCT operations
G_revsym	len	Advance input bitstream len bits, return the next 16 bits with bits reversed in virtual register symbol, and the count of leading ones or zeros in the virtual register <i>nzero</i>
G_getsym	len	Advance input bitstream len bits, return the next 16 bits in virtual register symbol and the count of leading ones or zeros in the virtual register <i>nzero</i>
G_align	mask	Align input and/or output bitstreams on byte boundary. The mask value specifies whether input or output or both bitstreams are aligned. Alignment of input bitstream may cause values to be reread. Alignment of output bitstream may cause truncation.
G_setopt	see G_setopt details	Set GB options
G_splice	num	Splice num bits of data into the output bitstream
G_hwacc	see G_hwacc details	Use specified hardware accelerated table for interpreting current symbol value
G_adv2	unused	Shift the input stream by the length computed in the previous G_hwacc G_hwmot, G_hwlum, or G_hwchr function.
G_write	0..15	Read or Set I/O channel information

G_write

The G_write sets and reads the configuration of the GB engine 18 as to what Data Streamer buffers (Figure 1) are affected by the input and output requests and the current state of the input and output processing.

5 When a value of '1' is supplied as the second operation (K2 or RI2 value), the Data Streamer buffer configuration information is written to the GB engine 18. When a value of '0' is supplied as the second operation, the current configuration of the GB engine 18 for this information is returned in the *symbol* register 36.

10 The value to write is taken from the decoded RI1 value of the instruction subsequent to the READ_GB_x in the VLx instruction stream. The value is interpreted as:

bit 15 - If set, do not generate output as input bitstream is shifted.

bit 14 - If set, turn allow output bitstream to be written to output buffer 44.

15 bit 13 - If set, count only what is specified in bit 12. This applies to the counting of leading ones or zeros. If clear, this specifies that the number of leading ones or zeros (depending on value of first bit in current symbol) will be counted and returned in virtual register *nzero* for GetBits operations that cause this counting to occur.

20 bit 12 - If bit 13 is set, this specifies to count only leading zeros if bit 12 is clear, otherwise, count only leading ones.

bits 11:6 - The DS buffer ID from which the input stream is drawn

bits 5:0 - The DS buffer ID to which the output stream is sent.

For example, the following example shows configuration of the input and
25 output streams:

```
# configuration information is at offset L_GBDATA
SET_K(L_GBDATA);
```

read this configuration into register *tmp3*

READ_R(W,tmp3,acc);

1 cycle latency on RAM operation

NOP(0,tmp3,tmp3);

5

write the configuration information

READ_GB_K(0,G_write,1); # Write value 0, read value 1

ADD_R(0,tmp3,tmp3); # Set up port for GB G_write

ADD_R(0,tmp3,tmp3); # Set up port for GB G_write

ADD_R(0,tmp3,tmp3); # Set up port for GB G_write

10

G_setopt

The G_setopt command configures the mode of the GB engine 18. This tells the GB engine 18 information such as whether to count 1's or 0's for the symbol processing, and whether input or output I/O is enabled. The

15

configuration information is passed as the value of K2 or R12.

Configuration is dependent on the setting of particular bits:

Bit 5: set to 0

Bit 4: if set to 1, force DCT processing mode

Bit 3: what to count as leading bits: 0 or 1

20

Bit 2: if 0, count both 0 or 1; if 1 count only what is in Bit 3

Bit 1: if 1, do not read any more of input stream into input buffer 42

Bit 0: if 1, do not generate any output to output buffer 44

For example, the following example shows configuration of GetBits

READ_GB_K(0,G_setopt, 0b00101); # Write output, count 0s

25

G_revsym

Shift K2 or *RI2 bits from the input buffer 42. Return the next 16 bits in the input buffer 42 in bit reversed order into the virtual register *symbol* 36.

5 G_getsym

Shift K2 or *RI2 bits from the input buffer for the next symbol. Return the next 16 bits as current symbol in the *symbol* virtual register 36.

G_align

- 10 Align either the input bitstream or output bitstream or both on the nearest byte alignment, causing truncation on output or re-read on input if not already aligned. Splicing of padding bits prior to use of this instruction is recommended for there to be no loss in data on output.

K2 or *RI2 specified whether to align input or output or both.

- 15 Bit 1: if set, align output bitstream

Bit 0: if set, align input bitstream

G_splice

- 20 Splice in a specified number of bits from a specified value into the output bitstream. The K2 or *RI2 specifies the number of bits to be spliced into the output stream.

Splice data is provided to the GB engine 18 by the result of the instruction immediately subsequent to the READ_GB_x. GetBits splices this data Most Significant Bit first starting at bit 15.

- 25 For example, the following example shows splicing of data into the output stream:

READ_GB_K(0,G_splice,16); # splice 16 0s

SET_K(0); # value is 0

Example 2:

READ_GB_K(0,G_splice,4); # splice 4 bits from the result

ADD_K(0,sdata,0); # using an ALU op to provide the data

5 Example 3:

READ_GB_R(0,G_splice,encsym); # splice len is low 5 bits

SHR_K(0,encsym,8); # moves length out of data

7.5.4.2.1 G_dct

10 This is for DCT processing of symbols. The activity performed by the GB engine 18 occurs in multiple cycles.

G_hwacc

15 The G_hwacc functions causes the GB engine 18 to decode the current virtual register *symbol* value against the specified hardware accelerated MPEG2 table and return the value of that table:

K2 or R2 Value Returned value in virtual register *symbol* 36

G_hwchr	Bits 3:0 contain the length of the decoded symbol
	Bits 15:4 contain the dct_dc_size_chrominance value in UIMSB order (bitreversed) that corresponds with the variable length code in the virtual register <i>symbol</i> 36
G_hwlum	Bits 3:0 contain the length of the decoded symbol
	Bits 15:4 contain the dct_dc_size_luminance value in UIMSB order (bit reversed) that

corresponds with the variable length code in the virtual register *symbol 36*

- G_hwmot** Bits 7:0 contain the signed integer value motion code value for the decode variable length code in the virtual register *symbol 36*
- Bits 15:8 contain the motion residual in UIMSB order (bitreversed). The motion residual length is supplied to the GB engine 18 as the result of the 3rd-6th instruction that follows the READ_GB_x(0,G_hwacc,G_hwmot).
- G_nzpa** Bits 4:0 contain the results of taking the current virtual register *symbol* value, shifting out the number of bits specified by the virtual register *nzero* plus 1. Only the next 5 bits of the symbol are returned in the virtual register *symbol 36*.
- G_mbi** Bits 4:0 contain the length of the decoded symbol
- Bits 10:5 contain the macroblock_type interpreted for I-pictures using the current value of the virtual register *symbol 36* as the variable length code.
- G_mbp** Bits 4:0 contain the length of the decoded symbol
- Bits 10:5 contain the macroblock_type interpreted for P-pictures using the current value of the virtual register *symbol 36* as the variable length code.
- G_mbb** Bits 4:0 contain the length of the decoded symbol
- Bits 10:5 contain the macroblock_type interpreted for B-pictures using the current value of the virtual register *symbol 36* as the variable length code.
- G_mai** Bits 4:0 contain the length of the decoded symbol
- Bits 10:5 contain the macroblock_address_increment using the

current value of the virtual register *symbol* 36
as the variable length code.

G_cbp

Bits 4:0 contain the length of the decoded
symbol

Bits 10:5 contain the coded_block_pattern
using the current value of the virtual register
symbol 36 as the variable length code.

G_adv2

Advance the input stream by the value calculated as a result of the
5 previous G_hwacc, G_hwmot, G_hwlum, or G_hwchr value. The next 16 bits are
returned as the virtual register *symbol* value.

IOIF(W,R11,K5)

Cycles: 1

10 Function: This command is primarily used to send issue directives to
external devices and to control some of the CPU internals.
(IO plus Internal interface)

Example: The IOIF instruction can be used two ways; the mode is
chosen by the value of the K5 operand as follows:
15 K5 = 2: perform DsContinue
K5 = 5: perform Fastbranch operation

DsContinue: For the DsContinue mode, R11 must be a 6 bit value that
indicates the DataStream channel ID that a DTS
Continue will be sent to.

20 IOIF.fastbranch: Fastbranch operation allows the CPU to iterate in
place using its instruction buffer only. This both eliminates
calls to the RAM for instructions and provides a no-delay
branch to the top of the loop. For this mode, a control value
must be placed in the accumulator prior to executing the
IOIF instruction. This control value defines loop size.

25 Example of IOIFfastbranch:

For a loop that executes 6 instructions, the code needs to be specified as follows:

NOP(0,0,0)

Cycles: 1

5 Function: This instruction provides a No-Op.

Example: NOP(0,0,0). (The operands can be non-zero.)

Description: No visible operation is performed.

DCT MODE

10 The GB engine 18 of the VLx processor 12 is capable of extremely efficient variable-length symbol parsing. This works in the following way:

Each variable length symbol is grouped in terms of the number of leading zeros or leading ones. The choice between leading zeros or ones is determined in advance and set as an option to the GB 18.

15 Variable Length Decode (VLD) Lookup Tables (LUTs) (not shown in Figure 2) are then set up with one LUT per symbol group (ie. One LUT for all the symbols with zero leading zeros, one LUT for all symbols with 1 leading one, one LUT for all symbols with 2 leading ones – etc...). For MPEG video, variable-length symbols are used to represent run and level pair, which can be further
20 processed into DCT coefficients. For MPEG2 there are 13 different group (ie 13 different LUTs) required. The VLx processor 12 maintains 13 special registers called tbase 0-12 (not shown in Figure 2). These tbase registers are meant to be set to the addresses of the 13 different LUTs. Each tbase register must be set in the following fashion;

25 Bit 15 => 1 :0 leading 1's or 0's
 0: 1 or more leading 1's or 0's

Bits 14-11: log base 2 of the number of elements in the corresponding
LUT

Bits 10-0: Address of the LUT in CM1

The tables themselves are setup in a special way. This is best described
5 by example:

If we look at all the symbols with 2 leading zeros, we have:

0100s

011s

0101s

10 Where s is the sign bit which follows the symbol (it can be either 0 or 1
depending on the sign of the decoded DCT coefficient).

After the leading zeros, there is always a 1. This one need not be
represented in the LUT since it is redundant. The GB 18 will automatically look
only at the symbol past this redundant 1. Also the sign bit is not decoded via
15 LUT but simply appended after it has been read from the table. The number of
entries in the LUT is determined by looking at the longest symbol in a group.
The bits between the redundant 1 and the sign bit are bit-reversed and then are
used as the index into the LUT by the GB 18. In our example, the longest
symbol is 0100s. There are 2 bits between the redundant 1 and the sign bit.
20 This the LUT for symbols with 1 leading zero will have a 2-bit index (ie there will
be 4 entries). In constructing the table, all symbols shorter than the longest
symbol should be padded with don't care bits. In our example we then have:

0100s

011sx

25 0101s

Each entry in the LUT is 16 bits and encodes the run, level and symbol
length, which correspond to the symbol. The run is encoded in 5 bits, the level in
6, and the length in 5 bits. The actual LUT in this case thus looks like:

LUT INDEX 0: 0000000001000101 #run= 0 lev= 2 len= 5 #symbol=0100s

LUT INDEX 1: 0000100000100100 #run= 1 lev= 1 len= 4 #symbol=011s

LUT INDEX 2: 0001000000100101 #run= 2 lev= 1 len= 5 #symbol=0101s

LUT INDEX 3: 0000100000100100 #run= 1 lev= 1 len= 4 #symbol=011s

5 Note that the shorter symbols (like 011) are replicated in the table due to the padding of the don't care bits.

Once all the tables are setup, the GB 18 is ready to parse the symbols.

