Contents

1 Godson-2E Micro Architecture 1
1.1 Godson Series Processors 1
1.2 Godson-2E Micro Architecture Overview 1
1.3 Fetching and Decoding 4
1.4 Register Renaming 5
1.5 Issuing and Reading operands 6
1.6 Execution and Functional Units 8
1.7 Commit and Reorder Queue 9
1.8 Branch canceling and Branch Queue 10
1.9 Memory Subsystem 11
2 Instruction Set Overview 15
3 Memory Management 21
3.1 Translation Lookaside Buffer 21
3.1.1 Joint TLB 21
3.1.2 Instruction TLB 22
3.1.3 Hits and Misses 22
3.1.4 Multiple Matches 23
3.2 Processor modes 23
3.2.1 Processor Operating Modes 23
3.2.2 Addressing mode 24
3.2.3 Instruction set mode 24
3.2.4 Endian mode 24
3.2.5 Address Spaces 24
3.2.6 Virtual Address Space 24
3.2.7 Physical Address Space 24
3.2.8 Virtual-to-Physical Address Translation 24
3.2.9 User Address Space 26
3.2.10 Supervisor Space 27
3.2.11 Kernel Space 29
3.3 System Control Coprocessor 31
3.3.1 Format of a TLB Entry 31
3.3.2 CP0 Registers 33
3.3.3 Virtual-to-Physical Address Translation Process 33
3.3.4 TLB Exceptions 34
3.3.5 TLB Instructions 34
3.3.6 Code examples 35
4 Cache Organization and Operation 37

4.1 Cache Overview 37
4.1.1 Non-Blocking Caches.. 37
4.1.2 Replacement Algorithm... 38
4.1.3 Cache Attributes 38
4.2 Primary Instruction Cache... 39
4.2.1 Instruction Cache Organization... 39
4.2.2 Accessing Instruction Cache.. 40
4.3 Primary Data Cache 40
4.3.1 Data Cache Organization.. 41
4.3.2 Accessing the Data Cache 42
4.3.3 Processing Data Cache Miss 42
4.4 Secondary Cache.. 43
4.4.1 Secondary Cache Organization 43
4.4.2 Accessing the Secondary Cache 43
4.5 Cache Coherency 44
4.5.1 Cache Coherency Attributes 44
4.5.2 Uncached, Blocking (Coherency Code 2) 45
4.5.3 Writeback (Coherency code 3) 46
4.5.4 Uncached Accelerated (Coherency Code 7).. 46
4.6 Cache Maintenance 46
5 CP0... 49
5.1 Index Register (0).. 50
5.2 Random Register (1) 51
5.3 EntryLo0 (2), and EntryLo1 (3) Registers 51
5.4 Context (4) 53
5.5 PageMask Register(5) 54
5.6 Wired Register (6).. 55
5.7 BadVAddr Register (8)... 55
5.8 Count and Compare Registers(9 and 11) 56
5.9 EntryHi Register (10)... 57
5.10 Status Register (12) 57
5.11 Cause Register(13) 60
5.12 Exception Program Counter (14) 61
5.13 Processor Revision Identifier (PRID) Register 62
5.14 Config Register (16).. 63
5.15 Load Linked Address (LLAddr) Register (17)... 64
5.16 Watch Register 64
5.17 Xcontext Register(20) 64

5.18 Diagnostic Register(22) 65
5.19 Performance Counter Registers (24,25) 66
5.20 TagLo (28) and TagHi (29) Registers 68
5.21 ErrorEPC Register(30) 69
5.22 CP0 Instructions 69
6 CPU Exceptions 71
6.1 Causing and Returning from an Exceptions 71
6.2 Exception Vector Locations 71
6.3 TLB Refill Vector Selection 72
6.4 Priority of Exceptions 72
6.5 Cold Reset Exception 73
6.6 Soft Reset Exception 74
6.7 NMI Exception 76
6.8 Address Error Exception 77
6.9 TLB Exceptions 77
6.10 TLB Refill Exceptions 78
6.11 TLB Invalid Exception 79
6.12 TLB Modified Exception 80
6.13 Bus Error Exception 81
6.14 Integer Overflow Exception 82
6.15 Trap Exception 82
6.16 System Call Exception 83
6.17 Breakpoint Exception 84
6.18 Reserved Instruction Exception 85
6.19 Coprocessor Unusable Exception 85
6.20 Floating-Point Exception 86
6.21 Watch Exception 87
6.22 Interrupt Exception 88
7 Floating-Point Unit 91
7.1 Overview 91
7.2 FPU Programming Model 92
7.2.1 Floating-Point Registers 92
7.2.2 Floating-Point Control Registers 93
7.3 FPU Instruction Set Overview 96
7.4 FPU Formats 99
7.4.1 Floating-Point Format 99
7.4.2 Multimedia Format 100
7.5 FPU Instruction Pipeline Overview 101

7.6 FPU Exceptions 102
8 Privileged Instruction 109
8.1 CP0 Move Instructions 109
8.1.1 DMFC0 Instruction 109
8.1.2 DMTC0 Instruction 110
8.1.3 MFC0 Instruction 110
8.1.4 MTC0 Instruction 111
8.1.5 Usable CP0 Move Instruction in User Mode 111
8.2 TLB Access Instructions 111
8.2.1 TLBP Instruction 111
8.2.2 TLBR Instruction 112
8.2.3 TLBWI Instruction 113
8.2.4 TLBWR Instruction 113
8.3 ERET Instruction 114
8.4 CACHE Instruction 115
8.4.1 Index Invalidate (I) 116
8.4.2 Index WriteBack Invalidate (D) 116
8.4.3 Index WriteBack Invalidate (S) 116
8.4.4 Index Load Tag (D) 117
8.4.5 Index Load Tag (S) 117
8.4.6 Index Store Tag (D) 117
8.4.7 Index Store Tag (S) 118
8.4.8 Hit Invalidate (D) 118
8.4.9 Hit Invalidate (S) 118
8.4.10 Hit WriteBack Invalidate (D) 119
8.4.11 Hit WriteBack Invalidate (S) 119
8.4.12 Index Load Data (D) 119
8.4.13 Index Load Data (S) 120
8.4.14 Index Store Data (D) 120
8.4.15 Index Store Data (S) 120
9 DDR SDRAM Control Interface 121
9.1 DDR SDRAM Controller Functional Overview 121
9.2 DDR SDRAM Read Protocol 122
9.3 DDR SDRAM Write Protocol 122
9.4 DDR SDRAM Configuration 123
9.5 DDR SDRAM Sampling Mode Configuration 125
10 Performance Tuning 127
10.1 User instruction Latency and Repeat Rate 127

10.2 Instruction extensions........128
10.3 Instruction Stream........129
10.3.1 Instruction alignment........129
10.3.2 Branch handling........129
10.3.3 Improving Instruction Stream density........131
10.3.4 Instruction scheduling........131
10.4 Memory accesses........132
10.5 Other Tips........132
AppendixA Godson new integer instructions........133
1. MULT.G — Multiply Word (Godson2)........133
2. MULTU.G — Multiply Unsigned Word (Godson2)........134
3. DMULT.G — DoubleWord Multiply (Godson2)........135
4. DMULTU.G—Doubleword Multiply Unsigned (Godson2)........136
5. DIV.G —Divide Word (Godson2)........137
6. DIVU.G — Divide Unsigned Word (Godson2)........138
7. DDIV.G — Doubleword Divide (Godson2)........139
8. DDIVU.G — Doubleword Divide Unsigned(Godson2)........140
9. MOD.G —MOD Word (Godson2)........141
10. MODU.G — Mod Unsigned Word (Godson2)........142
11. DMOD.G — Doubleword Mod (Godson2)........143
12. DMODU.G — Doubleword Mod Unsigned(Godson2)........144
AppendixB Godson new float-point instructions........145
1. MADD.fmt— Floating-Point Multiply Add........145
2. MSUB.fmt— Floating-Point Multiply Subtract........146
3. NMADD.fmt— Floating-Point Negative Multiply Add........147
4. NMSUB.fmt— Floating-Point Negative Multiply Subtract........148

Figure 1-1 Microarchitecture of Godson-2E..........13
Figure 2-1 CPU Instruction Formats..........15
Figure 3-1 Overview of a Virtual-to-Physical Address Translation..........25
Figure 3-2 64-bit Mode Virtual Address Translation..........26
Figure 3-3 User Virtual Address Space as viewed from User Mode..........27
Figure 3-4 User and Supervisor Address Spaces; viewed from Supervisor mode..........28
Figure 3-5 User, Supervisor, and Kernel Address Space viewed from Kernel mode..........29
Figure 3-6 Format of a TLB Entry..........31
Figure 3-7 EntryHi Register Format..........32
Figure 3-8 PageMask Register Format..........32
Figure 3-9 EntryLo0 and EntryLo1 Register Formats..........32
Figure 3-10 TLB Address Translation..........34
Figure 4-1 Instruction Cache Organization..........39
Figure 4-2 Instruction Cache Line Format..........39
Figure 4-3 Accessing the Instruction Cache..........40
Figure 4-4 Data Cache Organization..........41
Figure 4-5 Data Cache Line Format..........41
Figure 4-6 Accessing the Data Cache..........42
Figure 4-7 Accessing the Secondary Cache..........44
Figure 5-1 Index Register..........50
Figure 5-2 Random Register..........51
Figure 5-3 Fields of the EntryLo0 and EntryLo1 Registers..........52
Figure 5-4 Context Register Format..........53
Figure 5-5 PageMask Register..........54
Figure 5-6 Wired Register Boundary..........55
Figure 5-7 Wired Register..........55
Figure 5-8 BadVAddr Register Format..........56
Figure 5-9 Count and Compare Registers..........56
Figure 5-10 EntryHi Register..........57
Figure 5-11 Status Register..........58
Figure 5-12 Cause Register Format..........60
Figure 5-13 EPC Register Format..........62
Figure 5-14 Processor Revision Identifier Register Format..........62
Figure 5-15 Config Register Format..........63

Figure 5-16 Watch Register Formats 64
Figure 5-17 XContext Register Format 65
Figure 5-18 Diagnostic Register 66
Figure 5-19 Performance Counter Registers Format 66
Figure 5-20 TagLo and TagHi Register (P-cache) Formats 69
Figure 5-21 ErrorEPC Register Format 69
Figure 7-1 The organization of the functional units in Godson-2E's architecture 92
Figure 7-2 FP Control/Status Register Bit Assignments 94
Figure 7-3 Floating-Point Format 99
Figure 7-4 packed unsigned half-word format 101
Figure 7-5 packed signed half-word format 101
Figure 9-1 DDR SDRAM read protocol 122
Figure 9-2 DDR SDRAM write protocol 123
Figure 9-3 DDR SDRAM sampling mode when memory to core ratio is 1:10 125

Table 2-1 CPU Instruction Set: Load and Store Instructions......17
Table 2-2 CPU Instruction Set: Arithmetic Instructions (ALU Immediate)......17
Table 2-3 CPU Instruction Set: Arithmetic (3-Operand, R-Type)......18
Table 2-4 CPU Instruction Set: Multiply and Divide Instructions......18
Table 2-5 CPU Instruction Set: Jump and Branch Instructions......19
Table 2-6 CPU Instruction Set: Shift Instructions......19
Table 2-7 CPU Instruction Set: Special Instructions......20
Table 2-8 CPU Instruction Set: Exception Instructions......20
Table 2-9 CP0 Instructions......20
Table 3-1 Processor operating modes......23
Table 3-2 TLB Page Coherency (C) Bit Values......33
Table 3-3 Memory Management-Related CP0 Registers......33
Table 3-4 TLB Instructions......35
Table 4-1 Cache attributes......38
Table 4-2 Godson-2E Cache Coherency Attribute......45
Table 5-1 Coprocessor 0 Registers......49
Table 5-2 Fields in the Index Register......50
Table 5-3 Fields in the Random Register......51
Table 5-4 Description of EntryLo Registers' Fields......52
Table 5-5 Context Register Fields......53
Table 5-6 Mask Field Values for Page Sizes......54
Table 5-7 Fields in the Wired Register Field......55
Table 5-8 EntryHi Register Fields......57
Table 5-9 Fields in the Status Register......58
Table 5-10 Fields in the Cause Register......60
Table 5-11 Cause Register ExcCode Field......61
Table 5-12 PRId Register Fields......62
Table 5-13 Fields in the Config Register......63
Table 5-14 WatchHi and WatchLo Register Fields......64
Table 5-15 Fields in the XContext Register......65
Table 5-16 Diagnostic Register Fields......66
Table 5-17 Control Fields Format......67
Table 5-18 Count Enable Bit Definition......67
Table 5-19 Counter 0 Events......67

Table 5-20 Counter 1 Events........68
Table 5-21 Cache Tag Register Fields........69
Table 5-22 CP0 Instructions........69
Table 6-1 Exception Vector Addresses........71
Table 6-2 Exception Priority Order........73
Table 7-1 FCR0 Fields........93
Table 7-2 Control/Status Register Fields........94
Table 7-3 Rounding Mode Bit Decoding........96
Table 7-4 floating-point instructions in Godson-2E FPU........97
Table 7-5 Paired-single (PS) instructions in Godson-2E FPU........98
Table 7-6 Equations for Calculating Values in Single and Double-Precision Floating-Point Format........100
Table 7-7 Floating-Point Format Parameter Values........100
Table 7-8 Minimum and Maximum Floating-Point Values........100
Table 7-9 Default FPU Exception Actions........104
Table 8-1 Godson-2E Privileged Instructions........109
Table 8-2 CP0 Move Instructions........109
Table 8-3 CACHE Instruction Op Field Encoding........115
Table 9-1 DDR SDRAM chip supported........121
Table 9-2 DDR SDRAM configuration register........123
Table 9-3 Sample point configuration register........126
Table 10-1 Latencies and Repeat Rates for User Instructions........127

1 Godson-2E Micro Architecture

1.1 Godson Series Processors

The Godson processors include three series. The Godson-1 series of processors and IPs mainly focus 32-bit embedded applications, the Godson-2 series of processors mainly focus on 64-bit high-end embedded and desktop applications, while the multiple-issue Godson-3 processors focus on server and high performance computing applications.