When put into DCT mode (via a READ-GB_K or BRACIND instruction) the GB 18 will look into the bitstream, determine the number of leading zeros (one ones).

10 The GB 18 then determines which LUT to use based on the number of leading zeros. The tbase register (not shown in Figure 2) corresponding to the chosen LUT is used to figure out the LUT address and how many bits in the symbols to use as the LUT index. The index is then extracted and bit reversed. The resulting LUT index is then used to make the lookup. The lookup result is then
15 used to drive the run and the level registers with the correct run and level values which correspond to the symbol being decoded. The length value is used to extract the sign bit and to drive the dct sign register with the value of the sign bit. Finally the length value is used to advance the bitstream to a new symbol.

20 Escape codes and EOB are denoted by setting level to 0. If the GB 18 is being put into DCT mode via the BRACIND instruction, then the branch will actually take effect only when level is zero. In this way a DCT tight loop can be setup such that a branching outside the loop occurs on EOB or escape code.

From the foregoing it will be appreciated that, although specific
embodiments of the invention have been described herein for purposes of
25 illustration, various modifications may be made without deviating from the spirit and scope of the invention.

APPENDIX A

CHAPTER 2 *Variable Length Decode/Encode Unit*

PRELIMINARY

1. Abstract

This chapter describes the Map1000's Variable Length Decoder/Encoder Unit (VLx). The VLx is a programmable unit optimized for interpreting or creating bitstreams with variable-length encoded data.

The VLx supplements the VLIW nature of processing on the Map1000 core. It is highly effective at processing or creating bitstreams as used in MPEG or JPEG with good performance.

Within this document, the VLx architecture and instruction set are described.

2. Introduction

This chapter provides a description of the Map1000's Variable Length Encode/Decode (VLx) Unit, which handles creation or processing of bitstreams. The VLx Unit is comprised of a 16-bit CPU, an I/O device -- the GetBits engine -- that functions as a very large bit shifter, and access to coprocessor memory that can be shared with the Map1000 VLIW core.

2.1 Definitions of Terms and Notation

The following notation is used frequently in this chapter.

acc	The VLx virtual register holding the result of the last arithmetic or logical instruction
arithmetic coding	A lossless data compression technique
CM	MAP1000 coprocessor memory; the VLx uses coprocessor memory bank 1
CPU	The central processing unit of the VLx which supports simple arithmetic and logical operations, control operations, load/store operations, and interactions with the GB
DCT coefficient	The amplitude of a specific cosine basis function; the VLx can be programmed to extract these from MPEG bitstreams as part of MPEG2 decode
DS	Map1000 Data Streamer
fastbranch	VLx CPU loops on instructions already within the instruction buffer and does not do instruction prefetch from CM
FFU	MAP1000 coprocessor unit; the VLx shares use of CM1 with the 3D graphics accelerator
GB	GetBits engine; the I/O device portion of the VLx that acts as a large bitshifter and which has specialized MPEG2 decode features
JPEG	Joint Photographic Experts Group

HDTV	High Definition TV
Huffman coding	a lossless data compression technique
MP@ML	Main Profile at Main Level; profiles and levels provide an identification of the subset of MPEG2 syntax and semantics, and bit rates through the VBV; see H.262 for MP@ML constraints.
MPEG	Motion Picture Experts Group
MPEG2	See ITU H.262 or ISO/IEC 13818-2
register	a general purpose register
RF	VLx register file containing 32 16-bit general purpose registers
symbol	a variable length sequence of bits that have a meaning in a given context in a bitstream; see VLC; a VLx virtual register that is the first 16 bits of the I/O input buffer that can optionally be viewed in VLC represents depending on GB operation
tbase register	a general purpose register that holds the index to a software table used by the GB for DCT processing. Registers r0:r12 are the tbase registers.
virtual register	a read only register value that is a view into some state of the GetBits engine or the CPU; the view is modified by execution of instructions
VBV	video buffering verifier; a buffer that provides a constraint on the variability of the data rate that an encoder may produce or a decoder consume
VLC	variable length coding; a reversible procedure for coding that assigns shorter code-words to frequent events and longer code-words to less frequent events
VLd	Variable Length Decode
VLe	Variable Length Encode
VLx	MAP1000 Variable Length Decode/Encode Unit

2.2 Reference List

- [MPEG2] ITU-T H.262 or ISO 13818-2 *Information Technology - Generic Coding of Moving Pictures and Associated Audio Information: Video*
- [MPEG4 CD] ISO/IEC 14496-2 *Information Technology - Coding of Audio-Visual Objects: Visual*

2.3 Design Philosophy

The VLx engine is designed to assist the MAP1000 in decoding or encoding variable length coded symbols such as those found in an MPEG bitstream. This decode/encode process is inherently sequential. Each symbol or structure within the bitstream takes its interpretation from the symbols that have preceded it. And the number of bits that comprise each symbol is either known *a priori* based on context, or is encoded as a length value within the bitstream, preceding the symbol value itself. For example, the following figure shows an extract of an MPEG2 bitstream:

'...0000 0000 0000 0000 0000 0001 0000 1101 0011...'

'0000 0000 0000 0000 00001'b start code prefix

'01'b slice start code, macroblock type, or a number of other interpretations depending on context

'00'b picture start code, luminance value of 1, chrominance value of 0, or a number of other interpretations depending on context

'10'b macroblock type in B-picture, chrominance value of 2, or a number of other interpretations depending on context

'110'b luminance value of 4, chrominance value of 3, DCT coefficient value of 0 run, level 2, or a number of other interpretations depending on context.

Figure 2-1. Example of MPEG2 symbol values

Data compression schemes, such as run-length encoding, Huffman encoding or arithmetic encoding, use variable length codes that represent sequences of values in order to compress data in a lossless

way. Video compression standards, such as MPEG2, use these techniques to compress data (encode) prior to transmission, and decompress (decode) data after transmission.

The VLx is a programmable engine dedicated to the processing needs of encoding or decoding variable length data such as that within an MPEG bitstream. The programmable nature allows an encoder or decoder programs to be written or adapted over time to support new features or uses. The VLx provides flexibility in the processing of multiple bitstreams and communicating with the Map1000 VLIW core. Efficient processing has been achieved by designing the hardware to minimize critical loops in processing variable length data. In addition special purpose hardware supports MPEG2 decode functionality.

For a task such decoding HDTV MPEG bitstreams, the VLx can be programmed to perform the following types of activities

- Preprocess the MPEG bitstream to build structures that define the higher level syntactic structure of the stream: picture, slice, macroblock, etc.
- Decode DCT coefficients
- Filter a bitstream
- Create an MPEG bitstream

The design intent was to handle the following requirements:

- preparsing HDTV MPEG streams at 19.4 MBits/second into an 8MBit VBV
- generating decimated coefficients to display HDTV at MP@ML resolutions
- simultaneously handle encoding and decoding of MPEG streams at MP@ML
- simultaneously handle multiple streams

2.4 Features / Major Functional Characteristics

VLx characteristics are summarized in the following table and sections.

Component	Function
16 bit CPU	Processes VLx instructions read from Coprocessor Memory (CM1)
instruction prefetch buffer	holds 8 16-bit instructions
register file	32 16-bit registers; some of these are true general purpose registers; some of these are shared between the CPU and GB
virtual registers	read only view into special state of the CPU, GB input buffer, and CM load staging area
I/O Buffers	buffers for data transfer over the I/O Bus; I/O transfers are in 4 byte quantities
GetBits engine	Logically a large shifter processing data sent and received over the I/O bus; also includes special purpose hardware for return of hardware-encode values for specific MPEG2 VLCs
Memory	Optimized access path to Coprocessor Memory (CM1), requiring that no other FFU unit use CM1 while the VLx is operating (other units external to the FFU can continue to access CM1, but VLx will have prioritized access)

Table 2-1 Map1000 VLx characteristics

3. Background

The video decode process reconstructs video frames from coded bitstreams. This process is effectively done by developing a program for the VLx that handles the sequential processing (as for example, bitstream parsing or decoding of DCT coefficients) and using this in conjunction with one or more VLIW core programs that handle other parts of the decode process.

The VLx with a decoder program running, is one of the first consumers of a coded bitstream. The bitstream can be prepared so that high level syntactic structures such as MPEG2 picture, slice or macroblock can be interpreted. For the VLx MPEG2 decoder program, this is a matter of shifting through the incoming bitstream and building up a resultant higher syntactic structure that can be shared with the VLIW program.

The VLx also supports DCT coefficient decoding and other MPEG2 table lookup functionality through special purpose functionality in the GetBits engine.

3.1 Architectural Position of the VLx Within Map1000

Significant connections of the VLx with other elements within the Map1000 are shown in (the figure) below:

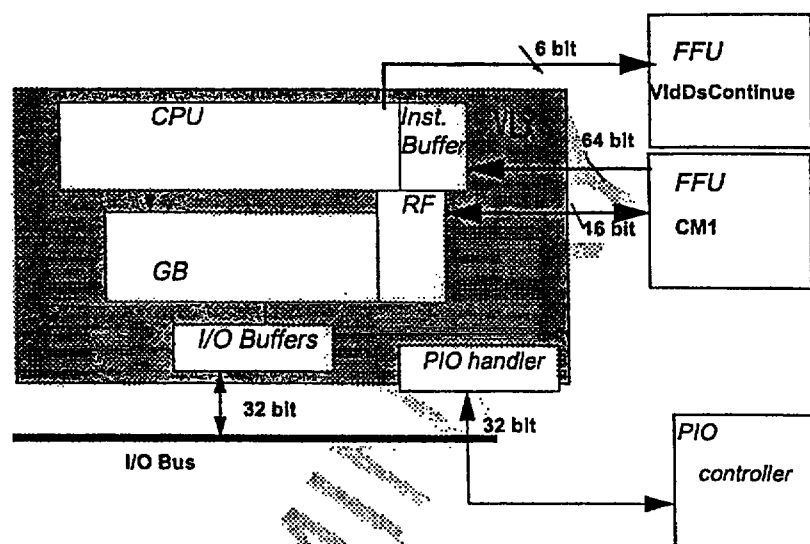


Figure 3-1. VLx Within the Map1000 Architecture

4. Overview of VLx

This section introduces the high-level organization of the VLx and identifies and summarizes its main components. VLx internal structure and operation are discussed in a separate section due to the complexity of this element.

4.1 VLx Internal Organization

The following diagram shows the high-level internal architecture of the VLx Unit.

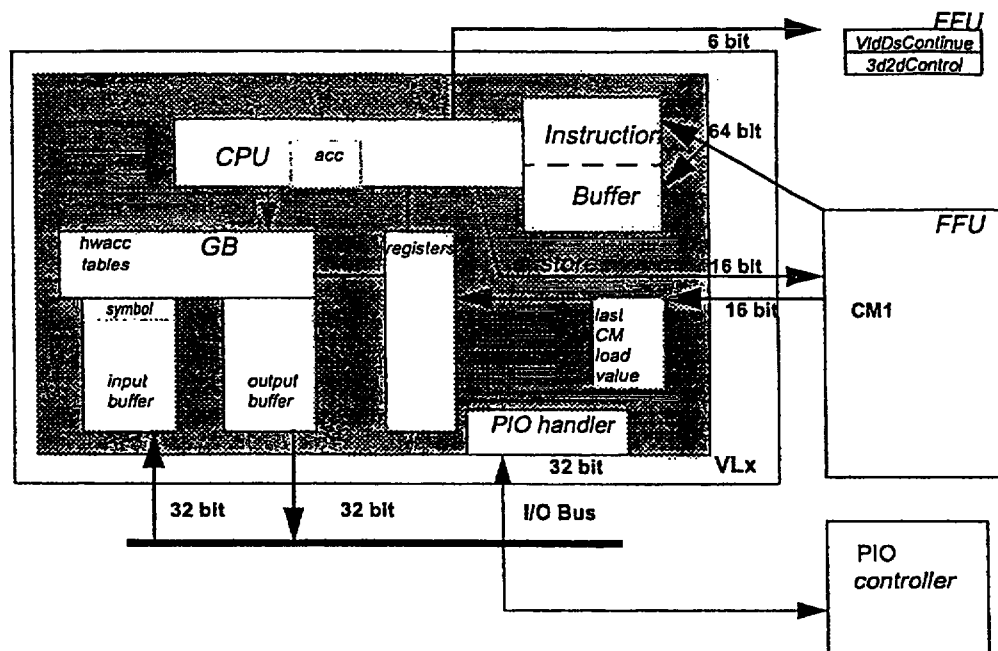


Figure 4-1. Top-Level Organization of the VLx Unit

4.2 Overview of VLx Components

The VLx consists of 16-bit processor and a GetBits engine which communicates with the VLx CPU and which also functions as an I/O device. The CPU and GetBits engine have access to a register file of 32 general purpose registers. The CPU can access CM for both instruction prefetch operations and load/store operations. The Get-Bits engine can both consume and produce data on the I/O bus.

4.2.1 Registers

The VLx CPU operates on 32 16-bit registers: r0:r31. Of these 32 registers, the VLx CPU and GB share access to 13 registers (r0:r12) for special table processing. There are also virtual registers which share addressing ports with four of the general purpose register (r0:r3). These virtual registers are read-only, in that they are a view into some state of the machine that can change as a side-effect of instruction processing.

4.2.1.1 REGISTER SPECIFICATION IN INSTRUCTION

Up to two registers can be specified in an instruction. The first register is denoted R1. The second register, if specified is denoted R2. The positioning of the register determines the interpretation of virtual register specification.

The R1 register is the destination for result writebacks if the write-back version of the instruction is used. The result will also be available in the *acc* virtual register on the cycle following instruction

4.2.1.2 GENERAL REGISTERS

The general registers r4:r31 can be read and written by instructions. Registers r0:r3 share their address port with the virtual registers. These four general registers can only be written by RAM read (ld.w) instructions. Any attempt to read these register locations will result in a read to the virtual register value. GB is able to read general registers r0:r3 as part of the DCT processing.

There is a once cycle latency on register writeback. This means that the register contents are for the modified register are only available for the next instruction in the *acc* virtual register.

4.2.1.3 VIRTUAL REGISTERS

The virtual registers share addressing ports with four of the general purpose registers (r0:r3). This means that if these registers are accessed by the VLx CPU, the value used is going to come from some state of the machine and not the general register.

Virtual registers cannot be written. Writeback to a virtual register will cause the general register that shares that virtual register port to be written. The VLx assembler recognizes the virtual registers by name and will flag write or writeback operation attempts to these virtual registers.

4.2.1.4 USE OF GENERAL REGISTERS BY GB

The GB uses registers r0:r12 in DCT processing mode. When the GB is accessing these registers, access by the VLx CPU to the register file will result in a register file conflict. Thus, the VLx CPU is not permitted to access the register file during that cycle. Registers r0:r12 are read by the GB so that appropriate lookup tables for DCT coefficient processing can be addressed. Example code for DCT processing is designed so as to avoid these conflicts.

4.2.1.5 REGISTER SUMMARY

The following table depicts a summary of register use. The R1 and

Register	Virtual Register Mnemonic	Used by GB as table base for DCT processing	Value when in R1 position	Value when in R2 position
r0	<i>acc</i>	yes	last ALU result	last ALU result
r1	<i>dctSign, run</i>	yes	GBsign? 0 : (RF[r25]<10:5>) << 1	value of bits 15:11 of last CM load value plus 1
r2	<i>symbol</i>	yes	first 16 bits of GB input buffer interpreted by GB mode	first 16 bits of GB input buffer interpreted by GB mode
r3	<i>isZero, nZero</i>	yes	0 for ALU instructions	GB count of leading 0 or 1 bits in <i>symbol</i>
r4..r12		yes	RF[register]	RF[register]
r13..r24			RF[register]	RF[register]
r25	<i>lev</i>		RF[register]	value of bits 10:5 of last CM load value if call.dct was used
r26..r31			RF[register]	RF[register]

Table 4.1 Register Summary

R2 positions are described in Section 5., *VLx Instruction Set*, on page 26.

4.2.2 Coprocessor Memory

The VLx uses coprocessor memory located in the FFU. For the MAP1000, this is a 4K region of memory. When the FFU 3D Control register is in vld_mode, the VLx has a one cycle turn-around to

memory requests (either 64 bit instruction fetches or 16 bit loads/stores) and the memory bank cannot be used by any other FFU, such as the 3D unit.

This memory region is only accessible to FFU external blocks, such as the DS or PIO Controller, when the FFU 3D Control register is in vld_mode. VLx requests take priority over other FFU external block requests.

The VLx addresses CM1 in 16-bit quantities, where a quantity of 0 specifies the first 16-bit location in the coprocessor memory bank.

4.2.3 VLx CPU

The VLx CPU is a 16-bit processor that supports simple arithmetic operations (adds, subtracts, shifts) on the 32 16-bit registers described. The CPU can also initiate loads/stores from/into CM. Special instructions control the GB. The clock for the CPU can be stopped/started via the VLx PIO register setting.

The CPU continually processes instructions that have been prefetched into its instruction buffer. The instruction buffer holds eight 16-bit instructions in two four instruction quantities. The CPU initiates instruction prefetch of four 16-bit instructions (64 bits) in time to prevent the stalling of the instruction pipeline except in the case of a branch or when three consecutive load/store operations prevent instruction prefetch from happening in time in time for use. A special mode of the CPU (fastbranch) allows the CPU to loop executing the instructions in the instruction buffer and not do any instruction prefetch.

The VLx starts executing instructions at the beginning of CM1 (offset 0 from CM1 base) on PIO reset and the VLx clock is enabled. Instructions are decoded and then executed, one per cycle, with the results of the instruction execution for arithmetic and logical operations being available in the virtual register *acc* in the next cycle. Optional register writebacks are done in the following cycle.

The VLx CPU pipeline has 3 stages, as depicted in the table below:

R/D	Instruction decode, register read
EX	Operation execution
WB	[optional] writeback of results

Table 4-2 VLx CPU pipeline

The instruction prefetch is ongoing, happening once within every four cycles unless a branch occurs. In the case of branch, instruction prefetch for target instruction and ones following to fill the instruction buffer will occur potentially in two consecutive cycles. During a branch the CPU pipeline will suspend processing for three cycles as the branch target is calculated, read from CM and then decoded.

The VLx instruction set is described in Section 5. The instructions fall into the following categories:

- Arithmetic operations, logical, and shifting operations such as add, subtract, and, or, xor, shift left, shift right, endian swap
- branch operations, goto, branch on condition, fastbranching
- GB control operations
- CM load/store operations

4.2.4 GetBits Engine

The GetBits engine functions both as a large bitshifter and an I/O device. It is designed to help in the encoding or decoding of VLCs in a bitstream. In addition, it has special logic for interpretation of parts of an MPEG2 bitstream so that it assists in DCT coefficient extraction or construction of a motion vector.

The GetBits engine has the following capabilities:

- Perform a per byte bit reversal on incoming bits from the I/O input bitstream (received via the DS). This keeps MPEG2 bits handled by the GetBits engine in a contiguous order so that the GB can function as a giant shifter.

- Perform a per byte bit reversal on outgoing bits on the I/O output upstream (sent to the DS). This preserves the appropriate endianness for interaction with the VLIW program.
- Shift in data from the I/O input stream and make the first 16 bits of this data available to the program in the virtual register *symbol*, optionally causing shifted bits to be placed into the output stream.
- Endian swap the view of the *symbol* value.
- Use the current *symbol* value to lookup the result value in a limited set of hardware encoded MPEG2 tables, and then return the table value as the *symbol* value.
- Use the current *symbol* value in conjunction with registers 0:12 to return a CM address as the *symbol* value.
- Splice data into the output stream.
- Save and restore bitstream I/O state information so that other I/O bitstream pairs can be handled.

The GetBits engine (GB) is controlled by the VLx CPU. The CPU controls the GB by specifying an operation and up to two operands. The operation is specified as the K1 operand in the VLx instruction. The first GB operand comes directly from the GB instruction in the VLx instruction at location K1. The second GB operand, optional based on operation type, is taken from the the K2 or R2 operand value of the instruction. Some GB operations require additional information. This information may come from the last ALU result state made visible in the virtual register *acc* of the VLx instruction immediately following the one containing the GB operation.

The GB pipeline varies according to GB operation. There are a couple variants on the pipeline. The first pipeline is one when bits are being shifted out of the input buffer. A second pipeline is for splicing bits into the output buffer. A third pipeline is for saving/writing

GB configuration. A fourth pipeline is for hardware accelerated table

Decode	Receive and decode VLx CPU directive
Get GBSign	Based on the bitshift length, determine the GBSign which is used in determining <i>dctSign</i>
Shift and Count	For GB shifting operations, shift the appropriate number of bits from the input buffer; will yield an updated <i>symbol</i> value; count the number of leading one or zero bits depending on how GB is configured; will yield an updated <i>nZero</i> value. This may optionally splice consumed bits into the output buffer.
DCT lookup	If in DCT mode, read <i>tbase[nZero]</i> register and calculate table index; yields an updated <i>symbol</i> value. See "DCT Processing" on page 24.

Table 4-3 GB Bit shift operation pipeline

lookup values. These are not true pipelines in that each stage of the pipeline is not manipulating unique resources. Instead, the GB operations should be viewed as being two to four cycles in latency.

4.2.4.1 I/O INTERFACE

The DMA bus is the primary means for moving bitstreams from the DS buffer into the GetBits engine. The GetBits engine uses two slots on the bus, one for output, one for input. The DS buffers that supply and consume data can be configured into the GB.