Multiple levels of parallelism can be explored to improve performance of a processor. In instruction level, out-of-order execution and superscalar technique allow the processor to schedule the execution of instructions in a maximum throughput. In data level, vector instructions that are implemented with SIMD technique enable the processor to generate multiple results with one instruction. In thread level, multithreading enables multiple threads to run simultaneously on a single or multiple processors.

The Godson-1 processor implements single-issue out-of-order execution pipeline, with static branch prediction and blocking cache; the Godson-2 processor implements superscalar out-of-order execution pipeline, with dynamic branch prediction and non-blocking cache, it also implements some fix-point SIMD instructions by reusing the floating-point datapaths; and the Godson-3 processor will further implement multiple core technology.

1.2 Godson-2E Micro Architecture Overview

Godson-2E is an enhanced version of the previous Godson-2[1] which is a four-issue general-purpose RISC microprocessor that implements the 64-bit MIPS instruction set. Its main architectural improvement over the previous Godson-2 includes increasing the number of entries of reorder buffer (from 32 to 64) and memory queue (from 16 to 24) to reduce pipeline stall, upgrading two floating point units (one for addition/subtraction and one for multiplication) to two MAC (multiply and accumulation) floating point units, enhancing the memory performance with a on-chip 512KB L2 cache and an on-chip memory controller, and some other improvements such as speculative forwarding, prefetching and store fill buffer

optimization[] to reduce memory access latency and memory bandwidth requirements. Besides, Godson-2E is physically implemented and fabricated with the STmicro 90nm technology to reach the main frequency over 1GHz.

The four-way superscalar of Godson-2E raises extremely high requirements for inter-instruction dependency resolving and instruction/data providing. Godson-2E employs out-of-order execution and aggressive memory hierarchy design to improve pipeline efficiency.

Out-of-order execution is a combination of the register renaming, dynamic scheduling, and branch prediction techniques, which reduces pipeline stalls caused by WAR (write after read) and WAW (write after write) hazards, RAW (read after write) hazards, and control hazards respectively. Godson-2E has a 64-entry physical register file for fix- and floating-point register renaming respectively. A 16-entry fix-point reservation station and a 16-entry floating-point reservation station is responsible for out-of-order instruction issuing, while a 64-entry ROQ (reorder queue) ensures that out-of-order executed instructions are committed in the program order. For precise branch prediction, a 16-entry BTB (branch target buffer), a 4K-entry BHT (branch history table), a 9-bit GHR (global history register), and a 4-entry RAS (return address stack) is used to record branch history information.

The memory hierarchy of Godson-2E processor also provides potential for high performance. Godson-2E has a 64KB instruction cache, a 64KB data cache, and a 512KB level-two cache, all four-way set associative. The on-chip 333MHz DDR memory controller allows Godson-2E to achieve high memory bandwidth with low latency. The fully associative TLB of Godson-2E has 64 entries each maps an odd and an even page. A 24-entry memory access queue that contains a content-addressable memory for dynamic memory disambiguation allows Godson-2E to implement out-of-order memory access, non-blocking cache, load speculation, and store forwarding.

Godson-2E has two fix-point functional units, two floating-point functional units, and one memory access unit. The floating-point units can also execute 32- or 64-bit fix-point instructions and 8- or 16-bit SIMD fix-point instructions through extension of the *fmt* field of the floating-point instructions.

The basic pipeline stages of Godson-2E include instruction fetch, pre-decode, decode, register rename, dispatch, issue, register read, execution, and commit. Figure 1-1shows major sections of Godson-2E.

In fetch stage, the instruction cache and instruction TLB (Translation Lookahead Buffer) is read according to the content of PC (program counter). Four new instructions are sent to IR (instruction register) if the instruction fetch is TLB hit and cache hit.

In pre-decode stage, branch instructions are found and their branch directions are dynamically predicted.

In decode stage, the four instructions in IR are decoded into internal format of Godson-2E and are sent to the register renaming module.

In register rename stage, a new physical register is allocated for each logical destination register, and the logical source register is renamed to the latest physical register allocated for the same logical register. Inter-instruction dependencies among four instructions mapped in the same cycle are also checked. The renamed instructions are latched for being sent to reservation stations and queues in next cycle.

In dispatch stage, renamed instructions are dispatched to the fix- or floating-point reservation station for being executed, and are sent to the reorder queue for in-order graduation. Associated instructions are also sent to branch queue and memory queue. Each empty entry of reservation stations and queues selects among four dispatched instructions in this cycle.

In issue stage, one instruction with all required operands ready is selected from the fix- or floating-point reservation station for each functional unit. When there are multiple instructions ready for the same functional unit, the oldest one is selected. Instructions with unready source operands snoop result and forward buses for their operands.

In register read stage, the issued instruction reads its source operands from the physical register file and is sent to the associated functional units. It may also get the data directly from one of the result buses if its source register number matches the destination register number of the result bus.

In execution stage, instructions are executed according to its type and execution results are written back to the register file. Result buses are also sent to the reservation station for snooping and to the register mapping table to notify that the associated physical register is ready.

In commit stage, up to four instructions can be committed in program order per cycle. Committed instructions are sent to the register mapping module to confirm the mapping of its destination register and release the old one. They are also sent to the

memory access queue to allow committed store instructions to write cache or memory.

The Godson-2E processor has been physically implemented based on the ASIC flow with some manual placement and a number of crafted cells and macros. To reduce clock cycle time, some data path modules or modules with replicated structure were manually mapped to the cell library from a structural Verilog model, and were manually placed in a bit-sliced way. The crafted cells and macros include some basic cells such as flip-flops, NANDs, NORs, AOIs, MUXs, buffers and inverters with different sizes; some double height cells such as 4-, 6-, or 8-bit comparator, 4-bit flip-flops, and full adder; a 64*64 register file with 4 write ports and 4 read ports; and a special ram macro for TLB. The useful clock skew technique is used for critical path pipeline stage to borrow time from adjacent pipeline stages.

Godson-2E was fabricated with STmicro's 7-metal 90nm CMOS process. The chip includes 47 million transistors, and the area of the chip is 6,800 micrometers by 5,200 micrometers. The highest frequency of the chip is 1.0GHz, in which the power consumption is ranged from 5.0-7.0 watt depending on the applications.

The 1GHz Godson-2E has the peak performance of 4GFLOPS and 8GFLOPS for double- and single-precision floating point calculation respectively. The SPEC CPU2000 rate of Godson-2E is higher than 500. The prototype Linux-PC based on the Godson-2E processor can smoothly run most of desktop applications such as Debian windows system, Mozilla browser, OpenOffice, and mplayer media player, etc.

1.3 Fetching and Decoding

The Godson-2E pipeline begins with the fetch stage, in which four instructions are fetched in parallel at any word alignment within an eight-word instruction cache line. In each cycle, the processor compares tags read from the cache to physical addresses translated from ITLB (instruction TLB) to select the data from the correct way. On cache misses a refill request will be raised.

The sixteen-entry ITLB is a subset of the main TLB. It is different from the main TLB that each ITLB entry maps only one page. When the ITLB misses, the processor creates an internal Godson-2E instruction which looks for the entry in the main TLB and fills the ITLB. Normal TLB exception will rise if the missing page is not in the main TLB too.

In the following pre-decode and decode stages, the four instructions in IR are

decoded into internal instruction format of Godson-2E and are sent to the register renaming module. Only one branch instruction can be decoded in one cycle. BHT is used for predicting direction of conditional branch, while BTB and RAS are used for predicting target pc.

The BHT contains a 9-bit global history register (GHR) and 2K-entry pattern history table (PHT). Each PHT entry has a 2-bit saturating up/down counter. The counter is increased by one if the prediction is right, and is decreased by one otherwise. The high order bit of the counter is used for branch prediction.

The 16-entry BTB predicts the target PC of the jump register instruction. Each BTB entry contains the PC and target PC of the jump register instruction. Besides, a 2-bit saturating up/down counter is associated with each BTB entry. On replacement, entries with counter values 0 or 1 will be replaced prior to others.

MIPS instruction set does not provide call or return instruction, it normally uses branch/jump and link instruction and the "jump register 31" instruction instead. Godson-2E implements a four-entry return address stack. The decoding of a branch and link instruction causes its PC+8 to be pushed to the RAS, while the decoding of a "jump register 31" instruction causes the target PC to be popped from the RAS. Each branch instruction saves the top-of-stack pointer of the RAS to repair the top-of-stack pointer of the RAS after branch misprediction.

1.4 Register Renaming

Godson-2E implements two 64-entry physical register file for fix-point and floating-point register rename. Correspondingly, two 64-entry physical register-mapping tables (PRMT) are maintained to build the relationship between physical and architectural registers. Each PRMT entry has the following fields. (1) *State*: each physical register is in one of four states, MAP_EMPTY, MAP_MAPPED, MAP_WTBK, and MAP_COMMIT. (2) *Name*: the identifier of the associated architectural register to which this physical register is allocated. (3) *Valid*: this bit is used to mark the latest allocation of a given architectural register if more than one physical registers are allocated to it. Besides, The PRMT also includes fields used to restore the register mapping on mispredicted branch canceling.

In register rename stage, the PRMT is associatively looked up for the two source register *src1, src2* and the destination register *dest* of each instruction to find the associated latest mapped physical register *psrc1, psrc2,* and *odest*. Besides, a free

physical register *pdest* whose state is MAP_EMPTY is allocated to the destination register *dest*, and the state of the newly allocated physical register is set to MAP_MAPPED. The valid bit of the *pdest* entry is set to “1” and the valid bit of the *odest* entry is set to “0” to reflecting that *pdest* becomes the latest allocated physical register for the *dest* architectural register.

Since four instructions are mapped concurrently, inter-instruction dependencies among instructions mapped at the same cycle should be checked. If the source register *src1* of an instruction is identical to the destination register *dest* of a previous instruction mapped at the same cycle, the physical register corresponds to *src1* should be *pdest* of this previous instruction, rather than the *psrc1* looked up from the PRMT. This is also true for *psrc2* and *odest*.

Since register renaming, the processor determines dependencies simply by comparing physical register name. These physical register names *psrc1, psrc2,* and *pdest* are sent to the reservation station, while the *odest* field is kept in the reorder queue. After an instruction is executed, its associated PRMT entry is set to MAP_WTBK state so that following instructions that read this physical register know that the value is ready in the register file. When an instruction is committed, it sets the *pdest* entry of PRMT to MAP_COMMIT state and the *odest* entry to MAP_EMPTY state, which means its destination register contents is regarded as the processor state and the previous contents for this destination register is discarded.

It can be seen from the above register rename process that there may be multiple physical registers allocated to the same architectural register because a logical register may have a sequence of values as it is written by instructions in the pipeline. Physical registers assigned to the same logical register hold both committed values and temporary results as instructions flow through the pipeline. A physical register is written exactly once for each assignment of it.

1.5 Issuing and Reading operands

Register renamed instructions are latched and then sent to the reservation station to be scheduled for execution. Godson-2E has two independent group reservation stations. Fix-point and memory instructions are sent to the fix-point reservation station. Floating-point instructions are sent to the floating-point reservation station. Each reservation station has 16 entries and can accept as many as four instructions per cycle.

In the register rename stage, the PRMT is looked up to see whether the associated operand has been generated and written back to the physical register. If the PRMT indicates that operand is not ready, the reservation station snoops the result buses and forward buses for that operand. The associated ready bit is set to ready if the destination register of one of the snooped buses matches the source register of incoming instructions or instructions in the reservation station.

Result and forward buses stem from the five functional units. The result buses send out the execution results of functional units, while the forward buses forecast which result will be sent out in next cycle. By snooping the forward buses, issued instructions can get operands directly from the result buses before they are written back to the register file. Hence, there is no delay slot for one-cycle instructions such as fix-point add and subtract, shift, and logic instructions.

The reservation stations can issue as many as five operand-ready instructions to the five functional units. If there are multiple operand-ready instructions for the same functional unit, the oldest one is issued. To record the age of each instruction, an *age* field is added to each entry of the reservation station. It is set to a low value when an instruction enters the reservation station, and is increased by one each time an instruction of the same functional unit enters the reservation station.

Issued instructions read their operands from the physical register file. Godson-2E has one fix-point physical register file and one floating-point physical register file, both with the size of 64*64. Issued instructions read operands from the register file before they are sent to functional units for execution.

The fix-point register file has three write ports and seven read ports. The ALU1 fix-point unit uses one write port and three read ports (for move conditional instructions), while the ALU2 and the memory unit uses one write port and two read ports each. The floating-point register file has three write ports and seven read ports. The FALU1 and FALU2 floating-point unit uses one write port and three read ports (for MAC instruction) each. Besides, floating-point load instructions use one write port and floating-point store instructions use one read port of the floating-point register file.

Execution results are written back directly to the register file, and can also be bypassed to following instructions which is RAW dependent on it.

1.6 Execution and Functional Units

Instructions are sent to functional or memory units for execution after reading operations. Godson-2E has two fix-point functional units ALU1 and ALU2, and two floating-point functional units FALU1 and FALU2.

The ALU1 unit executes fix-point addition, subtraction, logical, shift, comparison, trap, conditional move, and branch instructions. All ALU1 instructions are executed and written back in one cycle and have no delay slot with the help of forwarding logic.