There is one input buffer and one output buffer. When the GB input is enabled, the GB makes four byte read requests, consuming data from a DS buffer as long as there is room in the input buffer and data to be consumed. A per byte bit reversal occurs in the hardware as bits are placed in the input buffer, thus keeping the bitstream contiguous. As GB operations are executed as a result of VLx program instructions, the bits in the input buffer are shifted out of the input buffer. As soon as there is space in the input buffer for another four byte transfer, another request is made to the DS.

The GB can only process input data if it has been configured to enable input, there is a sufficient quantity of unprocessed bits in the input buffer, and there is room in the output buffer (if output is also

enabled). The VLx program configures the GB for I/O and is responsible for checking to see if there are sufficient bits or space prior to issuing a GB operation that will consume or produce the bits. This check is done using a `br.gbstall` instruction.

Output is generated to the output buffer either through explicit placement of data (e.g., splicing into an output bitstream) or as a side effect of shifting through symbols in the input buffer (filtering or pre-parsing). Output can only be generated if the GB is configured for output and if there is sufficient free space in the output buffer to write more bits into it. When there are at least four bytes worth of bits to send in the output buffer, the GB will initiate a four byte transfer to the DS via the DMA bus. A per byte bit reversal occurs on the DMA write out.

For MAP1000, the input buffer is 112 bits. The output buffer size is 92 bits. Sufficient input exists in the input buffer for shifting when 40 bits have been read. The output buffer must have room for 21 bits for a GB output operation to execute successfully.

4.2.4.2 GETBITS CONFIGURATION

The GB I/O is configured by the VLx program. The program indicates whether or not input or output is enabled, and if so, from or to which DS buffer. The program has the ability to read back the current configuration, which includes not only whether or not the I/O is enabled and which DS buffers are involved, but also the current state of the input and output buffer (e.g., the bits in the buffers and the current bit index).

This allows the VLx program to swap out and restore state for interactions with particular DS buffers. For example, if two buffers were used for the processing of each bitstream (one for input, one for output), multiple bitstreams could be supported by swapping the GB state (saving state to CM and restoring state from CM) at convenient points (for example, between pictures) to support concurrent processing of multiple bitstreams.

4.2.4.3 GETBITS MODES

The GB supports multiple modes of operations. The first is the straightforward consumption of bits (optionally passing consumed bits into the output buffer as well). The pipeline for this can be found in Table 4-3, "GB Bit shift operation pipeline," on page 19. The value of the virtual registers *symbol*, *dctSign*, and *nZero* may be affected. If the GB is processing DCT coefficients (DCT processing), the GB will access the register file to read the base registers in the DCT lookup stage of the pipeline.

A second mode of operation is one where bits are explicitly pushed, or *spliced*, into the output buffer. While in this mode, there is no change to the input buffer. Bits spliced into the output buffer may trigger an I/O write to occur if this causes the output threshold to be crossed. There is no change on any of the GB related virtual register states for this mode of operation.

Decode Splice operation	Receive and decode VLx CPU directive
Get Splice Length	This comes from R2 or K2 on the GB operation.
Splice in Bits	Bits to splice are taken from the virtual register <i>acc</i> , splicing the number of bits into the output buffer in MSB (bit 15 first) order.

Table 4-4 GB Splice pipeline

A third mode of operation is for handling the configuration of the GB itself. In this mode, like in the splice case, a second operand, the *acc* value, is optionally used by the GB. In this third mode, the GB can be configured as to which DS buffers it will interact with across the I/O bus. The I/O state for a particular bitstream can be read or restored. Note that this save/restore has to be done in its entirety to preserve the complete state of the I/O. Another configuration choice is to change the input and/or output buffers current bit position by moving it to a byte aligned boundary (padding with 0s in the input).

buffer, truncating on the output buffer). The *symbol* value is affected when the GB I/O states are being modified.

Decode	Receive and decode VLx CPU directive
Configure GB options	Set options as specified
Count leading zero or one bits	<i>nZero</i> changes if options setting has modified what gets counted
DCT lookup	If in DCT mode, read <i>tbase[nZero]</i> register and calculate table index; yields an updated <i>symbol</i> value.

Table 4-5 GB I/O setting pipeline

Decode	Receive and decode VLx CPU directive
Save state	Which state to save comes from R2 or K2 on the GB operation. Which state information saved is R2 or K2 value plus 1. The state is made available in the <i>symbol</i> register. See Table 4-7
Write new state	Write value of R1 operand of instruction immediately subsequent to GB operation as the state information for GB.

Table 4-6 GB I/O state swap pipeline

The VLx GB I/O state is shown in Table 4-7. The GB I/O state

GB operand 1 value	GB I/O State written	GB <i>symbol</i> result
0	-----	GB control state
1	GB control state	output buffer control
2	output buffer control	opaque (I/O buffer data)
3..13	opaque (I/O buffer data)	opaque (I/O buffer data)
14	opaque (I/O buffer data)	input buffer control
15	input buffer control	GB control state

Table 4-7 VLx GB I/O state includes

- GB control state: leading zeros or leading ones being counted for *nZero*, counting only leading zeros or ones, or both, DS buffer index for input source, DS buffer index for output target (see Table 5-23, "GB Control Bits," on page 47).
- Output buffer control: automatically generate as output, bit position in the output buffer (see Table 5-24, "GB output buffer control," on page 47).
- bits in the output buffer
- bits in the input buffer
- Input buffer control: bit position in the input buffer

A fourth processing mode is support in the GB hardware for lookup of particular MPEG2 table values given the current *symbol* value. This is described in Section 4.2.4.4, *MPEG2 Hardware Accelerated Processing*, on page 23.

4.2.4.4 MPEG2 HARDWARE ACCELERATED PROCESSING

In addition to acting as a large bitshifter and I/O device, the GB supports accelerated processing of MPEG2 bitstreams. First, it supports a DCT coefficient extraction mode. While in this mode, the GB can do lookup in software data tables to determine run, level information from DCT coefficient tables. This is done using a combination of initializing the *tbase* registers (those general registers shared by the GB

and the CPU) and VLC encoded data tables in CM. For more details, refer to Section 4.2.4.5, *DCT Processing*, on page 24.

Other special MPEG2 support includes hardware tables supporting lookup of appropriate value(s) corresponding to VLCs for

- dct_dc_size_luminance
- dct_dc_size_chrominance
- motion_code
- coded_block_pattern
- macroblock_address_increment
- macroblock_type in I-pictures
- macroblock_type in P-pictures
- macroblock_type in B-pictures

For these hardware accelerated lookup functions, the *symbol* value takes its form from the value of the table, NOT the first 16 bits of the input buffer. This is described in detail in Section 5.8.4, *GB MPEG2 Hardware Table lookup operations*, on page 49, and Section 5.8.5, *GB DCT operations*, on page 53.

4.2.4.5 DCT PROCESSING

When GB is processing DCT coefficients, it uses the *symbol* value determined from the shifting operations to generate a lookup into the appropriate software tables (supporting Table 0 - MPEG2 B.14, or Table 1 - MPEG2 B.15), and makes the CM address of the appropriate table entry available as the *symbol* value until the GB mode is reset by another GB operation.

The difference is the DCT lookup stage of processing (see Table 4-3, "GB Bit shift operation pipeline," on page 19). During this stage of processing, the GB

- Reads the register indicated by the value of *nZero*. This is the *tbase* register. The *nZero* value will be a value between 0 and 12. If the value is 0, *tbase0* (r0) is used by GB. If the value is 12, *tbase12* (r12) is used. Likewise, the appropriate register is used for values in between.

- The value stored in the *tbase* register is interpreted as three fields: *tskip*, *tkeep*, and *tbase*. *Tbase* is a pointer to a

15	14:11	10:0
<i>tskip</i>	<i>tkeep</i>	<i>tbase</i>

location in CM that is the base address for a subtable for all the VLCs with *nZero* number of leading 1s or 0s as appropriate. *Tkeep* is used to construct a bitmask that identifies how much of the symbol is to be examined beyond the leading signature: *tkeepMask*, where the number of bits set is specified by the *tkeep* value. *Tskip* identifies whether the first bit is to be counted in computing the table offset. If the subtable only contains leading zeros or ones, but not both, the first bit is ignored in the count of bits. This halves the size of the subtable.

$$nzero^1 = (\text{counting either 1 or 0 bits}) ? nZero : ((nZero == 0) ? 1 : nZero) \quad (\text{EQ 1})$$

- The value that gets returned as the *symbol* value is shown below.

$$symbol = tbase + ((symbol \gg (nzero^1 + tskip)) \& tkeepMask) \quad (\text{EQ 2})$$

NOTE: The *symbol* value on the right side of the equation comes from the first 16 bits of the input buffer. The end result in the *symbol* virtual register is a CM address that can be used to read a table entry value for the VLC in question. The first 16 bits of the input buffer are not changed by this calculation in the operation. They are just interpreted by the mode of the GB.

The setup of the tables to support MPEG2 Table Zero and Table One is described in Section 6.4, *DCT Table Setup*, on page 60.

5. VLx Instruction Set

The VLx CPU is a 16-bit processor. All instruction words are 16 bits. For all instruction formats, a 5-bit field indicates the operation type. Depending on the operation type, there is

- a 1-bit writeback indicator, and two 5-bit register operands, or
- a 1-bit writeback indicator, one 5-bit register operand and one 5-bit constant value, or
- a 1-bit writeback indicator and one 5-bit constant value and one 5-bit register operand, or
- a 1-bit writeback indicator and two 5-bit constants, or
- an 11 bit constant value.

This is summarized in the table below

15	11	10 9	5 4	0	bits
operation	wb	R1	R2		
operation	wb	R1	K2		
operation	wb	K1	R2		
operation	wb	K1	K2		
operation	K11				

Figure 5-1. VLx Instruction Format

R1 and R2 are register addresses identifying either a general register or a virtual register. K1 and K2 are constants. K11 is an 11 bit constant. Wb, if set, indicates that R1 is also a destination register for result writeback.

NOTE: The destination for the writeback is always the general register (r0:r31), and never the virtual register that shares the port. For example, an `add.w acc, r13` will result in `r0 = *acc + *r13`.

5.1 Notation and Other Conventions

VLx instructions are of the form:

instr[.w] [R1] [R2|K2]

instr K11

macro(R2|K2)

instr specifies the instruction

macro specifies an alternative form for instructions targeting GB, or for instructions that take operands from alternative sources

.w indicates that the result will be written back to the register specified in R1 with a two cycle latency

R1 a register source operand, and optionally the destination result

R2 a register source operand

K2 a 5 bit constant source operand

K11 an 11 bit constant source operand

***R1** the value contained in the register specified by R1

***R2** the value contained in the register specified by R2

CM[x] the 16-bit value at the coprocessor memory location specified where **x** is a 16-bit addressing offset from the beginning of coprocessor memory. The **x** value can be ***R1**, ***R2**, **K2**, or **K11**.

&CM[x] a location in coprocessor memory specified by **x**.

5.1.1 Example Operation

The bit settings for the instruction word are shown in the first table. For

15	11	wb	9	5	4	0
0	0	0	0	0	w	R1 R1 R1 R1 R1 R2 R2 R2 R2 R2

this example, the instruction opcode is 5-bit value of 0, indicating a **nop** instruction. The writeback field has a value of either 1 or 0. If the value

is 1, the results of the operation will be written into the general register indicated by the 5-bit register address denoted by the R1.