The ALU2 unit executes fix-point addition, subtraction, logical, shift, comparison, multiplication, and division instructions. Fix-point multiplication is fully pipelined and has a latency of four cycles. Fix-point division uses the SRT algorithm and is not fully pipelined, the latency of fix-point division ranges from 4 to 37 cycles depending on the operands. All other ALU2 instructions can be executed and written back in one cycle and have no delay slot with the forwarding logic.

The fully pipelined FALU1 unit executes floating-point addition, subtraction, multiplication, multiplication and accumulation, absolute, negation, conversion, comparison, and branch instructions. The latency of floating-point absolute, negation, comparison and branch are two cycles. The latency of conversion instructions is four cycles. The latency of floating-point addition, subtraction, multiplication, multiplication and accumulation are six cycles.

The FALU2 executes floating-point addition, subtraction, multiplication, multiplication and accumulation, division, and square root instructions. The latency of fully pipelined floating-point addition, subtraction, multiplication, multiplication and accumulation are six cycles. The division and square root use the SRT algorithm and are not fully pipelined. The latency of single/double precision floating-point division ranges from 4 to 10/17 cycles, the latency of floating-point division ranges from 4 to 16/31 cycles, depending on the operands.

The floating-point multiply-add-fused (FMAF) unit has been a key feature in many commercial processors, which executes $C \pm (A \times B)$ as a single instruction, with no intermediate rounding. The standard operations floating-point add and floating-point multiply can be performed using this unit by making B=1 for addition and C=0 for multiplication. In godson-2E processors, both FALU1 and FALU2 floating point unit have a FMAF unit, which executes double or single precision floating-point multiply-add, multiply and addition instructions. It also supports the

paired-single instructions which execute two single floating-point multiplication, addition, multiply-add operation concurrently in one instruction. The FMAF is partitioned in five pipeline stages. The first stage mainly operates the bit inversion and alignment of the significant of C in parallel with the booth encoding of multiply. The second stage uses two 14-2 CSA tree to compress the multiply partial products and the C operator mantissa at the same time. As a consequence, the delay of stage-two and stage-three are balanced in our proposed FMAF pipelined structure, and also we can easily support the paired-single instructions by using two separate CSA tree to operate two single precision operations with little change. To make the combination of addition and rounding possible, we anticipate the normalization (LZA) in stage-three and detect the sign of addition results. The fourth stage encodes the LZA outcome to normalize the carry-save product. In stage five a 51-bits dual adder is used to compute the most-significant bits, and the remaining least-significant bits are input to the logic for the calculation of the carry into the most-significant part and for the calculation of the rounding and sticky bits. Finally the carry and the sticky bits are used to select the two outputs of dual adder to be the result of multiply-add operation.

Besides executing floating-point instructions, the floating-point functional units can also execute 32- or 64-bit fix-point instructions (arithmetic, logic, shift, compare, and branch) and 8- or 16-bit SIMD fix-point instruction through extension of the *fmt* field of the floating-point instructions.

1.7 Commit and Reorder Queue

The reorder queue holds all instructions after register mapping and before they are committed. After instructions are executed and written back, the reorder queue commits them in the program order. The reorder queue can hold as many as 64 instructions concurrently.

Reorder queue can accept as many as four mapped instructions per cycle. Newly entered instructions are set to ROQ_MAPPED state. After the instruction is written back, its state in reorder queue is set to ROQ_WTBK for ordinary instructions and ROQ_BRWTBK for branch instructions. The state of branch instructions are set to ROQ_WTBK after the branch result has been sent to other parts of the processor through the branch bus to justify branch prediction tables and to cancel instructions following mispredicted branches. ROQ_WTBK instructions can be committed if they reach the head of the reorder queue.

Reorder queue graduates as many as four ROQ_WTBK instructions in the queue head per cycle. When an instruction graduates, its *pdest* and *odest* fields are sent to the register mapping module to confirm the mapping of *pdest* entry as the processor state and to free the mapping of *odest* entry, it also informs the memory queue that corresponding store instructions can start to modify memory.

For precise exception handling, exceptions are not processed as soon as they occur. They are recorded in the reorder queue instead. When the exception instruction reaches the head of the reorder queue, the exception information is sent out through exception bus. All following instructions are cancelled, exception information is recorded in the CP0 registers, and the PC is set to the entry point of exception handler.

1.8 Branch canceling and Branch Queue

A branch instruction enters the branch queue at the same time it is sent to the reorder queue and the reservation station. At most one branch instruction can be accepted by the branch queue per cycle. The branch queue can hold as many as eight branch instructions concurrently.

The branch queue provides information necessary for execution when a branch instruction is issued to be executed. The information includes the PC value for branch and link instructions, and the predicted taken bit for conditional branch instructions.

After a branch instruction is executed, execution results specific to branch instructions are written back to the branch queue. The results include the target PC for JR and JALR instructions, the branch direction for conditional branch instructions, and a bit indicating whether the branch prediction is error. The branch instruction execution result should be feedback to the instruction fetch part before it can be committed. Besides correcting mispredicted branches, the branch execution result is also used to justify the BHT, BTB, RAS, and GHR for branch prediction.

In case of incorrect prediction, instructions that following the mispredicted branch instruction should be cancelled. The key issue is for each instruction in the pipeline to decide whether it is before or after the mispredicted branch. Godson-2E divides the continuous instruction stream into basic blocks separated by branch instructions. Each instruction is assigned a branch queue position identifier *brqid* that can be regarded as its basic block number. For branch instruction, this identifier indicates its position in the branch queue; for ordinary instruction, this identifier indicates its previous branch instruction position in the branch queue. In this way,

each instruction can determine its relative position to the mispredicted branch by comparing its *brqid* with the *brqid* of the mispredicted branch. Delay slot instructions should be paid special attention in branch canceling.

1.9 Memory Subsystem

Memory references are issued out-of-order to the address calculation unit. The Godson-2E memory access pipeline is split into four stages. (1) In the first stage, address is calculated and the CAM of TLB is searched to form the index of TLB RAM. (2) In the second stage, TLB RAM is accessed in parallel with cache RAM access. Tag compare is also performed at this stage, but value selection according to tag compare result is delayed to next cycle. (3) In the third stage, access value is formed according to the tag compare result of last stage, memory access exception bits are also form at this stage. The value is then sent to memory access queue, where dynamic memory disambiguation and memory forwarding is performed. (4) Finally the results are written back when ready.

The 64-entry fully associative TLB contains a CAM part that is used to do associative search of virtual addresses and a RAM part which stores physical page numbers and page protect bits. The CAM lookup is done in address calculation stage to avoid the need of asynchronous RAM. To reduce hardware cost, Godson-2E uses 40-bit virtual address and 40-bit physical address instead of the rarely needed 64-bit.

The 64-KB four-way set associative primary data cache is virtually indexed and physically tagged so that accesses can happen in parallel with TLB lookups. The replacement policy is random, but two continuous replacement of the same block is avoided by hardware. To reduce chip area and ease physical design, single port RAM is used for both tag and data. Godson-2E allows simultaneous loads and write-back of stores provided they access different banks to alleviate cache access conflict. When cache port conflict does occur among refills, loads (stores read only the tag array) and write-back of stores (which write cache data only), refills have the highest priority while write-back of stores have the lowest priority.

Memory access queue is the core unit of Godson-2E memory subsystem. It can track up to 24 in-flight memory loads or stores. Loads and stores enter the queue out-of-order, but an in-order architectural memory model is maintained. Multiple cache misses and hit under miss are allowed. Using a physical address CAM, the memory access queue dynamically performs disambiguation and forwarding between

accesses. When a load enters the queue, it checks all older stores for possible bypass for each byte it needs. When a store enters the queue, it checks all younger loads in the queues until another younger store to the same byte to decide whether to forward value to them. The queue snoops cache refill and replace operations too.

The miss queue sits below the memory queue in the Godson-2E memory hierarchy. It connects instruction cache, data cache, L2 cache, DDR memory controller, and SysAD system bus controller. The miss queue accepts both instruction miss requests and data miss requests, accesses L2 cache on L1 cache miss, further accesses lower memory hierarchy through processor interface on L2 cache miss, and deliver L2 cache or memory access results to L1 and/or L2 cache. Miss queue implements the store fill buffer optimization which gathers L1 miss store operations for full modified cache blocks and refill the gathered cache block directly to L1 cache to avoid unnecessary memory access.

The 512KB L2 cache is four-way set associative. The block size of L2 cache is 32-byte which is the same as that of the L1 cache. The L2 cache accepts L2 cache access or refill request from miss queue, and sends access results back to miss queue. It also accepts L1 cache write back requests directly from L1 cache and sends L2 cache write back requests directly to lower level memory hierarchy. The fully pipelined L2 cache of Godson-2E runs at the same frequency as the processor core and has a access latency of five cycles.

The 64-bit on-chip DDR memory controller allows Godson-2E to achieve high memory bandwidth with low latency. The 64-bit SysAD processor interface supports up to eight split transactions and remote memory access capability.

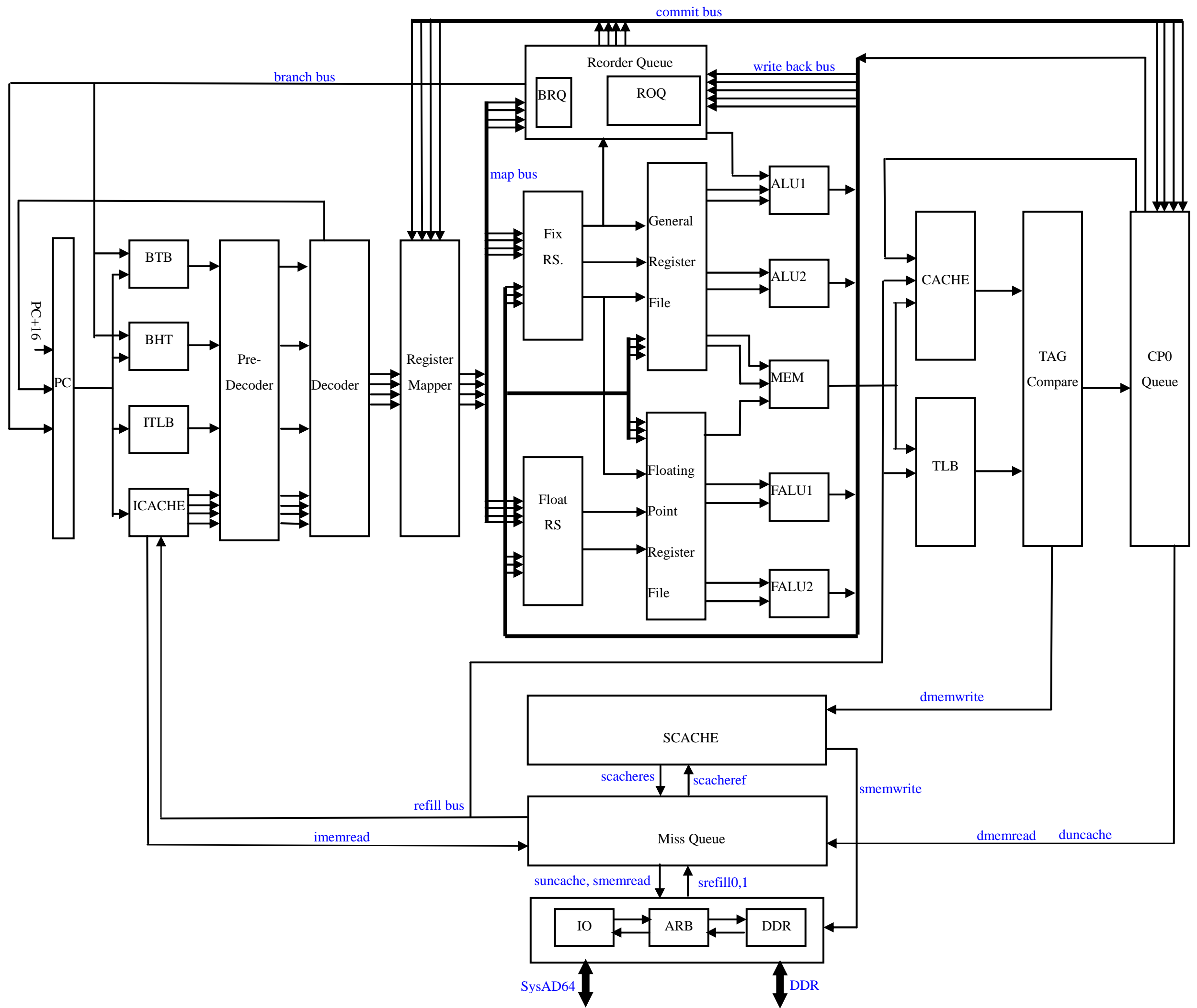
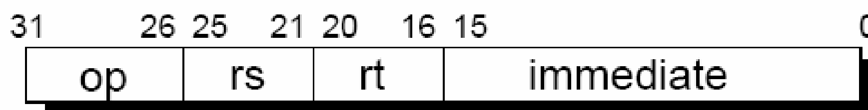


Figure 1-1 Microarchitecture of Godson-2E

2 Instruction Set Overview

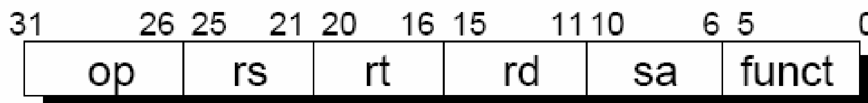
Each CPU instruction consists of a single 32-bit word, aligned on a word boundary. There are three instruction formats—immediate (I-type), jump (J-type), and register (R-type)—as shown in Figure 2-1. The use of a small number of instruction formats simplifies instruction decoding, allowing the compiler to synthesize more complicated (and less frequently used) operations and addressing modes from these three formats as needed.