Instruction Decode Cycle	Instruction Execution Cycle	Instruction Writeback Cycle
example.w R1, R2	acc = example result	R1 = *acc

Table 5-1 Example Instruction

Instructions are presented in the form of tables. The left column is the instruction. The number of columns following the instruction indicate the cycle after which modified register contents are available for use. For example, on the example instruction, the new value is available in the acc register after 1 cycle of latency, whereas the value is available in the R1 register on the subsequent cycle.

5.2 No Operation

5.2.1 nop

15	11	wb	9	5	4	0
0	0	0	0	0	w	R1 R1 R1 R1 R1 R2 R2 R2 R2 R2

Decode	Execution	Writeback
nop		
nop R1,R2		
nop.w R1,R2		R1 = *acc

Table 5-2 Single cycle no operation.

No operation is performed. Optional, register port values are set.

5.3 Register Setting Operations

5.3.1 Set virtual register *acc* to a 11 bit constant

15	11	10													0
1	1	0	0	1	K11	K11	K11	K11	K11	K11	K11	K11	K11	K11	K11

Decode	Execution	Writeback
setacc K11	$acc = K11$	

Table 5-3 Set virtual register *acc*

Sets the virtual register *acc* to a value specified by the 11 bit constant.

5.3.2 Set general register to contents of *acc*

15	11	wb	9												0
0	0	0	0	1	1	R1	R1	R1	R1	R1	0	0	0	0	0

Decode	Execution	Writeback
setreg R1		$R1 = *acc$

Table 5-4 Set general register

Sets the general register specified by R1 with the virtual register *acc* value. The value will be available in the register after 2 cycles.

5.4 Arithmetic and Logic Operations

5.4.1 Add

15	11	wb	9	5	4	0
0	0	1	0	0	w	R1 R1 R1 R1 R1
0	0	1	0	1	w	R1 R1 R1 R1 R1
						R2 R2 R2 R2 R2

Decode	Execution	Writeback
add R1,R2	$acc = *R1 + *R2$	
add.w R1,R2	$acc = *R1 + *R2$	R1 = *acc
add R1,K2	$acc = *R1 + K2$	
add.w R1,K2	$acc = *R1 + K2$	R1 = *acc

Table 5-5 Add

Addition results are available in virtual register *acc* for the next instruction or in the result register two cycles later.

5.4.2 Subtract

15	11	wb	9	5	4	0
0	0	1	1	0	w	R1 R1 R1 R1 R1
0	0	1	1	1	w	R1 R1 R1 R1 R1
						R2 R2 R2 R2 R2

Decode	Execution	Writeback
sub R1,R2	$acc = *R1 - *R2$	
sub.w R1,R2	$acc = *R1 - *R2$	R1 = *acc
sub R1,K2	$acc = *R1 - K2$	
sub.w R1,K2	$acc = *R1 - K2$	R1 = *acc

Table 5-6 Subtract

Subtraction results are available in virtual register *acc* for the next instruction or in the result register two cycles later.

5.4.3 Bitwise Logical AND

15	11	wb	9	5	4	0
0	1	1	0	0	w	R1 R1 R1 R1 R1 K2 K2 K2 K2 K2
0	1	1	0	1	w	R1 R1 R1 R1 R1 R2 R2 R2 R2 R2

Decode	Execution	Writeback
and R1,R2	$acc = *R1 \& *R2$	
and.w R1,R2	$acc = *R1 \& *R2$	$R1 = *acc$
and R1,K2	$acc = *R1 \& K2$	
and.w R1,K2	$acc = *R1 \& K2$	$R1 = *acc$

Table 5-7 Bitwise Logical AND

Contents of R1, R2 and constant K2 are treated as 16 bit boolean strings. Logical AND results are available in virtual register *acc* for the next instruction or in the result register two cycles later.

5.4.4 Bitwise Logical OR

15	11	wb	9	5	4	0
0	1	1	1	0	w	R1 R1 R1 R1 R1 K2 K2 K2 K2 K2
0	1	1	1	1	w	R1 R1 R1 R1 R1 R2 R2 R2 R2 R2

Decode	Execution	Writeback
or R1,R2	$acc = *R1 \mid *R2$	
or.w R1,R2	$acc = *R1 \mid *R2$	$R1 = *acc$
or R1,K2	$acc = *R1 \mid K2$	
or.w R1,K2	$acc = *R1 \mid K2$	$R1 = *acc$

Table 5-8 Bitwise Logical OR

Contents of R1, R2 and constant K2 are treated as 16 bit boolean strings. Logical OR results are available in virtual register *acc* for the next instruction or in the result register two cycles later.

5.4.5 Bitwise Logical Exclusive OR

15	11	wb	9	5	4	0
0	0	0	1	0	w	R1 R1 R1 R1 R1
0	0	0	1	1	w	R1 R1 R1 R1 R1
						R2 R2 R2 R2 R2

Decode	Execution	Writeback
xor R1,R2	$acc = *R1 \wedge *R2$	
xor.w R1,R2	$acc = *R1 \wedge *R2$	R1 = *acc
xor R1,K2	$acc = *R1 \wedge K2$	
xor.w R1,K2	$acc = *R1 \wedge K2$	R1 = *acc

Table 5-9 Bitwise Logical Exclusive OR

Contents of R1, R2 and constant K2 are treated as 16 bit boolean strings. Logical exclusive OR results are available in virtual register *acc* for the next instruction or in the result register two cycles later.

5.4.6 Optional Endian swap, Bitwise Shift Left

15	11	wb	9	5	4	0
0	1	0	1	0	w	R1 R1 R1 R1 R1
0	1	0	1	1	w	R1 R1 R1 R1 R1

Decode	Execution	Writeback
swapsl R1,R2	$acc = ((*R2 \& 0x10) ? Swap(*R1) : *R1) << (*R2 \& 0xf)$	
swapsl.w R1,R2	$acc = ((*R2 \& 0x10) ? Swap(*R1) : *R1) << (*R2 \& 0xf)$	R1 = *acc
swapsl R1,K2	$acc = ((K2 \& 0x10) ? Swap(*R1) : *R1) << (K2 \& 0xf)$	
swapsl.w R1,K2	$acc = ((K2 \& 0x10) ? Swap(*R1) : *R1) << (K2 \& 0xf)$	R1 = *acc

Table 5-10 Optionally Endian swap R1 before bit shift left

If Bit 4 of *R2 or K2 is set, *R1 are endian reversed. Bits 3:0 specify the shift left amount. Shift results are available in virtual register *acc* for the next instruction or in the result register two cycles later. Vacated bit positions are filled with a 0 valued bit.

5.4.7 Optional Endian swap, Bitwise Shift Right

15	11	wb	9	5	4	0
0	1	0	0	0	w	R1 R1 R1 R1 R1 K2 K2 K2 K2 K2
0	1	0	0	1	w	R1 R1 R1 R1 R1 R2 R2 R2 R2 R2

Decode	Execution	Writeback
swapsr R1,R2	$acc = ((*R2 \& 0x10) ? Swap(*R1) : *R1) \\ >> (*R2 \& 0xf)$	
swapsr.w R1,R2	$acc = ((*R2 \& 0x10) ? Swap(*R1) : *R1) \\ >> (*R2 \& 0xf)$	R1 = *acc
swapsr R1,K2	$acc = ((K2 \& 0x10) ? Swap(*R1) : *R1) \\ >> (K2 \& 0xf)$	
swapsr.w R1,K2	$acc = ((K2 \& 0x10) ? Swap(*R1) : *R1) \\ >> (K2 \& 0xf)$	R1 = *acc

Table 5-11 Optionally Endian swap R1 before bit shift left

If Bit 4 of *R2 or K2 is set, *R1 are endian reversed. Bits 3:0 specify the shift right amount. Shift results are available in virtual register *acc* for the next instruction or in the result register two cycles later. Vacated bit positions are filled with a 0 valued bit.

5.5 Coprocessor Memory Reference Operations

Coprocessor memory is addressed in 16-bit quantities from the VLx. Load targets and store destinations are specified in an offset from the beginning of coprocessor memory.

5.5.1 Load

15	11	wb	9	5	4	0
1	0	1	0	0	w	R1 R1 R1 R1 R1 R2 R2 R2 R2 R2

Decode	Execution	Writeback
ld R1,R2	load staging = CM[*R2]	
ld.w R1,R2	load staging = CM[*R2]	R1 ← *load staging

Table 5-12 Load from Coprocessor Memory

The loaded 16-bit value from coprocessor memory is available in the result register two cycles later. The ld without writeback only affects the state of the load staging area, CM, and as a side effect, this can cause the values of *run* and *lev* virtual registers to change, but nothing else useful happens. If the writeback bit is set, the value loaded is actually loaded into the R1 register.

5.5.2 Store

15	11	wb	9	5	4	0
1	0	1	0	1	w	R1 R1 R1 R1 R1 R2 R2 R2 R2 R2

Decode	Execution	Writeback
st R1,R2	CM[*R2] = *R1	

Table 5-13 Store to Coprocessor Memory

This instruction writes the value in the register specified by R1 into the memory location at the address specified by the value of R2.

5.6 Branch Operations

Branch operations cause a three cycle suspend in execution of the instruction stream (as if three nop operations were inserted; GB operations continue to progress). An exception to this is a fast-branch, which loops on instructions already in the instruction buffer.

5.6.1 Goto

15	11	10		0
1	0	0	0	1
	K11	K11	K11	K11

Decode	Execution	Writeback	
goto K11	issue 64 bit instruction fetch to &CM[K11 & ~0x3]		decode instruction at &CM[K11]

Table 5-14 Goto

This is an unconditional branch to the target address specified by K11.

5.6.2 Branch Based on virtual register acc value equal to 0

15	11	10		0
1	0	0	0	0
	K11	K11	K11	K11

Decode	Execution	Writeback	
br.acceq0 K11	if (acc == 0) issue 64 bit instruction fetch to &CM[K11 & ~0x3]		decode instruction at &CM[K11]

Table 5-15 Branch on acc value equal to 0

This is an conditional branch to the target address specified by K11 based on the state of the virtual register *acc*. If the *acc* value is equal to 0, the branch is taken.

5.6.3 Branch Based on virtual register *acc* value greater than 0

15	11	10					0
1	0	0	1	1	K11	K11 K11 K11 K11 K11	K11 K11 K11 K11 K11

Decode	Execution	Writeback	
br.accgt0 K11	if (<i>acc</i> > 0) issue 64 bit instruction fetch to &CM[K11 & ~0x3]		decode instruction at &CM[K11]

Table 5-16 Branch on *acc* value greater than 0

This is a conditional branch to the target address specified by K11 based on the state of the virtual register *acc*. If the *acc* value is greater than 0, the branch is taken. Note that the *acc* value is treated as a signed 16-bit value.