I-Type (Immediate)



J-Type (Jump)

R-Type (Register)



op	6-bit operation code
rs	5-bit source register specifier
rt	5-bit target (source/destination) register or branch condition
immediate	16-bit immediate value, branch displacement or address displacement
target	26-bit jump target address
rd	5-bit destination register specifier
sa	5-bit shift amount
funct	6-bit function field

Figure 2-1 CPU Instruction Formats

The instruction set can be further divided into the following groupings:

• **Load and Store** instructions move data between memory and general registers.

They are all immediate (I-type) instructions, since the only addressing mode supported is base register plus 16-bit, signed immediate offset.

• **Computational** instructions perform arithmetic, logical, shift, multiply, and divide operations on values in registers. They include register (R-type, in which both the operands and the result are stored in registers) and immediate (I-type, in which one operand is a 16-bit immediate value) formats. Godson-2E also implements self-defined multiply , divide and modulus operations which have a single general purpose destination register instead of the paired hi and lo registers.

• **Jump and Branch** instructions change the control flow of a program. Jumps are always made to a paged, absolute address formed by combining a 26-bit target address with the highorder bits of the Program Counter (J-type format) or register address (R-type format). Branches have 16-bit offsets relative to the program counter (I-type). Jump And Link instructions save their return address in register 31.

• **Coprocessor** instructions perform operations in the coprocessors. Coprocessor load and store instructions are I-type. Godson-2E has two coprocessors: coprocessor 0 (system coprocessor) and coprocessor 1 (float pointer coprocessor).

Coprocessor 0 instructions perform operations on CP0 registers to control the memory management and exception handling facilities of the processor. These are listed in Table 2-9.

Coprocessor 1 instructions include float pointer instructions, multi-media extended instructions and Godson-extended fix pointer computational instructions. They all operate on float pointer registers. Chapter 8 provides summary of these instructions and Appendix B-D give complete description of each instruction.

• **Special** instructions perform system calls and breakpoint operations. These instructions are always R-type.

• **Exception** instructions cause a branch to the general exception handling vector based upon the result of a comparison. These instructions occur in both R-type (both the operands and the result are registers) and I-type (one operand is a 16-bit immediate value) formats.

Tables 2-1 through 2-9 list all prior instructions except coprocessor 1 instructions.

Table 2-1 CPU Instruction Set: Load and Store Instructions

OpCode	Description	MIPS ISA
LB	Load Byte	I
LBU	Load Byte Unsigned	I
LH	Load Halfword	I
LHU	Load Halfword Unsigned	I
LW	Load Word	I
LWU	Load Word Unsigned	I
LWL	Load Word Left	I
LWR	Load Word Right	I
LD	Load Doubleword	III
LDL	Load Doubleword Left	III
LDR	Load Doubleword Right	III
LL	Load Linked	I
LLD	Load Linked Double	III
SB	Store Byte	I
SH	Store Halfword	I
SW	Store Word	I
SWL	Store Word Left	I
SWR	Store Word Right	I
SD	Store Doubleword	III
SDL	Store Doubleword Left	III
SDR	Store Doubleword Right	III
SC	Store Conditional	I
SCD	Store Conditional Double	III
SYNC	Sync	I

Table 2-2 CPU Instruction Set: Arithmetic Instructions (ALU Immediate)

OpCode	Description	MIPS ISA
ADDI	Add Immediate	I
DADDI	Doubleword Add Immediate	III
ADDIU	Add Immediate Unsigned	I
DADDIU	Doubleword Add Immediate Unsigned	III
SLTI	Set on Less Than Immediate	I
SLTIU	Set on Less Than Immediate Unsigned	I
ANDI	And Immediate	I
ORI	Or Immedidate	I
XORI	Exclusive Or Immediate	I
LUI	Load Upper Immediate	I

Table 2-3 CPU Instruction Set: Arithmetic (3-Operand, R-Type)

OpCode	Description	MIPS ISA
ADD	Add	I
DADD	Doubleword Add	III
ADDU	Add Unsigned	I
DADDU	Doubleword Add Unsigned	III
SUB	Subtract	I
DSUB	Doubleword Subtract	III
SUBU	Subtract Unsigned	I
DSUBU	Doubleword Subtract Unsigned	III
SLT	Set on Less Than	I
SLTU	Set on Less Than Unsigned	I
AND	And	I
OR	Or	I
XOR	Exclusive Or	I
NOR	Nor	I

Table 2-4 CPU Instruction Set: Multiply and Divide Instructions

OpCode	Description	MIPS ISA
MULT	Multiply	I
DMULT	Doubleword Multiply	III
MULTU	Multiply unsigned	I
DMULTU	Doubleword Multiply Unsigned	III
DIV	Divide	I
DDIV	Doubleword Divide	III
DIVU	Divide unsigned	I
DDIVU	Doubleword Divide Unsigned	III
MFHI	Move From HI	I
MTHI	Move To HI	I
MFLO	Move From LO	I
MTLO	Move To LO	I
MULTG	Godson-2E Multiply	GODSON-2E
DMULTG	Godson-2E Doubleword Multiply	GODSON-2E
MULTUG	Godson-2E Multiply unsigned	GODSON-2E
DMULTUG	Godson-2E Doubleword Multiply Unsigned	GODSON-2E
DIVG	Godson-2E Divide	GODSON-2E
DDIVG	Godson-2E Doubleword Divide	GODSON-2E
DIVUG	Godson-2E Divide unsigned	GODSON-2E
DDIVUG	Godson-2E Doubleword Divide Unsigned	GODSON-2E
MODG	Godson-2E Modulus	GODSON-2E
DMODG	Godson-2E Doubleword Modulus	GODSON-2E
MODUG	Godson-2E Modulus Unsigned	GODSON-2E
DMODUG	Godson-2E Doubleword Modulus Unsigned	GODSON-2E

Table 2-5 CPU Instruction Set: Jump and Branch Instructions

Opcode	Description	MIPS ISA
J	Jump	I
JAL	Jump and Link	I
JR	Jump Register	I
JALR	Jump And Link Register	I
BEQ	Branch on Equal	I
BNE	Branch on Not Equal	I
BLEZ	Branch on Less Than or Equal to Zero	I
BGTZ	Branch on Greater Than Zero	I
BLTZ	Branch on Less Than Zero	I
BGEZ	Branch on Greater Than or Equal to Zero	I
BLTZAL	Branch on Less Than Zero And Link	I
BGEZAL	Branch on Greater Than or Equal to Zero And Link	I
BEQL	Branch on Equal Likely	II
BNEL	Branch on Not Equal Likely	II
BLEZL	Branch on Less Than or Equal to Zero Likely	II
BGTZL	Branch on Greater Than Zero Likely	II
BLTZL	Branch on Less Than Zero Likely	II
BGEZL	Branch on Greater Than or Equal to Zero Likely	II
BLTZALL	Branch on Less Than Zero And Link Likely	II
BGEZALL	Branch on Greater Than or Equal to Zero And Link Likely	II

Table 2-6 CPU Instruction Set: Shift Instructions

OpCode	Description	MIPS ISA
SLL	Shift Left Logical	I
SRL	Shift Right Logical	I
SRA	Shift Right Arithmetic	I
SLLV	Shift Left Logical Variable	I
SRLV	Shift Right Logical Variable	I
SRAV	Shift Right Arithmetic Variable	I
DSLL	Doubleword Shift Left Logical	III
DSRL	Doubleword Shift Right Logical	III
DSRA	Doubleword Shift Right Arithmetic	III
DSLLV	Doubleword Shift Left Logical Variable	III
DSRLV	Doubleword Shift Right Logical Variable	III
DSRAV	Doubleword Shift Right Arithmetic Variable	III
DSLL32	Doubleword Shift Left Logical + 32	III
DSRL32	Doubleword Shift Right Logical + 32	III
DSRA32	Doubleword Shift Right Arithmetic + 32	III

Table 2-7 CPU Instruction Set: Special Instructions

OpCode	Description	MIPS ISA
SYSCALL	System Call	I
BREAK	Break	I

Table 2-8 CPU Instruction Set: Exception Instructions

OpCode	Description	MIPS ISA
TGE	Trap if Greater Than or Equal	II
TGEU	Trap if Greater Than or Equal Unsigned	II
TLT	Trap if Less Than	II
TLTU	Trap if Less Than Unsigned	II
TEQ	Trap if Equal	II
TNE	Trap if Not Equal	II
TGEI	Trap if Greater Than or Equal Immediate	II
TGEIU	Trap if Greater Than or Equal Immediate Unsigned	II
TLTI	Trap if Less Than Immediate	II
TLTIU	Trap if Less Than Immediate Unsigned	II
TEQI	Trap if Equal Immediate	II
TNEI	Trap if Not Equal Immediate	II

Table 2-9 CP0 Instructions

OpCode	Description	MIPS ISA
DMFC0	Doubleword Move From CP0	III
DMTC0	Doubleword Move To CP0	III
MFC0	Move From CP0	I
MTC0	Move To CP0	I
TLBR	Read Indexed TLB Entry	III
TLBWI	Write Indexed TLB Entry	III
TLBWR	Write Random TLB Entry	III
TLBP	Probe TLB from Matching Entry	III
CACHE	Cache Operation	III
ERET	Exception Return	III

3 Memory Management

The Godson-2E processor provides a full-featured memory management unit (MMU) which uses an on-chip translation lookaside buffer (TLB) to translate virtual addresses into physical addresses.

This section describes the processor virtual and physical address spaces, the virtual-to-physical address translation, the operation of the TLB in making these translations, the cache memories, and those System Control Coprocessor (CP0) registers that provide the software interface to the TLB.

3.1 Translation Lookaside Buffer

Mapped virtual addresses are translated into physical addresses using on-chip Translation Lookaside Buffers (TLB). [1] The primary TLB is the Joint TLB (JTLB). In addition, the Godson-2E processor contains separate Instruction and Data TLBs to avoid contention for the JTLB.

3.1.1 Joint TLB

For fast virtual-to-physical address translation, the Godson-2E uses a large, fully associative TLB that maps virtual pages to their corresponding physical addresses. As indicated by its name, the Joint TLB, or JTLB is used for both instruction and data translations. The JTLB is organized as pairs of even/odd entries, and maps a virtual address and address space identifier into the large, 64GByte physical address space. By default, the JTLB is configured as 64 pairs of even/odd entries to allow the mapping of 128 pages.

Two mechanisms are provided to assist in controlling the amount of mapped space and the replacement characteristics of various memory regions. First, the page size can be configured from 4KB to 16MB (in multiples of 4). A CP0 register, *PageMask*, is loaded with the desired page size of a mapping, and that size is stored into the TLB along with the virtual address when a new entry is written. Thus, operating systems can support different page sizes for different purpose while only

[1] There are virtual-to-physical address translations that occur outside of the TLB. For example, addresses in *the kseg0* and *kseg1* spaces are unmapped translations. In these spaces the physical address is 0x0 0000 0000 || VA[28:0].

one specific page size at the run time. In the future, Godson-2E will support multiple page size at the run time. Thus, operating systems can create special purpose maps; for example, a typical frame buffer can be memory mapped using only one TLB entry.

The second mechanism controls the replacement algorithm when a TLB miss occurs. The Godson-2E provides a random replacement algorithm to select a TLB entry to be written with a new mapping; however, the processor also provides a mechanism whereby a system specific number of mappings can be locked into the TLB, thereby avoiding random replacement. This mechanism allows the operating system to guarantee that certain pages are always mapped for performance reasons and for deadlock avoidance. This mechanism also facilitates the design of real-time systems by allowing deterministic access to critical software.

The JTLB also contains information that controls the cache coherency protocol for each page. Specifically, each page has attribute bits to determine whether the coherency algorithm is: uncached, non-coherent write-back, or uncached accelerated.

3.1.2 Instruction TLB

The Godson-2E use a 8-entry instruction TLB, or ITLB, to minimize contention for the joint TLB, eliminate the timing critical path of translating through a large associative array, and save power. Each ITLB entry maps only one page and the page size is specified by *PageMask* register. The ITLB improves performance by allowing instruction address translation to occur in parallel with data address translation. When a miss occurs on an instruction address translation by the ITLB, a randomly selected ITLB entry is filled from the joint TLB. The operation of the ITLB is completely transparent to the user.

3.1.3 Hits and Misses

If there is a virtual address match, or hit, in the TLB, the physical page number is extracted from the TLB and concatenated with the offset to form the physical address.

If no match occurs (TLB miss), an exception is taken and software refills the TLB from the page table resident in memory. Software can write over a selected TLB entry or use a hardware mechanism to write into a random entry.

3.1.4 Multiple Matches

The Godson-2E processor does not provide any detection or shutdown mechanism for multiple matches in the TLB. Unlike earlier MIPS designs, multiple matches do not physically damage the TLB. Therefore, multiple match detection is not needed. The result of this condition is undefined, and software is expected to never allow this to occur.

3.2 Processor modes

The Godson-2E has three operating modes, but unlike other mips processors, it only supports one addressing mode, one instruction set mode and one endian mode.

3.2.1 Processor Operating Modes

The three operating modes are listed in order of decreasing system privilege:

• **Kernel mode** (highest system privilege): can access and change any register. The innermost core of the operating system runs in kernel mode.

• **Supervisor mode**: has fewer privileges and is used for less critical sections of the operating system.

• **User mode** (lowest system privilege): prevents users from interfering with one another.

Selection between the three modes can be made by the operating system (when in Kernal mode) by writing into *Status* register's *KSU* field. The processor is forced into Kernel mode when the processor is handling an error (the *ERL* bit is set) or an exception (the *EXL* bit is set). Table 3-1 shows the selection of operating modes with respect to the *KSU*, *EXL* and *ERL* bits; the blanks in the table indicate *don't cares*.

Table 3-1 Processor operating modes

KSU 4:3	ERL 2	EXL 1	Description
10	0	0	User mode
01	0	0	Supervisor mode
00	0	0	Kernel mode
	0	1	Exception level
	1		Error level

3.2.2 Addressing mode

Godson-2E processor only supports 64-bit virtual memory addressing mode, but it is compatible with 32-bit virtual memory addressing mode.

3.2.3 Instruction set mode

Godson-2E processor implements a full feature MIPS III Instruction Set Architecture (ISA) plus some MIPS IV ISA instructions, like paired single, move condition and multiply add.

3.2.4 Endian mode

The Godson-2E processor can only operate in little-endian mode.

3.2.5 Address Spaces

This section describes the virtual and physical address spaces and the manner in which virtual addresses are converted or "translated" into physical addresses in the TLB.

3.2.6 Virtual Address Space

The Godson-2E processor has three virtual address spaces: User address space, Supervisor address space and Kernel address space. Each space is 64-bit and consists of several discontinued segments. The maximum segment size is 1 terabyte(2^{40}).

Section 4.3.4 to 4.3.6 describes these virtual address spaces separately.

3.2.7 Physical Address Space

Using a 36-bit address, the Godson-2E processor physical address space encompasses 64 gigabytes.

3.2.8 Virtual-to-Physical Address Translation

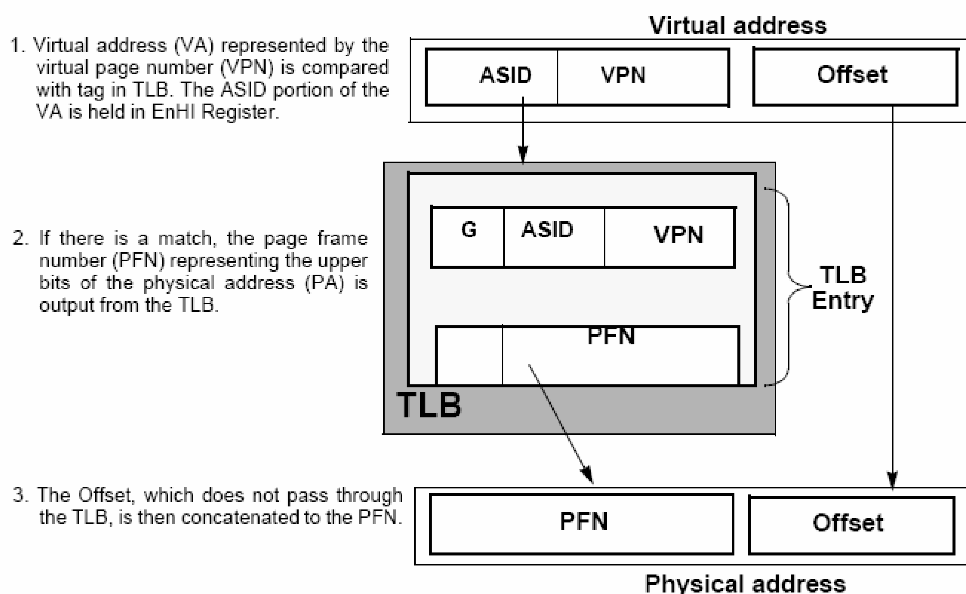


Figure 3-1 Overview of a Virtual-to-Physical Address Translation

Converting a virtual address to a physical address begins by comparing the virtual address from the processor with the virtual addresses in the TLB; there is a match when the virtual page number (VPN) of the address is the same as the VPN field of the entry, and either:

- the ***Global*** (***G***) bit of the TLB entry is set, or
- the ASID field of the virtual address is the same as the ASID field of the TLB entry.

This match is referred to as a *TLB hit*. If there is no match, a TLB Miss exception is taken by the processor and software is allowed to refill the TLB from a page table of virtual/physical addresses in memory.

If there is a virtual address match in the TLB, the physical address is output from the TLB and concatenated with the *Offset*, which represents an address within the page frame space. The *Offset* does not pass through the TLB.

Figure 3-1shows the translation of a virtual address into a physical address. As shown, the virtual address is extended with an 8-bit address space identifier (ASID), which reduces the frequency of TLB flushing when switching contexts. This 8-bit ASID is in the CP0 *EntryHi* register. The ***Global*** bit (***G***) is in each TLB entry.

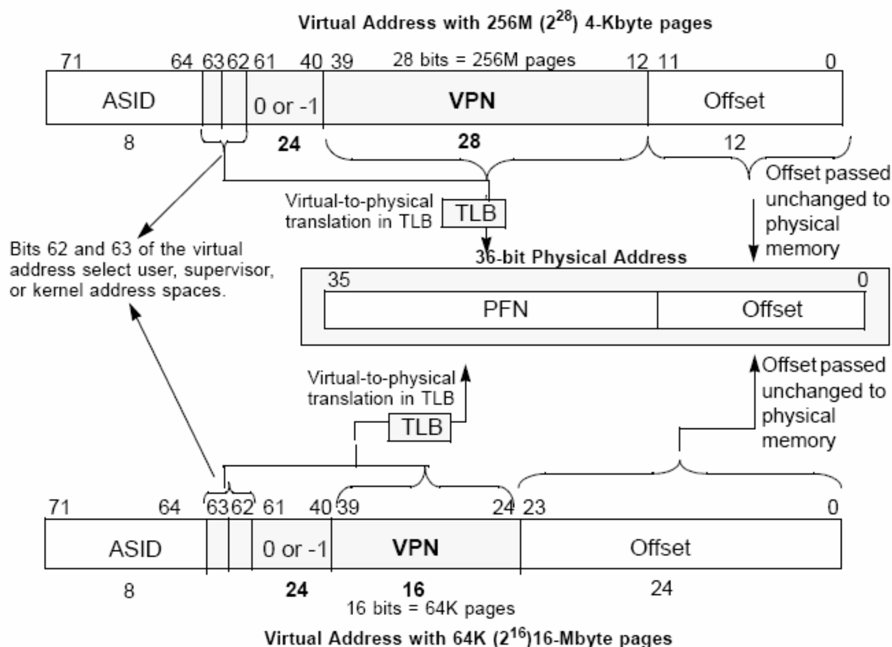


Figure 3-2 64-bit Mode Virtual Address Translation

Figure 3-2 shows the 64-bit mode virtual-to-physical-address translation. This figure illustrates the two extremes in the range of possible page sizes: a 4-Kbyte page (12 bits) and a 16-Mbyte page (24 bits).

• The top portion of Figure 3-2 shows a virtual address with a 12-bit, or 4-Kbyte, page size, labeled *Offset*. The remaining 28 bits of the address represent the VPN, and index the 256Mentry page table.

• The bottom portion of Figure 3-2 shows a virtual address with a 24-bit, or 16-Mbyte, page size, labeled *Offset*. The remaining 16 bits of the address represent the VPN, and index the 64Kentry page table.

3.2.9 User Address Space

In User address space, a single, uniform virtual address space—labeled Extended User segment (*xuseg*), is available and its size is 1 terabyte (2^{40} bytes) .

Figure 3-3 shows the range of User virtual address space. User space can be accessed from user, supervisor, and kernel modes.

The User segment starts at address 0 and the current active user process resides in xuseg. The TLB identically maps all references to xuseg from all modes, and controls cache accessibility.

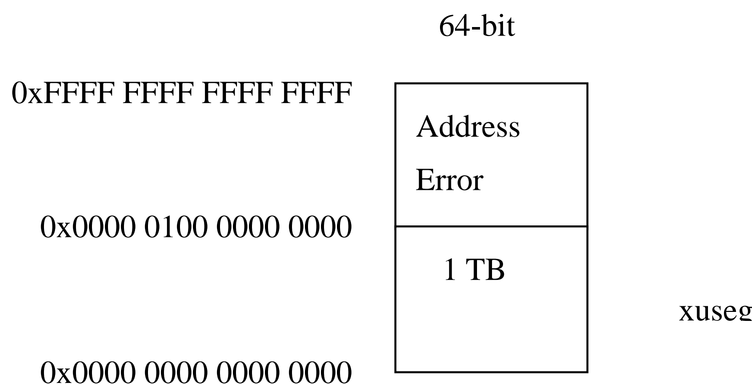


Figure 3-3 User Virtual Address Space as viewed from User Mode

All valid User mode virtual addresses have bits 63:40 equal to 0; an attempt to reference an address with bits 63:40 not equal to 0 causes an Address Error exception. TLB misses on *xuseg* space address use the XTLB refill vector. In Godson-2E processor, XTLB refill vector has the same entry with TLB refill vector in 32-bit mode.

3.2.10 Supervisor Space

Supervisor address space is designed for layered operating systems in which a true kernel runs in Kernel mode, and the rest of the operating system runs in Supervisor mode. The Supervisor address space provides code and data addresses for supervisor mode. TLB misses on supervisor space addresses are handled by the XTLB refill exception handler.

Supervisor space can be accessed from supervisor mode and kernel mode.

The processor operates in Supervisor mode when the *Status* register contains the following bitvalues:

- ***KSU*** = 01_2
- ***EXL*** = 0
- ***ERL*** = 0.

Figure 3-4 shows the User and Supervisor address spaces viewed from Supervisor mode.

64-bit Supervisor, User Space (xsuseg)

In Supervisor Mode when accessing User space and bits 63:62 of the virtual address are set to 00_2, the *xsuseg* virtual address space is selected; it covers the full 240 bytes (1 Tbyte) of the current user address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

This mapped space starts at virtual address 0x0000 0000 0000 0000 and runs through 0x0000 00FF FFFF FFFF.

Address	Region	Segment
0xFFFF FFFF FFFF FFFF	Address Error	
0xFFFF FFFF E000 0000	0.5GB Mapped	csseg
0xFFFF FFFF C000 0000	Address Error	
0x4000 0100 0000 0000	1TB Mapped	xsseg
0x4000 0000 0000 0000	Address Error	
0x0000 0100 0000 0000	1TB Mapped	xsueg
0x0000 0000 0000 0000		

Figure 3-4 User and Supervisor Address Spaces; viewed from Supervisor mode

64-bit Supervisor, Current Supervisor Space (xsseg)

In Supervisor space, when bits 63:62 of the virtual address are set to 01_2, the *xsseg* current supervisor virtual address space is selected. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

This mapped space begins at virtual address 0x4000 0000 0000 0000 and runs through 0x4000 00FF FFFF FFFF.

64-bit Supervisor, Separate Supervisor Space (csseg)

In Supervisor space, when bits 63:62 of the virtual address are set to 11_2, the *csseg* separate supervisor virtual address space is selected. Addressing of the *csseg* is compatible with addressing *sseg* in 32-bit mode. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

This mapped space begins at virtual address 0xFFFF FFFF C000 0000 and runs through 0xFFFF FFFF DFFF FFFF.

3.2.11 Kernel Space

Address	Region	Segment
0xFFFF FFFF FFFF FFFF	0.5GB Mapped	ckseg3
0xFFFF FFFF E000 0000	0.5GB Mapped	cksseg
0xFFFF FFFF C000 0000	0.5GB Unmapped	ckseg1
0xFFFF FFFF A000 0000	0.5GB Unmapped	ckseg0
0xFFFF FFFF 8000 0000	Address Error	
0xC000 00FF 8000 0000	Mapped	xkseg
0xC000 0000 0000 0000	Unmapped	xkphys
0x8000 0000 0000 0000	Address Error	
0x4000 0100 0000 0000	1TB Mapped	xksseg
0x4000 0000 0000 0000	Address Error	
0x0000 0100 0000 0000	1TB Mapped	xkuseg
0x0000 0000 0000 0000		

Figure 3-5 User, Supervisor, and Kernel Address Space viewed from Kernel mode

The processor operates in Kernel mode when the *Status* register contains one of the following values:

- ***KSU*** = 00_2
- ***EXL*** = 1
- ***ERL*** = 1

The processor enters Kernel mode whenever an exception is detected and it remains there until an Exception Return (**ERET**) instruction is executed or the ***EXL*** bit is cleared. The **ERET** instruction restores the processor to the address space existing prior to the exception.

Kernel virtual address space is divided into regions differentiated by the high-order bits of the virtual address, as shown in Figure 3-5. Figure 3-5 also lists the characteristics of the kernel mode segments.

64-bit Kernel, User Space (xkuseg)

In Kernel mode when accessing User space and bits 63:62 of the 64-bit virtual address are 00_2, the *xkuseg* virtual address space is selected; it covers the current user address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

64-bit Kernel, Current Supervisor Space (xksseg)

In Kernel mode when accessing Supervisor space and bits 63:62 of the 64-bit virtual address are 01_2, the *xksseg* virtual address space is selected; it is the current supervisor virtual space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

64-bit Kernel, Physical Spaces (xkphys)

In Kernel space, when bits 63:62 of the 64-bit virtual address are 10_2, the *xkphys* virtual address space is selected; it is a set of eight 2^{36}-byte kernel physical spaces. Accesses with address bits 58:36 not equal to zero cause an address error.

References to this space are not mapped; the physical address selected is taken from bits 35:0 of the virtual address. Bits 61:59 of the virtual address specify the cacheability and coherency attributes, as shown in Table 3-2.

64-bit Kernel, Kernel Space (xkseg)

In Kernel space, when bits 63:62 of the 64-bit virtual address are 11_2, the address space selected is one of the following:

• kernel virtual space, *xkseg*, the current kernel virtual space*;* the virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

• one of the four 32-bit kernel compatibility spaces, as described in the next section.

64-bit Kernel, Compatibility Spaces

In Kernel space, when bits 63:62 of the 64-bit virtual address are 11_2, and bits 61:31 of the virtual address equal –1. The lower two bytes of address select one of the

following 512-Mbyte compatibility spaces.