5.6.4 Complex Conditional Branch

15	11	wb	9	5	4	0
1	0	0	1	0	w	R1 R1 R1 R1 R1 K2 K2 K2 K2 K2

Decode	Execution	Writeback	
br R1,K2	if (K2 condition) { branch target = *R1; <i>acc</i> = *R1 & 0x3; issue 64 bit instruction fetch to &CM[<i>acc</i>] }		decode instruction at &CM[branch target]
call R1,K2	if (K2 condition) { branch target = *R1; <i>acc</i> = *R1 & 0x3; issue 64 bit instruction fetch to &CM[<i>acc</i>] }		decode instruction at &CM[branch target]

This is a conditional branch to the target address specified by *R1 based on any one of a number of conditions being true. The combinations can be specified by treating K2 as a mask. Any set of conditions within the same bank can be checked simultaneously. These are described in Table 5-17, "brai conditions," on page 38

K2 value	BANK [5:4] of K2	Bit for condition	Mnemonic	Condition Being tested
001--	00	2	gbpend	GB is in the midst of an I/O transaction
00-1-		1	dctmode	GB is in DCT coefficient inner loop processing
00--1		0	dtsbusy	DsContinue issued, but not yet sent to DS pending obtaining access to DTS
011--	01	2	gbstall	GB conditions do not allow bitshifting. The conditions that preclude this are 1) insufficient bits (<40) in the input buffer and GB I/O input is enabled, or 2) output is enabled, but there is insufficient free bit space (21 bits minimum)
01-1-		1	acceq0	acc == 0
01--1		0	acc4eq0	~(acc & 0xf)
101--	10	2	always	
10-1-		1	acc8eq0	~(acc & 0xff)
10--1		0	lsb0	CM load stage value is not zero
111--	11	2	gbstall	see above
11-1-		1	acclt0	acc < 0
11--1		0	accgt0	acc > 0

Table 5-17 brai conditions

Note that a branch if acc not equal to zero (accne0) can be achieved with a K2 value of '11011'b.

5.6.5 Fastbranch

15	11	wb	9	5	4	0
1	1	0	1	0	0	R1 R1 R1 R1 R1 0 0 1 0 1

Decode	Execution	Writeback
ioif R1, fastbranch	turn off instruction prefetching, set fastbranch target to loop offset	

Table 5-18 Fastbranch

A fastbranch is a one to eight instruction loop, where all the instructions reside in the instruction buffer, so there is no delay to return to the beginning of the loop, and there is no ongoing instruction prefetch. The loop is set up with the ioif instruction. Instructions in the code stream need to be aligned within an eight instruction block, with the last instruction of the loop at the end of the block. The ioif is used to set the VLE CPU into fastbranch mode, and to specify the starting location within the instruction buffer for the beginning of the loop. The starting location is specified as *R1. The starting location is precalculated, either by the assembler or by a programmer, according to the following equation:

$$(4 * InnerLoopOffset) + 2 \quad (EQ 3)$$

where *InnerLoopOffset* is a number corresponding to which instruction in the aligned eight instruction block is the first one of the loop. An *InnerLoopOffset* value of 0 indicates first instruction in the aligned eight instruction block.

NOTE:

Compiler support can hide this alignment issue. For example, a fastloop {instr ... instr} construct can be put in place, where the instructions embedded are aligned appropriately, and the ioif fastbranch is inserted in the instruction stream.

5.7 DS Operations

5.7.1 DsContinue

15	11	wb	9	5	4	0
1	1	0	1	0	0	R1 R1 R1 R1 R1 0 0 0 1 0

Decode	Execution	Writeback
ioif R1, dscontinue	send a DsContinue for DS channel specified by *R1	

Table 5-19 *DsContinue*

This instruction sends a DsContinue request to the DS for the channel identifier specified as *R1. There is no way for the VLx program to check that the channel is halted prior to issuing this operation. The *dtbusy* branch must be checked prior to issuing this operation since a DsContinue issued by the VLx while another DsContinue is in progress for the VLx will cause the channel identifier to be overwritten, and only one DsContinue to be issued to the DS.

5.8 GB Operations

15	11	wb	9	5	4	0
1	0	1	1	0	0	K1 K1 K1 K1 K1
1	0	1	1	1	0	K1 K1 K1 K1 K1
						R2 R2 R2 R2 R2

GetBits operations are issued from the VLE CPU to the GB. There are two instructions that are used to explicitly pass off an operation to GB: The *gb*. The K1 operand translates directly to a GB operation.

Decode	Execution	Writeback	
<i>gb</i> K1,K2	send the GB operation specified by K1 to the GB with K2 as a GB operand; GB determines <i>GBsign</i>	GB operation specific.	GB operation specific.
<i>gb</i> K1,R2	see above using *R2 in place of K2	see above	see above

Table 5-20 *gb* operation

The K2 or *R2 is the argument to the GB operation. For the MPEG2 table lookup functions, the K1 operand specifies that a hardware accelerated table lookup to one of these tables is being requested. The K2 or *R2 operand specifies which table is being used.

5.8.1 GB Bit Shifting operations

GB K1 mnemonic	K1 value
G_GETSYM	0b00011
G_REVSYM	0b00001
G_ADV2	0b10100

Decode	Exe/GB sign	Writeback/GB shift & count
gb G_GETSYM,K2 gb G_GETSYM,R2	send the GB operation specified by K1 to the GB with K2 or *R2 as a GB operand; GB determines <i>GBsign</i> ; <i>GBsign</i> affects <i>dctSign</i> value	GB shifts input buffer by length specified in K2 or *R2; new <i>nZero</i> is determined, <i>symbol</i> value modified if shift amount greater than 0; GB mode for <i>symbol</i> presents <i>symbol</i> value as in input buffer
gb G_REVSYM,K2 gb G_REVSYM,R2	send the GB operation specified by K1 to the GB with K2 or *R2 as a GB operand; GB determines <i>GBsign</i> ; <i>GBsign</i> affects <i>dctSign</i> value	GB shifts input buffer by length specified in K2 or *R2; new <i>nZero</i> is determined, <i>symbol</i> value modified if shift amount greater than 0; GB mode for <i>symbol</i> presents <i>symbol</i> value in reverse bit order until changed
gb G_ADV2,K2 gb G_ADV2,R2	send the GB operation specified by K1 to the GB with K2 or *R2 as a GB operand; GB determines <i>GBsign</i> ; <i>GBsign</i> affects <i>dctSign</i> value	GB shifts input buffer by length kept in internal register set by last GB hardware accelerated table lookup value; new <i>nZero</i> is determined, <i>symbol</i> value modified if shift amount greater than 0; GB mode for <i>symbol</i> presents <i>symbol</i> value in reverse bit order

The GB consumes bits in the input buffer under programmatic control. The pipeline for when the virtual registers are affected is described in Figure 4-3, "GB Bit shift operation pipeline," on

page 19.

The *symbol* value is the next 16 bits from the current position of the input buffer index. For example, the following table shows what value is presented according to the bits in the bitstream. The 1 and 0 bits represent the VLC 0b'00001' which is followed by a value of 's v0 v1 v2'. Shifting the VLC and its value out of the input stream would cause 10 bits to be consumed - the VLC, the value, and the sign bit.

15															0	bit position
x	x	x	x	x	x	s	v2	v1	v0	1	0	0	0	0	0	values in input stream
												*	*	*	*	nZero = 5
						*										GBsign = s
x	x	x	x	x	x	s	v2	v1	v0	1	0	0	0	0	0	G_GETSYM symbol
0	0	0	0	0	0	1	v0	v1	v2	s	x	x	x	x	x	G_REVSYM symbol

Table 5-21 Symbol example

5.8.2 GB Splicing operations

GB K1 mnemonic	K1 value
G_SPLICE	0b01111

Splicing operations only change the state of the output buffer. Vir-

Decode	Exe/GB	Writeback/GB splice
gb G_SPLICE,K2 gb G_SPLICE,R2	send the GB operation specified by K1 to the GB with K2 or *R2 as a GB operand - length	Splice in length into output buffer in most significant bit first order from <i>acc & lengthMask</i> where the lengthMask has one bit set for length specified

tual registers such as *dctSign*, *symbol* and *nZero* are unaffected. Bits are spliced into the output stream from most significant bit to least significant bit order using the *acc* value as the source bitstream.

5.8.3 GB Configuration operations

GB K1 mnemonic	K1 value
G_ALIGN	0b01000
G_SETOPT	0b01100
G_WRITE	0b11000

Configuration operations modify the state of the input buffer, the

Decode	Exe/GB	Writeback/GB config
gb G_ALIGN, K2 gb G_ALIGN, R2	send the GB operation specified by K1 to the GB with K2 or *R2 as a GB operand - ioBitMask	If (ioBitMask & 0x1), align input buffer position on byte alignment. This will cause a reread of previous bits if the input byte stream is not aligned and the <i>symbol</i> value is modified. If (ioBitMask & 0x2), align output buffer position on byte alignment. This will cause the output bitstream to be truncated to the nearest byte alignment. To not lose output bits, the programmer may choose to splice 7 0-valued bits into the output buffer before aligning the output buffer. This will function as a zero-pad.

gb G_SETOPT, K2 gb G_SETOPT, R2	send the GB operation specified by K1 to the GB with K2 or *R2 as a GB operand	Tell GB whether to count 0, 1 or both leading bits and whether I/O is enabled. The bit values for the operand are described in Figure 5.8.3.1, "GB setopt options," on page 46.
gb G_WRITE, K2 gb G_WRITE, R2	send the GB operation specified by K1 to the GB with K2 or *R2 as a GB operand - n	Swap GB I/O state. If n == 0, the current DS buffer information can be read; if n == 1, the DS buffer information will be written using the <i>acc</i> value resulting from execution of the following instruction. If n is anything other than 0 or 1, an I/O channel swap is in progress. In this case, a full sequence from n = 0 to n = 15 of G_WRITE is needed. Otherwise the GB I/O state is indeterminate. Note that changing the contents of the first 16 bits of the input buffer will change the <i>symbol</i> value. See "GB configuration options (G_write)" on page 47.

output buffer or the configuration of the GB. Note, that as the state of the input buffer is changed, the values seen through the *symbol*, *nZero*, and any values affected by *GBsign* may also be affected.

5.8.3.1 GB SETOPT OPTIONS

GB options are treated as a set of bit values. G_WRITE

Bit	Meaning
4	If set, forces GB into DCT processing mode
3	Specifies what to count for <i>nZero</i> as leading bits: 0 or 1. If clear, disregard bit 3 and count contiguous 0 or 1 bits depending upon the value of the first bit encountered. If
2	set, count only what is specified in bit 3.

Table 5-22 GB G_SETOPT bit settings

Bit	Meaning
1	If set, do not read any more input into the input buffer (input is disabled).
0	If set, generate output to the output buffer (enable output).

Table 5-22 GB G_SETOPT bit settings

For example, if a value of 0b'00101' is specified, both input and output are enabled, and only leading 0 bits will be counted.

5.8.3.2 GB CONFIGURATION OPTIONS (G_WRITE)

The GB configuration allows the I/O state to be read (if $n == 0$) or read/written (if $n > 0$). The n value determines what portion of the state is read/written. To determine what values of n affect what state, see Figure 4-7, "VLx GB I/O state," on page 23.

GB control bits specify the following.