• *ckseg0*. This 64-bit virtual address space is an unmapped region, compatible with the 32-bit address model *kseg0*. The ***K0*** field of the *Config* register controls cacheability and coherency.

• *ckseg1*. This 64-bit virtual address space is an unmapped and uncached, blocking region, compatible with the 32-bit address model *kseg1*.

• *cksseg*. This 64-bit virtual address space is the current supervisor virtual space, compatible with the 32-bit address model *ksseg*.

• *ckseg3*. This 64-bit virtual address space is kernel virtual space, compatible with the 32-bit address model *kseg3*.

3.3 System Control Coprocessor

The System Control Coprocessor (CP0) supports memory management, address translation, exception handling, and other privileged operations. Godson-2E CP0 contains a 64-entry TLB and 27 registers; each register has a unique identifier referred to as the register number. The sections that follow provide the summary of the memory management-related registers, Chapter 6 gives the complete description of each CP0 register.

3.3.1 Format of a TLB Entry

Figure 3-6 shows the TLB entry formats. Each field of an entry has a corresponding field in the *EntryHi*, *EntryLo0*, *EntryLo1*, or *PageMask* registers.

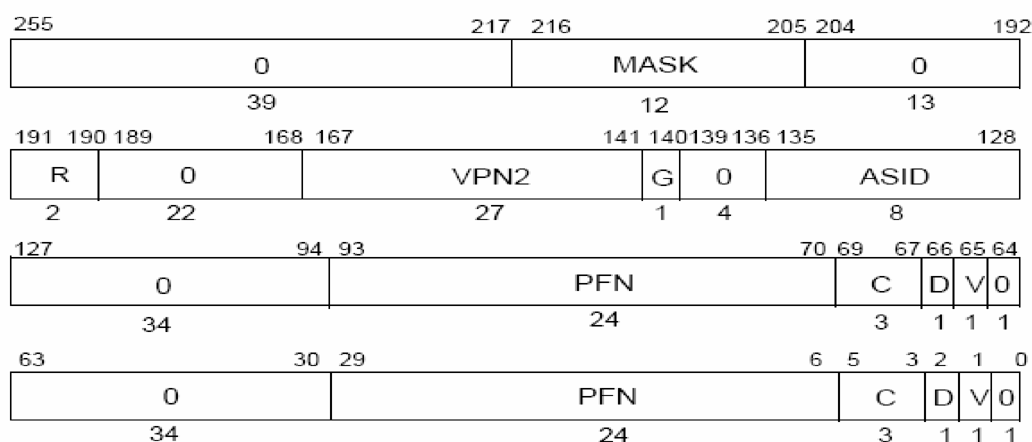
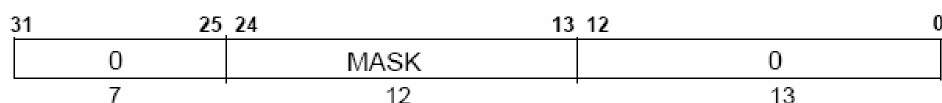


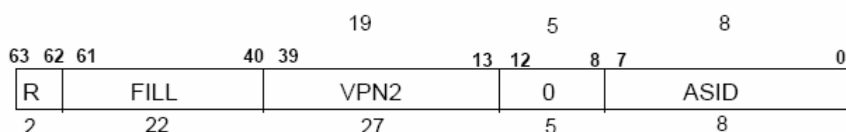
Figure 3-6 Format of a TLB Entry

The format of the *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* registers are nearly the same as the TLB entry. The one exception is the ***Global*** field (***G*** bit), which is used in the TLB, but is reserved in the *EntryHi* register. Figure 3-7, Figure 3-8, and Figure 3-9 describe the formats for the *PageMask*, *EntryHi*, *EntryLo0/EntryLo1* registers.



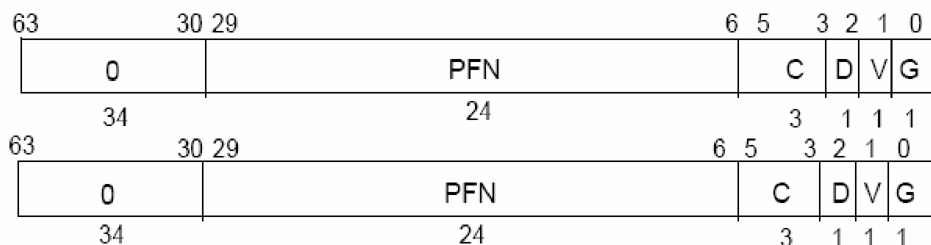
Mask.....Page comparison mask.
0...........Reserved. Must be written as zeroes, and returns zeroes when read.

Figure 3-7 EntryHi Register Format



VPN2 ... Virtual page number divided by two (maps to two pages).
ASID .... Address space ID field. An 8-bit field that lets multiple processes share the TLB; each process has a distinct mapping of otherwise identical virtual page numbers.
R .... Region. (00 → user, 01 → supervisor, 11 → kernel) used to match $vAddr_{63...62}$
Fill .... Reserved. zero on read; ignored on write.
0........... Reserved. Must be written as zeroes, and returns zeroes when read.

Figure 3-8 PageMask Register Format



PFNPage frame number; the upper bits of the physical address.
C Specifies the TLB page coherency attribute; see Table 18.
D Dirty. If this bit is set, the page is marked as dirty and, therefore, writable. This bit is actually a write-protect bit that software can use to prevent alteration of data.
V Valid. If this bit is set, it indicates that the TLB entry is valid; otherwise, a TLBL or TLBS miss occurs.
G Global. If this bit is set in both Lo0 and Lo1, then the processor ignores the ASID during TLB lookup.
0 Reserved. Must be written as zeroes, and returns zeroes when read.

Figure 3-9 EntryLo0 and EntryLo1 Register Formats

The TLB page coherency attribute (*C*) bits specify whether references to the page should be cached; if cached, the algorithm selects between several coherency attributes. Table 3-2 shows the coherency attributes selected by the ***C*** bits.

Table 3-2 TLB Page Coherency (C) Bit Values

C(5:3) Value	Page Coherency Attribute
0	**Reserved**
1	**Reserved**
2	**Uncached**
3	**Cacheable noncoherent (Writeback)**
4	**Reserved**
5	**Reserved**
6	**Reserved**
7	**Uncached Accelerated**

3.3.2 CP0 Registers

Table 3-3 lists the CP0 registers used by the MMU, chapter 6 provides complete description of each CP0 registers..

Table 3-3 Memory Management-Related CP0 Registers

Register No.	Register Name
0	Index
1	Random
2	EntryLo0
3	EntryLo1
5	PageMask
6	Wired
10	EntryHi
15	PrID
16	Config
17	LLAdr
28	TagLo
29	TagHi

3.3.3 Virtual-to-Physical Address Translation Process

During virtual-to-physical address translation, the CPU compares the 8-bit ASID (if the ***Global*** bit, ***G***, is not set) of the virtual address to the ASID of the TLB entry to see if there is a match. And the highest 15-to-27 bits (depending upon the page size) of the virtual address are compared to the contents of the TLB virtual page number also. If a TLB entry matches, the physical address and access control bits (***C***, ***D***, and ***V***) are retrieved from the matching TLB entry. While the ***V*** bit of the entry must be set for a valid translation to take place, it is not involved in the determination of a matching TLB entry.

Figure 3-10 illustrates the TLB address translation process.

3.3.4 TLB Exceptions

If there is no TLB entry that matches the virtual address, a TLB miss exception occurs. If the access control bits (***D*** and ***V***) indicate that the access is not valid, a TLB modification or TLB invalid exception occurs. If the ***C*** bits equal 010_2, the physical address that is retrieved accesses main memory, bypassing the cache.

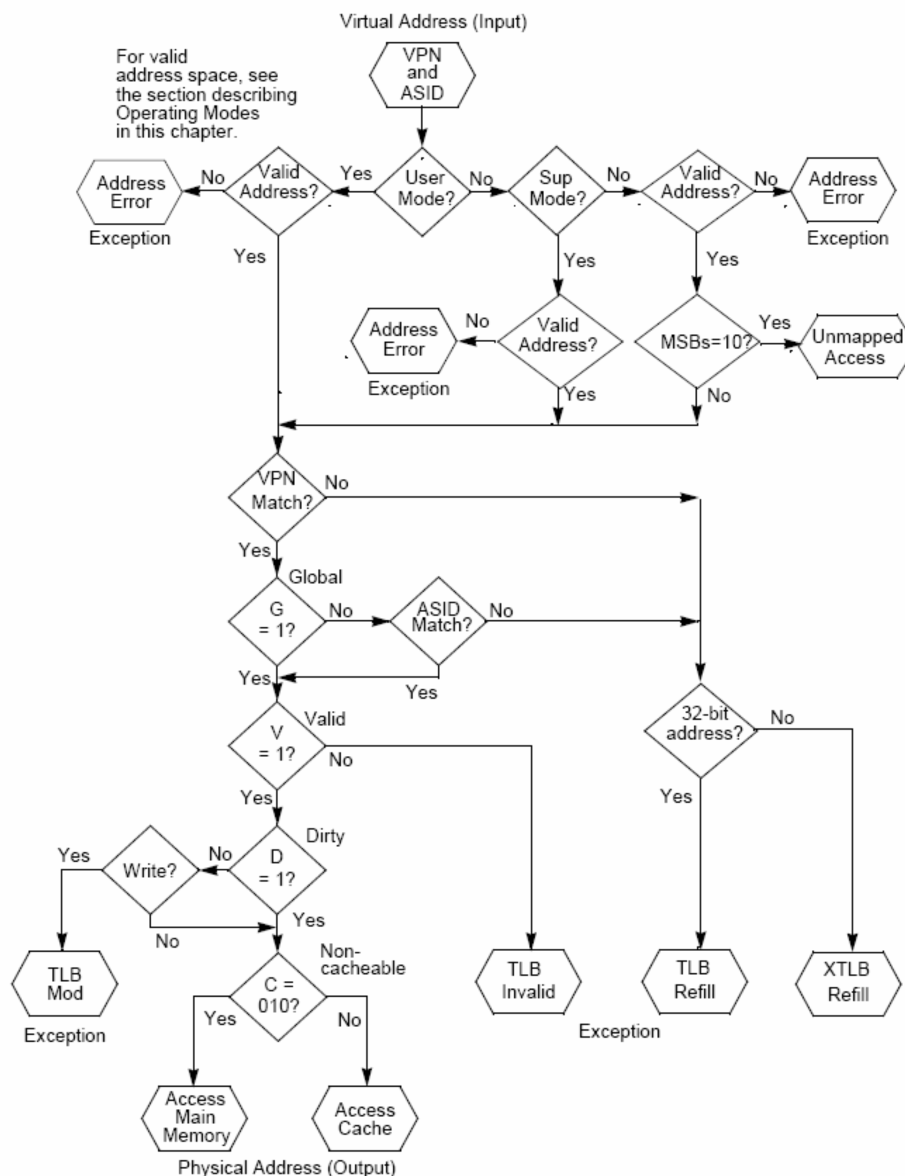


Figure 3-10 TLB Address Translation

3.3.5 TLB Instructions

Table 3-4 lists the instructions that the CPU provides for working with the TLB.

Table 3-4 TLB Instructions

Op Code	Description of Instruction
TLBP	Translation Lookaside Buffer Probe
TLBR	Translation Lookaside Buffer Read
TLBWI	Translation Lookaside Buffer Write Index
TLBWR	Translation Lookaside Buffer Write Random

3.3.6 Code examples

The first example is how to set up one TLB entry to map a pair of 4KB pages. A real time kernel might do something similar. Such simple kernels are only using the MMU for memory protection so the static mapping is sufficient. In statically mapped systems, all TLB exceptions are considered error conditions (access violations).

```
mtc0 r0,C0_WIRED # make all entries available to random replacement
li r2, (vpn2<<13)|(asid & 0xff);
mtc0 r2, C0_ENHI # set the virtual address
li r2, (epfn<<6)|(coherency<<3)|(Dirty<<2)|Valid<<1|Global)
mtc0 r2, C0_ENLO0 # set the physical address for the even page
li r2, (opfn<<6)|(coherency<<3)|(Dirty<<2)|Valid<<1|Global)
mtc0 r2, C0_ENLO1 # set the physical address for the odd page
li r2, 0 # set the page size to 4KB
mtc0 r2,C0_PAGEMASK
li r2, index_of_some_entry # needed for tlbwi only
mtc0 r2, C0_INDEX # needed for tlbwi only
tlbwr # or tlbwi
```

True virtual memory operating systems (like UNIX) use the MMU for both memory protection and swapping pages between main memory and a long term storage device. This mechanism allows programs to address more memory than is physically allocated on the system. This ondemand paging mechanism requires dynamic mapping of pages. The dynamic mapping is implemented through the different types of MMU exceptions. The TLB Refill exception is the most common exception within such systems. The following is an example of a possible TLB Refill exception handler.

```
refill_exception:
mfc0 k0,C0_CONTEXT
sra k0,k0,1 # index into the page table
lw k1,0(k0) # read page table
lw k0,4(k0)
sll k1,k1,6
srl k1,k1,6
mtc0 k1,C0_TLBLO0
sll k0,k0,6
srl k0,k0,6
mtc0 k0,C0_TLBLO1
tlbwr # write a random entry
eret
```

This exception handler is kept very simple and short as it is executed often enough to affect system performance. This is the reason that the TLB Refill exception is allocated its own exception vector. This code assumes that the required mapping has been already set up in the main page table held in main memory. If this is not true then a second exception, a TLB Invalid exception, will be taken after the ERET instruction. The TLB Invalid exception happens much less frequently, which is fortunate as it has to calculate the desired mapping, possibly reading portions of the page table from long term storage. The TLB Mod exception is used to implement read-only pages and to mark which pages have been modified for process cleanup code. To further protect different processes and users from each other, true virtual memory operating systems execute user programs in user mode. Below is an example of how to enter user mode from kernel mode.

```
mtc0 r10, C0_EPC # assume r10 holds desired usermode address
mfc0 r1, C0_SR # get current value of Status register
and r1,r1, ~(SR_KSU || SR_ERL) # clear KSU and ERL field
or r1, r1, (KSU_USERMODE || SR_EXL) # set usermode and EXL bit
mtc0 r1, C0_SR
eret # jump to user mode
```

4 Cache Organization and Operation

The Godson-2E contains three separate caches:

• Primary Instruction Cache: This 64 Kbyte, 4-way set associative cache contains only instruction information.