Bit	Meaning
15	If set, do not generate output as input bitstream is shifted
14	If set, enable output
13	If clear, disregard bit 12 and count contiguous 0 or 1 bits depending upon the value of the first bit encountered. If set, count only what is specified in bit 12.
12	Specifies what to count for $nZero$ as leading bits: 0 or 1.
11:6	The DS Buffer ID for the buffer from which the input buffer will be populated.
5:0	The DS Buffer ID to which output data will be sent.

Table 5-23 GB Control Bits

The output buffer control state includes the following fields.

Bit	Meaning
15:10	Reserved
9:5	Specifies the current bit position index. New bits will be added at this position.

Table 5-24 GB output buffer control

Bit	Meaning
4:2	Reserved.
1	If set, output can be generated to the output buffer (output is enabled).
0	Specifies whether output is generated as input is consumed.

Table 5-24 *GB output buffer control*

The input buffer control is as follows.

Bit	Meaning
15:11	Reserved
10:5	Specifies the current bit position index. Bits will be consumed from this position.
4:0	Reserved.

Table 5-25 *GB input buffer control*

5.8.4 GB MPEG2 Hardware Table lookup operations

15	11	wb	9	5	4	0
1	0	1	1	0	0	1
0	1	0	0	0	0	0
K2	K2	K2	K2	K2	K2	K2
1	0	1	1	1	0	1
0	1	0	0	0	0	0
R2	R2	R2	R2	R2	R2	R2

These operations correspond to a lookup of a value in an MPEG2

GB K2 or *R2 mnemonic	GB K2 or *R2 value
G_CHR	0b00000
G_LUM	0b00001
G_MOT	0b00011
G_MBI	0b00101
G_MBP	0b00110
G_MBB	0b00111
G_MAI	0b01000
G_CBP	0b01001
G_NZPA	0b00100

table. The K1 value (G_HWACC) is always 0b10000. The K2 or *R2 indicates the MPEG2 table in which to do the lookup. For all the hardware encoded MPEG2 tables, an internal register is set with the length of the VLC plus the bit length of the value. The G_ADV2 GB operation must be used to cause the bitstream to advance over the VLC and its value to the next element of the bitstream for G_MOT

Table references are all to tables defined for MPEG2 in H.262.

Decode	Exe/GB	Writeback/GB hw table lookup	GB hw table lookup
gb G_HWACC, G_CHR	send the lookup request to the GB	Given the current <i>symbol</i> value, return as the <i>symbol</i> value the chrominance value in UIMSB (MSB in bit position 15) in bits 15:4 of <i>symbol</i> and the length from table B.13 as bits 4:0 of <i>symbol</i> .	
gb G_HWACC, G_LUM	send the lookup request to the GB	Given the current <i>symbol</i> value, return as the <i>symbol</i> value the luminance value in UIMSB (MSB in bit position 15) in bits 15:4 of <i>symbol</i> and the length from table B.13 as bits 4:0 of <i>symbol</i> .	
gb G_HWACC, G_MBI	send the lookup request to the GB		Given the current <i>symbol</i> value, look up the macroblock_type for I-pictures (table B.2). Return as the <i>symbol</i> value, the 6-bit type in bits 10:5 and the length of the symbol in bits 4:0.
gb G_HWACC, G_MBP	send the lookup request to the GB		Given the current <i>symbol</i> value, look up the macroblock_type for P-pictures (table B.3). Return as the <i>symbol</i> value, the 6-bit type in bits 10:5 and the length of the symbol in bits 4:0.

gb G_HWACC, G_MBB	send the lookup request to the GB		Given the current <i>symbol</i> value, look up the <i>macroblock_type</i> for I-pictures (table B.4). Return as the <i>symbol</i> value, the 6-bit type in bits 10:5 and the length of the symbol in bits 4:0.
gb G_HWACC, G_MAI	send the lookup request to the GB		Given the current <i>symbol</i> value, look up the <i>macroblock_address_increment</i> (table B.1). Return as the <i>symbol</i> value, the 6-bit value in bits 10:5 and the length of the symbol in bits 4:0.
gb G_HWACC, G_CBP	send the lookup request to the GB		Given the current <i>symbol</i> value, look up the <i>coded_block_pattern</i> (table B.9). Return as the <i>symbol</i> value, the 6-bit value in bits 10:5 and the length of the symbol in bits 4:0.
gb G_HWACC, G_NZPA	send the lookup request to the GB		Return as the <i>symbol</i> value (<i>symbol</i> >> (<i>nZero</i> +1)) & 0x1f

Note that the G_MOT is a multi-stage GB operation, requiring

Decode	Exe/GB 1	WB/GB 1	GB 1	Decode/GB 2	Exe/GB 2	GB 2	GB2
gb G_HWACC, G_MOT	send the lookup request to the GB	Given the current <i>symbol</i> value, lookup the motion code value		acc virtual register must remain unchanged; use this value to determine the number of bits that make up the motion residual in the bitstream			return as the <i>symbol</i> value, the motion residual in UIMSB (MSB in bit position 8) in bits 15:8, and the 8 bit signed motion code value in bits 7:0.

twice the number of GB cycles to perform.

5.8.5 GB DCT operations

15	11	wb	9	5	4	0								
1	0	1	1	0	0	0	0	0	0	K2	K2	K2	K2	K2
1	0	1	1	1	0	0	0	0	0	0	R2	R2	R2	R2

The GB is explicitly forced into DCT processing mode. The value of

Decode	Exe/GB	Writeback/GB shift and Count	GB DCT
gb G_DCT, K2 gb G_DCT, R2	send the GB operation specified by K1 to the GB with K2 or *R2 as a GB operand; GB determines GBsign; GBsign affects dctSign value	GB shifts input buffer by length specified in K2 or *R2; new nZero is determined, symbol value modified if shift amount greater than 0; GB mode for symbol presents symbol value as in input buffer	Using the nZero, read the corresponding r[nZero] register and calculate the table index value (See "DCT Processing" on page 24). Yields an updated symbol value.

the K1 operand is '0b0000'. The GB can also be put into DCT mode via the G_SETOPT. And the brai.dct will also put the GB into DCT mode. See Section 4.2.4.5, *DCT Processing*, on page 24, and Section 6.4, *DCT Table Setup*, on page 60 for more information.

5.8.6 VLx PIO Register Bits List

The following table lists the VLx PIO register bit uses. There is one

Register Field Mnemonic	Bits	Description
NormalRFDDataPort	31:16	RF data. Always updated with the values present on port 2 (R2) when read in this mode. There is no effect when it is written in this mode.
NormalRAMAddr	15:7	RAM address. Current location used by the VLx.
NormalGBPending	6	GB has outstanding request to IOBUS
NormalSetRFAddress	5	Must be 0 for this mode
NormalGBCool	4	Cool down the GB. Stops the GB from making any input data requests on the IOBUS
NormalBreak	3	Debug break. Must be 0 for this mode.
NormalReset	2	Causes VLx PC to be reset to 0 offset from CM1 bank.
NormalStep	1	Single cycle execution of the VLx CPU and GB
NormalRun	0	Turns on the VLx and GB clock
RFReadRFDDataPort1	31:16	RF data. Always updated with the values present on port 1 (R1) when read in this mode. There is no effect when it is written in this mode.
RFReadRFAddr	15:11	Identifies which VLx general register to read
RFReadWriteRF	10	Must be 0 for this mode
---unused---	9:7	
RFReadGBPending	6	Same as NormalGBPending
RFReadSetRFAddr	5	Must be 1 for this mode
RFReadGBCool	4	Same as NormalGBCool
RFReadBreak	3	Debug break. Must be 0 for this mode.

Table 5-26 PIO Register Bits of the VLx

Register Field Mnemonic	Bits	Description
RFRReadReset	2	Same as NormalReset
RFRReadStep	1	Same as NormalStep
RFRReadRun	0	Same as NormalRun
RFWWriteRFDataPort1	31:16	R1 port is set to this value when the register is written in this mode. Its value is undefined when read in this mode
RFWWriteRFAddr	15:11	Identifies which VLx general register to write
RFWWriteWriteRF	10	Must be 1 for this mode
--unused--	9:7	
RFWWriteGBPending	6	Same as NormalGBPending
RFWWriteSetRFAddr	5	Must be 1 for this mode. Forces an address onto R1 port.
RFWWriteGBCool	4	Same as NormalGBCool
RFWWriteBreak	3	Debug break. Must be 0 for this mode
RFWWriteReset	2	Same as NormalReset
RFWWriteStep	1	Same as NormalStep
RFWWriteRun	0	Same as NormalRun
--unused--	31:16	
SetBkptRAMAddrBkpt	15:7	RAM break point address in 4K RAM
SetBkptGBPending	6	Same as NormalGBPending
SetBkptSetRFAddr	5	Must be 0 for this mode
SetBkptGBCool	4	Same as NormalGBCool
SetBkptBreak	3	Debug break. Must be 1 for this mode
SetBkptReset	2	Same as NormalReset
SetBkptStep	1	Same as NormalStep
SetBkptRun	0	Same as NormalRun

Table 5-26 PIO Register Bits of the VLx
 register, defined in the MAP1000 PIO memory map. Depending on the mode of this register, the bits take on different values. There are four modes:

- Normal Mode: Break is 0 and NormalSetRFAddr is 0
- RF Read Mode: Break is 0, RFReadSetRFAddr is 1, and RFReadWriteRF is 0
- RF Write Mode: Break is 0, RFWriteRFAddr is 1, and RFWriteWriteRF is 1
- Breakpoint Mode: SetBkptBreak is 1, SetBkptRFAddr is 0

This table does not replace the MAP1000 PIO memory map, but is merely a convenience. The MAP1000 PIO memory map is definitive.

5.9 PIO Register Bit descriptions

5.9.1 Run

The Run bit (NormalRun, RFReadRun, RFWriteRun) indicates the status of the VLx CPU and GB clock. If the bit is set, the clock is turned on. If the bit is clear, the clock is turned off.

5.9.2 GBCool

The GBCool bit (NormalGBCool, RFReadGBCool, RFWriteGBCool) indicates whether the GB input is enabled. When the bit is set, input is disabled. When the bit is clear, input is enabled.

5.9.3 GBPending

The GBPending bit (NormalGBPending, RFReadGBPending, RFWriteGBPending) indicates whether or not there is an I/O transaction in progress for either GB input or output. This is a read only bit. If the bit is clear, no I/O transaction is in progress.

6. Programmer's Notes

6.1 Initialization

Prior to enabling the VLx CPU and GB clock,

- the PC for the CPU should be reset to 0 (a known state),
- the VLx program should be loaded into CM, and
- the CM1 should be enabled for access by the VLx.

When the CPU clock is enabled, the VLx will begin instruction execution. The GB will execute when GB operations are passed on to it from the CPU.

Prior to loading the VLx program into CM, the CM1 bank in the FFU must be enabled for VLx to allow access to the CM bank from the DTS. This requires setting the FFU into vld_mode via the unit bits in the 3d2dControlRegister.

The DsContinue issued from the VLx requires setup prior to use. The VLx supplies the VldDsContinue with the six bits (17:12) that identify the DS channel to be DsContinue'd. But the VLIW program is responsible for initializing the remaining bits of the VldDsContinue register, such that the write of the six bits from the VLx causes a PIO to be issued to the DS.

6.2 Shutdown/state save for context switch

Graceful shutdown or context switch of the VLx is initiated under the control of the program executing on the VLIW core, but requires cooperation from the currently executing VLx program as well. The sequence to do this would be as follows:

- VLIW program communicates to VLx program in an application specific manner that a shutdown/context switch is requested. One mechanism for this would be a write to an agreed upon *command* location in CM by the VLIW program which the VLx program would check at well-defined times, like when in a GBstall condition.

- VLIW program or VLx program disables GB I/O input which would eventually cause a GB stall if the VLx program is processing input.
- VLx program could flush GB output by splicing with 0 bits.
- VLx program disables I/O output.
- VLx program saves GB I/O channel state to CM.
- VLx program saves any pertinent program state (such as register values) to CM. The VLx program may be written in such a way that the first instructions executed after reset check to see if this is a fresh start or resumption by checking appropriate state and jumping to appropriate program location based upon state.
- VLx program signals VLIW program via application specific manner that state save has been completed.
- VLIW program stops the VLx CPU clock.
- VLIW program initiates transfer of appropriate CM contents to a safe location prior to releasing CM for use by another VLx or FFU program.

Abrupt shutdown of the VLx can be achieved by the following sequence:

- VLIW program disables GB I/O input requests,
- VLIW program some number of cycles for the VLx program to reach a stall state
- VLIW program turns off the VLx CPU clock.

Note that there is no inherent support for resuming VLx program execution following VLx preemption.

Also, note by turning off the VLx CPU clock, the VLx state is preserved. Thus, if the only resource that needed to be switched out for another use was the CM (as for running 3D TAC processing), this could be achieved by simply saving and restoring the CM state after doing an abrupt shutdown of the VLX as described above, and then re-enabling GB I/O input and turning the VLx CPU clock back on.

6.3 Channel swapping

Channel swapping to handle multiplexing of bitstreams is something that requires handling by the VLx program. There is no mechanism to do this entirely from the VLIW program without VLx program cooperation.

Channel swapping requires

- disabling GB I/O -- both input and output,
- swapping channel state (See "GetBits Modes" on page 21), and
- re-enabling I/O.

Channel swapping must be handled in its entirety. A partial swap will leave the GB I/O in an indeterminate state. One possible instruction sequence to perform this is shown in Table 6-1. Note that in this

```

register tmp3 r30;
register gbind r24;
L_GBSWAP:                                # save channel state by
setacc $;                                # making sure we are not in a GBstall or
br.gbstall acc;                           # GBpending state, and then switching
setacc $;                                # channel state in its entirety
br.gbpend acc;                            # NOTE: gbind is used to contain the data
setreg gbind, L_GBDATA000;                # location where other GB state resides
nop;
ld.w tmp3, gbind;                         # set up tmp3 with state
fastloop {                               # set up fast branch; gbind low 5 bits = 0
    gb G_WRITE, gbind;                   # Write value in tmp3 to state gbind
    add tmp3, 0;                          # tmp3 has state value to write
    add tmp3, 0;
    nop;                                  # note that symbol valid next cycle
    st symbol, gbind;                     # save symbol to CM at gbind location
    add gbind, 1;                          # increment to next state for swap
    br.acc4eq0 pcsave;                    # return to caller if gbind low 4 bits = 0
}

```

Table 6-1 GB channel swap example

```

        ld.w tmp3, gblnd;          # setup for next state to write into GB
    }

                                GBDATA000 needs to be aligned for G_write
                                which is looking at low 4 bits of address as
                                indicator of what to write/read

    align 16
    L_GBDATA000:
    data 0b 0000 0000 0000 0000    # N=0
    data 0b 0 00 00010 000011      # N=1, new in chan: 2, new out chan: 3
                                    # N=2, generate output as input consumed;
    data 0b 0000 0000 0000 0011    output enabled; no output in output buffer
    data 0 (12)                     # N=3-14 in/out data
    data 0b 0000 0 110000 100 00    # N=15, initial sequence to force input fill

```

Table 6-1 GB channel swap example

example, L_GBDATA000 is at a 16 byte aligned address. The G_write will use the lower 4 bits of the address to determine which values to write/read in this example.

6.4 DCT Table Setup

To use the GB for DCT processing, a table must be constructed as a series of lookup tables that reside in CM as data. The tbase registers must be initialized to identify the size and location of subtables that comprise the table. The tbase register value is described in Section 4.2.4.5, *DCT Processing*, on page 24. The tkeep indicates the size of the table, where the number of entries in the table is $2^{tkeep - tskip}$. The entries are organized by the values of the remaining bits in the VLC in bit reversed order, and there must not be any skipped entries. For example, if tkeep specified a value of 3 and tskip specifies a value of 1 for tbase1 where leading 0s are being counted, the sub-table will have four entries handling VLC '0100' (entry 0; MSB left most bit), VLC '0110' (entry 1), VLC '0101' (entry 2), and VLC '0111' (entry 3). The processing of the stream below for VLC '0101' (note that the oldest bit -- the MSB -- is at the right)

'xxxxxxxxxxxx1010'

will cause the GB to shift by two bits (tkeep-tskip), resulting in

'xxxxxxxxxxxx10'

and then mask these bits with the *tkeepMask* (see "DCT Processing" on page 24) causing entry '10', or 2, to be used in the subtable.

Note that if any of these entries' values were illegal or required special processing, specifying a length of zero in the table entry will terminate a dct loop. Also note that if VLCs '0110' and '0111' were really the same VLC (e.g., '011s'), the length could be specified as '4', and the last bit is handled as the sign bit.

The table entry contains run and level values as documented in MPEG2 B.14 and B.15. The table entry also contains a length field, which is the length of the VLC. The run and level values will become the virtual register *run* and *lev* values, respectively if the table entry is read from CM with a load operation.

15:11	10:5	4:0
run	level	length

The length is used to implicitly shift the bitstream by the length of the VLC matched. If a length of zero is specified in the table, the call.dct will branch out of the loop that is processing DCTs.

The resulting table structure and state set is shown in Figure 6-1. on page 61.

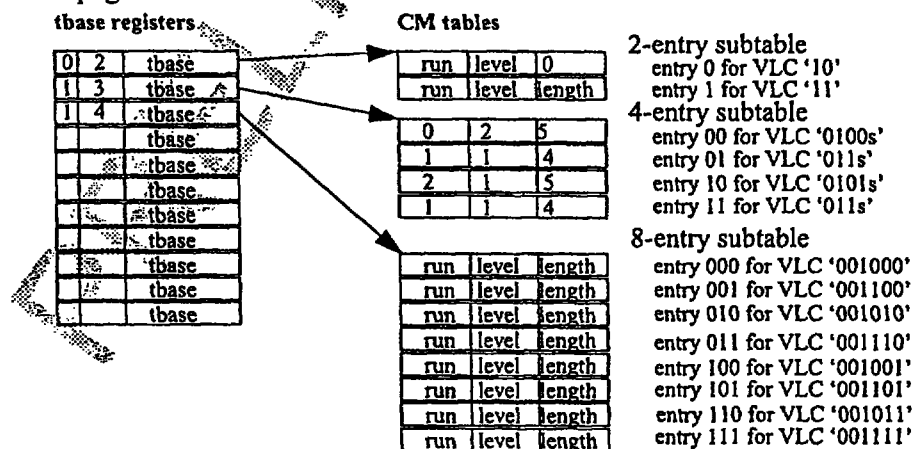


Figure 6-1. DCT Tbase register and table setup

There are two ways to put GB into DCT processing mode. The first is to explicitly set the mode:

`gb_setopt(0b1xxxx); # setting bit 4.`

This will force the symbol value to take on the DCT CM result for the current VLC.

Another way is to use the

`gb_dct(0);`

which will force the operation to be handled in DCT processing mode.

In addition, the call `dct(r)` is a special instruction that actually causes several operations to happen in parallel, in both the GB, the CPU, and for CM access.

- Using the current *symbol* value, a *ld.w lev, symbol* CM read operation is initiated. Note that this reads the CM specified by *symbol* which should already be set up to read a table entry value.
- If the length value from the table entry is 0, or if a *gbstall* condition is true, branch to the register specified.
- Otherwise, a special GB operation is initiated that is analogous to a `gb_dct(length-returned-from-last-CM-read)` is issued to GB.

7. Designer's Comments

- The GB, CPU, and CM interaction has been designed to support a five cycle DCT processing loop. The code for this needs to conform very closely to the example code to satisfy latency and conflict issues. Similar notes apply to the motion vector, luminance and other special accelerated processing.

PRELIMINARY



1. A variable-length encode/decode processor, comprising:
a central processing unit;
5 an instruction buffer coupled to the central processing unit; and
a getbits processing engine coupled to the central processing unit.
2. The processor of claim 1, further comprising a register file coupled to
the central processing unit and to the getbits processing engine.
10
3. The processor of claim 1, further comprising an input buffer coupled to
the getbits processing engine.
4. The processor of claim 1, further comprising an output buffer coupled
15 to the getbits processing engine.
5. The processor of claim 1 wherein the central processing unit includes a
virtual accumulator register.
- 20 6. The processor of claim 1, further comprising:
an input buffer coupled to the getbits processing engine; and
wherein the input buffer comprises a virtual symbol register.
- 25 7. A function processing block, comprising,
an instruction buffer;
a memory;
a getbits processing engine; and
a central processing unit coupled to the instruction buffer, the memory, and
the getbits processing engine.
30



8. The function processing block of claim 7 wherein the memory is coupled to the instruction buffer.

5 9. The function processing block of claim 7 wherein the memory is coupled to the getbits processing engine.

10 10. The function processing block of claim 7, further comprising a register file coupled to the memory, the getbits processing engine, and the central processing unit.

11. The function processing block of claim 7, further comprising:
an input buffer coupled to the getbits processing engine;
an output buffer coupled to the getbits processing engine; and
an input/output bus coupled to the input and output buffers.

15 12. A processor, comprising:
a core;
a controller coupled to the core; and
a function processing block coupled to the controller, the function processing
20 block including,

a memory,
an input/output bus, and
a variable-length encode/decode processor coupled to the memory
and to the input/output bus, the variable-length encode/decode processor
25 including,
a getbits processing engine coupled to the input/output bus, and
a central processing unit coupled to the getbits processing
engine.



13. A method, comprising:
processing data with a getbits engine of a variable-length encode/decode processor; and
controlling the getbits engine with a central processing unit of the variable-length encode/decode processor.
14. The method of claim 13, further comprising executing instructions stored in an instruction buffer with the central processing unit.
15. The method of claim 13, further comprising:
receiving data from an input/output bus with the getbits engine; and
providing data to the input/output bus with the getbits engine.
16. The method of claim 13, further comprising controlling the variable-length encode/decode processor with a core processor.
17. The method of claim 13, further comprising:
analyzing a variable-length symbol with the getbits engine; and
accessing a look-up table based on the value of the variable-length symbol.
18. The method of claim 13, further comprising:
analyzing a variable-length symbol with the getbits engine; and
accessing a look-up table based on the number of leading zeroes in the variable-length symbol.
19. The method of claim 13, further comprising:
analyzing a variable-length symbol with the getbits engine; and
accessing a look-up table based on the number of leading ones in the variable-length symbol.