• Primary Data Cache: This 64 Kbyte, 4-way set associative cache contains only data information.

• Secondary Cache: This on-chip, 512Kbyte, 4-way set associative， write-back cache contains both instruction and data information.

4.1 Cache Overview

The primary caches each require 4 cycles to access. Each primary cache has its own data paths, allowing both caches to be accessed simultaneously. Primary instruction cache has 128-bit read path and 64-bit refill path, while primary data cache has 64-bit read, write and refill data path all.

The secondary cache has a 256-bit data path and is accessed only on a primary cache miss. The secondary cache cannot be accessed in parallel with either of the primary caches and has a 11-cycle miss penalty on a primary cache miss. During a primary instruction or data cache refill, the secondary cache provides 64 bits of data every cycle following the initial 11-cycle latency.

The primary caches are virtually indexed and physically tagged, while the secondary cache is physically indexed and tagged. For current version chips, operating system is obliged to eliminate the potential for virtual aliasing. In the future, hardware would do it.

Having multiple cache hierarchies on-chip means that special consideration must be given during a primary cache flush operation. Without secondary cache, flushing of the primary caches causes the data to be moved to main memory. With secondary cache, using the same code sequence moves data to the secondary cache and secondary cache must be flushed in order to move the data to main memory.

4.1.1 Non-Blocking Caches

The Godson-2E implements a non-blocking architecture for caches. Non-blocking caches improve overall performance by allowing the cache to continue operating even though a cache miss has occurred.

In a typical blocking-cache implementation, the processor executes out of the cache until a miss occurs, at which time the processor stalls until the miss is resolved. The processor initiates a memory cycle, fetches the requested data, places it in the cache, and resumes execution. This operation can take many cycles depending on the design of the memory system.

In a non-blocking implementation, the caches do not stall on a miss. The Godson-2E supports at most 24 outstanding cache misses, which is limited by size of CP0 queue.

When a primary cache miss occurs, the processor checks the secondary cache to determine if the requested data is present. If the data is not present a main memory access is initiated.

The non-blocking caches in the Godson-2E allow for more efficient use of techniques such as loop unrolling and software pipelining. To take maximum advantage of the caches, code should be scheduled to move loads as early as possible, away from instructions that may actually use the data.

To facilitate systems that have I/O devices which depend on in-order loads and stores, the default setting for the Godson-2E is to force uncached references to be blocking.

4.1.2 Replacement Algorithm

The primary caches and secondary cache use random replacement algorithm.

4.1.3 Cache Attributes

Table 4-1 shows the attributes for the three caches.

Table 4-1 Cache attributes

Attribute	Instruction	Data	Secondary
Size	64KB	64KB	512KB
Associativity	4-way	4-way	4-way
Replacement Algorithm	Random	Random	Random
Line size	32 byte	32 byte	32 byte
Index	$vAddr_{13..0}$	$vAddr_{13..0}$	$pAddr_{16..0}$
Tag	$pAddr_{39..12}$	$pAddr_{39..12}$	$pAddr_{39..17}$
Write policy	n.a.	Write-back	Write-back
Read policy	Non-blocking (2 outstanding)	Non-blocking (16 outstanding)	Non-blocking (8 outstanding)
Read Order	Critical word first	Critical word first	Critical word first
Write Order	n.a.	Sequential	Sequential

4.2 Primary Instruction Cache

The primary instruction cache is 64 Kbytes in size and implements a 4-way set associative architecture. Line size is 32-bytes, or eight instructions. The 128-bit read path allows the Godson-2E to fetch four instructions per clock cycle which are passed to the superscalar dispatch unit.

4.2.1 Instruction Cache Organization

The instruction cache is organized as shown in Figure 4-1. The cache is 4-way set associative and contains 512 indexed locations. Each time the cache is indexed, the tag and data portion of each set are accessed. Each of the four tag addresses are compared against the translated portion of the virtual address to determine which set contains the correct data.

When the instruction cache is indexed, each of the four sets shown in Figure 4-1 returns a single cache line. Each cache line consists of 32 bytes of data, a 28-bit physical tag address, and 1 valid bit. Figure 4-2 shows the instruction cache line format.

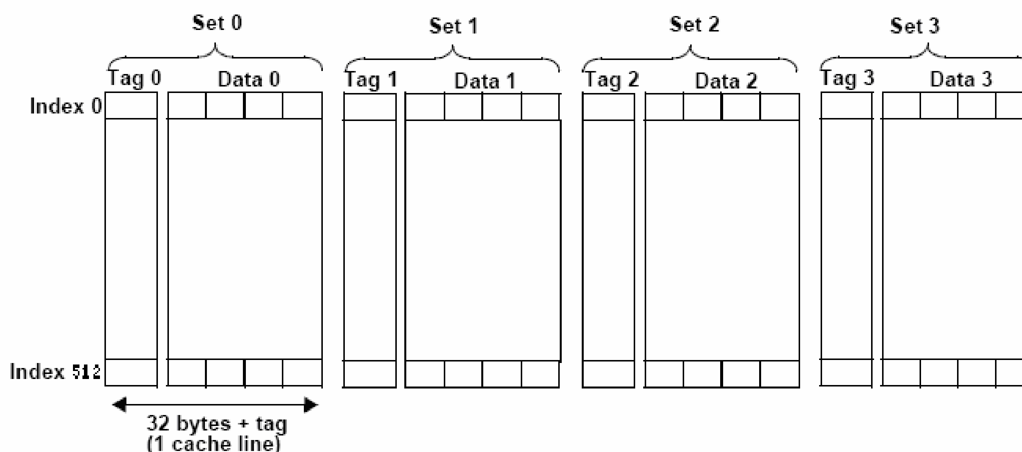


Figure 4-1 Instruction Cache Organization

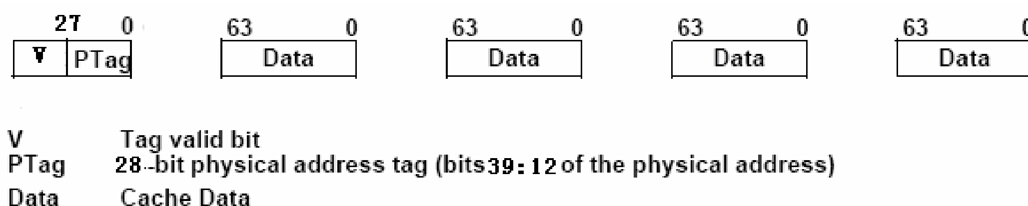


Figure 4-2 Instruction Cache Line Format

4.2.2 Accessing Instruction Cache

The Godson-2E implements a 4-way set associative cache that is virtually indexed and physically tagged. Figure 4-3 shows how the virtual address is divided on an instruction cache access.

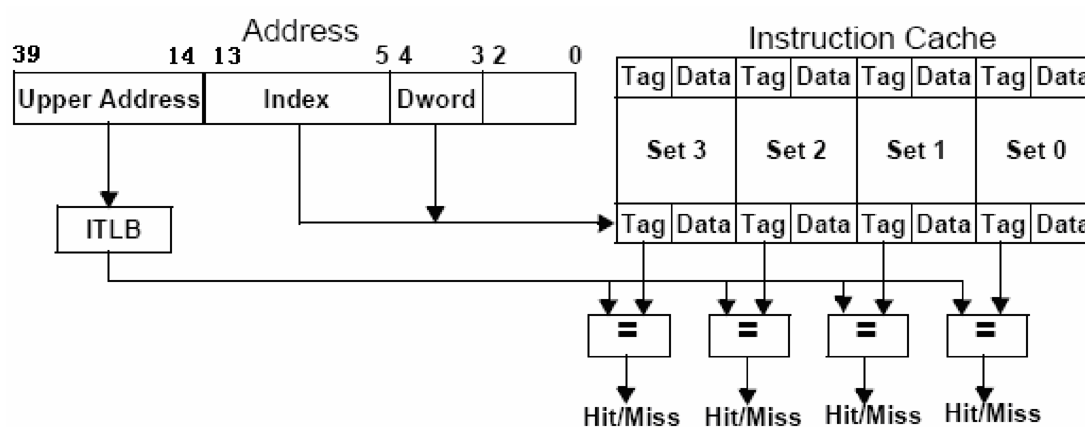


Figure 4-3 Accessing the Instruction Cache

The lower 14 bits of address are used for indexing the instruction cache as shown in Figure 4-3.Bits 13:5 are used for indexing one of the 512 locations. Within each set there are four 64-bit doublewords of data. Bits 4:3 are used to index one of these four doublewords. The tag for each cache line is accessed using address bits 13:5.

When the cache is indexed, the four blocks of data and corresponding physical address tags are fetched from the cache at the same time the upper address is being translated. The translated address from the instruction translation look-aside buffer (ITLB) is compared with each of the four address tags. If any of the four tags yield a valid compare, the data from that set is used. This is called a '*primary cache hit*'. If there is no match between the translated address and any of the four address tags, the cycle is aborted and a secondary cache access is initiated. This is called a '*primary cache miss'*.

4.3 Primary Data Cache

The primary data cache is 64 Kbytes in size and implements a 4-way set associative architecture. Line size is 32-bytes, or eight words. The data cache contains both 64-bit read path and write path. The data cache is used in write-back mode.

The data cache is virtually indexed and physically tagged. Operating system

helps eliminating the potential for virtual aliasing. The data cache is non-blocking, meaning that a miss in the data cache does not stall the pipeline.

The normal write policy is write-back, where a store operation to the data cache does not cause the secondary cache or main memory to be updated. The write-back protocol increases overall system performance by reducing bus traffic. Data is written to the slower memories only when a data cache line is replaced.

4.3.1 Data Cache Organization

The data cache is organized as shown in Figure 4-4. The cache is 4-way set associative and contains 512 indexed locations. Each time the cache is indexed, the tag and data portion of each set are accessed. Each of the four tag addresses are compared against the translated portion of the virtual address to determine which set contains the correct data.

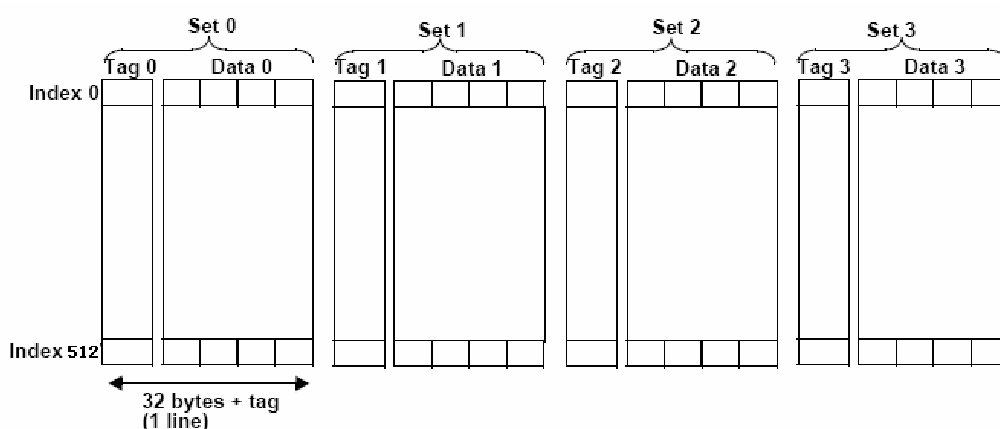


Figure 4-4 Data Cache Organization

When the data cache is indexed, each of the four sets shown in Figure 4-4 returns a single cache line. Each cache line consists of 32 bytes, a 28-bit physical tag address, 1-bit dirty and 2-bit cache status. Figure 4-5 shows the data cache line format.

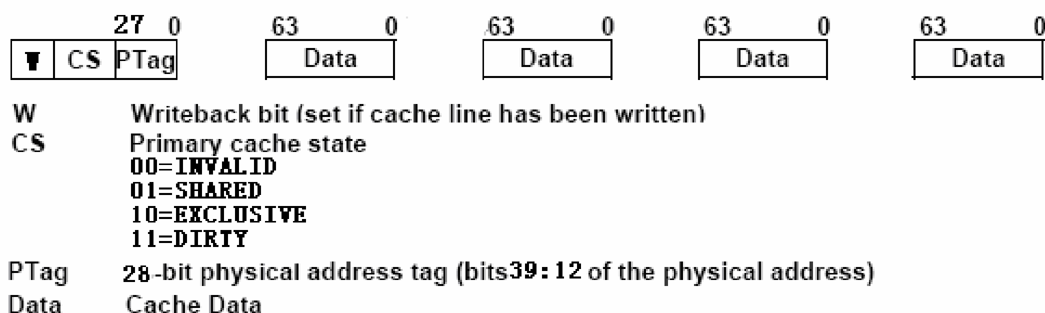


Figure 4-5 Data Cache Line Format

4.3.2 Accessing the Data Cache

The Godson-2E implements a 4-way set associative data cache that is virtually indexed and physically tagged. Figure 4-6 shows how the virtual address is divided on a data cache access.

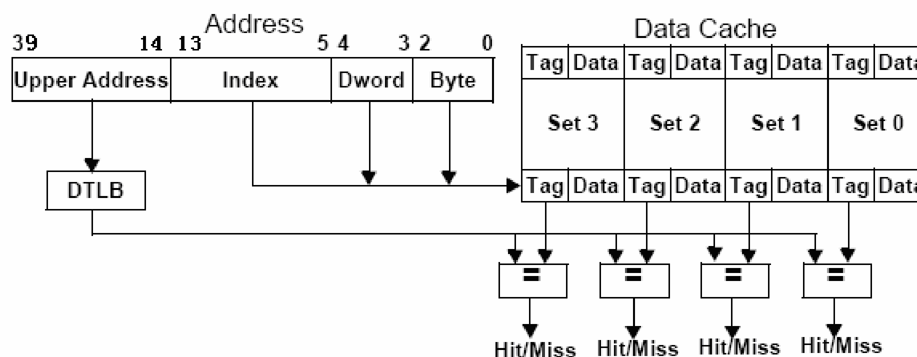


Figure 4-6 Accessing the Data Cache

The lower 14 bits of address are used for indexing the data cache as shown in Figure 4-6. Bits 13:5 are used for indexing one of the 512 locations. Within each set there are four 64-bit doublewords of data. Bits 4:3 are used to index one of these four doublewords. Bits 2:0 are used to index one of the eight bytes within each doubleword. The tag for each cache line is accessed using address bits 13:5.

4.3.3 Processing Data Cache Miss

Data cache load miss accesses secondary cache. If secondary cache hits, the block is fetched from secondary cache and is refill to data cache. If secondary cache misses, memory is accessed. The block is fetched from memory and is refilled to both data cache and secondary cache.

STORE FILL BUFFER policy that improves the bandwidth of microprocessor is adopted on data cache store miss. Data cache store miss instructions are not blocked in CP0 queue to wait for cache refill. If data cache store misses, store instructions committed exit from CP0 queue and are sent to miss queue. Thus this policy decreases CP0 queue full rate. Data cache store miss accesses secondary cache. If secondary cache hits, the block is fetched from secondary cache and is combined with the value that store instruction writes. Then, the block is refill to data cache. If secondary cache misses, secondary cache store miss instruction waits for collecting to fully modified block in miss queue. Fully modified block means that the whole block

is written by store instructions. Fully modified blocks are refilled to both data cache and secondary cache. Fully modified blocks need not access memory. Hence, this policy avoids unnecessary memory traffic. When load instruction accesses the same cache block, miss queue is full, SYNC instruction executes or cache instruction executes, cache store miss entries are not wait for collecting to fully modified block and access the memory. The block is fetched from memory and is combined with the value that store instruction writes. Then, the block is refill to both data cache and secondary cache.

Godson-2E implements STORE FILL BUFFER scheme in miss queue without adding separate store buffer. It decreases the hardware overhead and avoids the query overhead between miss queue and store buffer. STORE FILL BUFFER policy improves the bandwidth of Godson-2E significantly.

4.4 Secondary Cache

The Godson-2E implements an on-chip, four-way associative, write-back secondary cache. The cache size is 512Kbyte, and the line size is 32 bytes.

4.4.1 Secondary Cache Organization

The secondary cache is four-way set associative cache that contains instruction and data information. The Godson-2E supports secondary cache sizes of 512Kbytes.

Each indexed location in the cache contains four 64-bit doublewords. Each time the cache is indexed, the tag and data portion of each set are accessed. The tag address is compared against the translated portion of the virtual address to determine if the data resides in the cache.

When the secondary cache is indexed, each location contains a single cache line. Each cache line consists of 32 bytes of data, a 23-bit physical tag address, and two cache status bits.

4.4.2 Accessing the Secondary Cache

The secondary cache is only accessed on a primary cache miss. Once the processor has determined that the requested address does not match the corresponding primary cache tag, a secondary cache access is initiated. The secondary cache is physically indexed and physically tagged. The accessing process of secondary cache is shown in Figure 4-7.

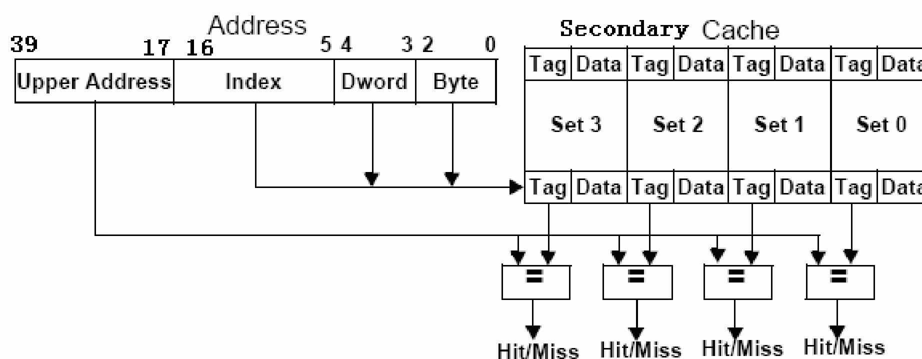


Figure 4-7 Accessing the Secondary Cache

The lower bits of address are used for indexing the data cache as shown in Figure 4-7. Bits 16:5 are used for indexing secondary cache. Within each indexed entry there are four 64-bit doublewords of data. Bits 4:3 are used to index one of these four doublewords. Bits 2:0 are used to index one of the eight bytes within each doubleword. The tag for each cache line is accessed using address bits 16:5.

4.5 Cache Coherency

Systems using more than one master must have a mechanism to maintain data consistency throughout the system. This mechanism is called a cache coherency protocol. The Godson-2E does not provide any hardware cache coherency. Cache coherency must be handled by software.

4.5.1 Cache Coherency Attributes

Cache coherency attributes are necessary to ensure the consistency of data throughout the system. Bits in the translation look-aside buffer (TLB) control coherency on a per-page basis. Specifically, the TLB contains 3 bits per entry that provide the coherency attribute types shown in Table 4-2.

The non-blocking coherencies implement a weakly ordered memory model. This model allows the following behaviors:

• The processor does not have to stall if a processor load request has not completed. Subsequent processor load or store operations may be started before the first has completed.

• Memory transactions can occur on the external pin bus out of program order. Program order of memory transactions can be enforced through the use of the **SYNC** instruction.

• Memory transactions can occur on the external pin bus even though the instructions which caused the memory transactions are later nullified in the pipeline due to an exception.

Such behaviors aid in achieving higher levels of processor throughput. However, some peripheral devices require a strongly ordered memory model (memory transactions occur in program order and only valid instructions can cause memory transactions). For this reason, it is strongly advised that such devices be referenced using the Uncached, Blocking coherency (Coherency Code 2).

Processor read requests using this coherency stall the processor until the transaction completes. Processor write requests using this coherency are given the highest priority for accessing the external pin bus. These two properties ensure that processor load and store instructions using this coherency are completed in program order. For this reason, kseg1 and the uncached section of xkphys use Coherency Code 2.

Table 4-2 Godson-2E Cache Coherency Attribute

Attribute Type	Coherency Code
Reserved	0
Reserved	1
Uncached, Blocking	2
Writeback, non-blocking	3
Reserved	4
Reserved	5
Reserved	6
Uncached Accelerated	7

The following subsections describe each of the coherency attributes listed in Table 4-2.

4.5.2 Uncached, Blocking (Coherency Code 2)

Lines within an *Uncached* page are never in a cache. When a page has the uncached coherency attribute, the processor issues a doubleword, partial-doubleword, word, or partial-word read or write request directly to main memory (bypassing all caches) for any load or store to a location within that page. No caches are accessed when this coherency attribute is active.

Processor read requests using this coherency stall the processor until the transaction completes while processor write requests using this coherency are given the highest priority for accessing the external pin bus. These two properties ensure

that processor load and store instructions using this coherency are completed in program order (strongly ordered memory model).

4.5.3 Writeback (Coherency code 3)

Lines with the *Writeback* attribute can reside in a cache. On a data cache store hit, only the data cache is modified. The secondary cache, and main memory are only modified if the cache line of a dirty block is needed for a newer access.

This mode allows the primary data cache to be filled on either a load miss or a store miss. A primary cache store hit causes data to be written to the primary data cache only. The secondary is modified only during block writebacks and line fills. Partial (non-blocking) stores are never written to the secondary caches. Main memory is modified only for block writebacks.

On a primary cache load or store miss, the Godson-2E checks the secondary cache for the requested address. If there is a secondary cache hit, the data is filled from the secondary cache. If a secondary cache miss occurs, the Godson-2E accesses the main memory with a block read request. Data is fetched from main memory and written to the secondary and primary caches.

This coherency follows the weakly ordered memory model described in section 4.5.1, "Cache Coherency Attributes".

4.5.4 Uncached Accelerated (Coherency Code 7)

Uncached accelerated is used for sequential same type uncached stores at a consecutive address space. A buffer is used to gather these stores until the buffer is full. The buffer size is the same as the cache line. Store to the buffer is just like it does to the cache. When the buffer is full, a block write is initiated. If the sequential stores is intervened by other uncached store, individual uncached stores are executed for the buffer content.

Uuncached accelerated attribute can accelerate sequential uncahced access, and it is useful for access to video memory.

4.6 Cache Maintenance

With multiple levels of on-chip memory, care must be taken to ensure that modified data has reached external memory before a process task switch. To flush all on-chip write buffers, software should use the **SYNC** instruction. This instruction will

stall the processor until all pending store operations have reached the external pin bus and all pending load operations have completed by writing their destination registers.

The **CACHE** instruction is used when performing maintenance of the caches. Godson-2E contains two “Hit” type cache operations for primary data cache: *Hit_Invalidate* and *Hit_Writeback_Invalidate*. The Godson-2E treats the “Hit” type **CACHE** operation much like a load instruction and allows the instruction to be pipelined. If there is no cache hit, the “Hit” type can be executed without any pipeline stall. If there is a cache hit, but the cache line is clean, the only latency incurred is that required for invalidating the tag RAM.

5 CP0

This chapter describes the Coprocessor 0 operations, including the CP0 register definitions and CP0 instructions implemented by the Godson-2E processor. The Coprocessor 0 (CP0) registers are used to control and represent the processor state. These registers can be read using MFC0/DMFC0 instructions and written using MTC0/ DMTC0 instructions. CP0 registers are listed in Table 5-1.

Coprocessor 0 instructions are usable if the processor is in Kernel mode, or bit 28 (*CU0*) of the *Status* register is set. Otherwise, executing one of these instructions generates a Coprocessor 0 Unusable exception.

Table 5-1 Coprocessor 0 Registers

Register No.	Register Name	Description
0	Index	Programmable register to select a TLB entry for reading or writing
1	Random	Pseudo-random counter for the TLB replacement
2	EntryLo0	Low half of the TLB entry for the even VPN (Physical page number)
3	EntryLo1	Low half of the TLB entry for the odd VPN (Physical page number)
4	Context	Pointer to the kernel virtual PTE table in the 32-bit addressing mode
5	Page Mask	Mask that decides the TLB page size
6	Wired	Number of wired TLB entries (the floor of the random replacement range, that is, the lowest TLB entry that can be used for random replacement)
7		Reserved
8	BadVaddr	Bad virtual address
9	Count	Clock counter
10	EntryHi	High half of the TLB entry (Virtual page number and ASID)
11	Compare	Counter compare
12	Status	Processor Status Register
13	Cause	Cause of the last exception
14	EPC	Exception Program Counter
15	PRID	Processor Revision Identifier
16	Config	Configuration Register (primary cache size, etc.)
17	LLAddr	Load Linked memory address
18	WatchLo	
19	WatchHi	
20	Xcontext	Pointer to the kernel virtual PTE table in the 64-bit addressing mode

Register No.	Register Name	Description
21		Reserved
22	Diagnose	Enable/disable BTB, RAS and flush ITLB
23		Reserved
24	PCLo	Low half of Performance Counter
25	PCHi	High half of Performance Counters
26		Reserved
27		Reserved
28	TagLo	Cache Tag register - low bits
29	TagHi	Cache Tag register - high bits
30	ErrorEPC	Error Exception Program Counter
31		

5.1 Index Register (0)

The *Index* register is a 32-bit, read/write register containing six bits to index an entry in the TLB. The highest-order bit of the register indicates the success or failure of a TLB Probe (TLBP) instruction.

The *Index* register also specifies the TLB entry accessed by TLB Read (TLBR) or TLB Write Index (TLBWI) instructions.

Figure 5-1 shows the format of the *Index* register; Table 5-2 describes the *Index* register fields.

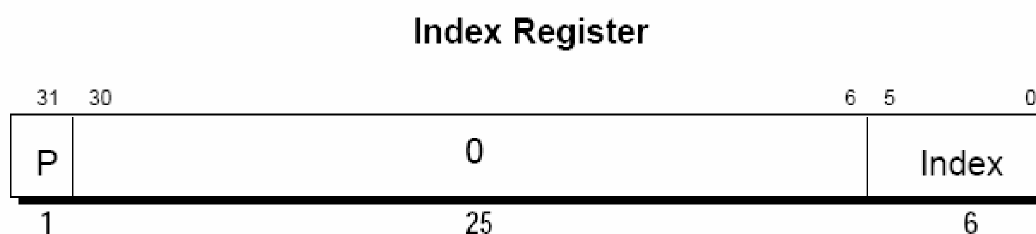


Figure 5-1 Index Register

Table 5-2 Fields in the Index Register

Field	Description
P	Probe failure. Set to 1 when the last TLBProbe (TLBP) instruction was unsuccessful.
Index	Index to the TLB entry accessed by the TLBRead and TLBIWrite instructions
0	Reserved. Must be written as zeroes, and returns zeroes when read.

5.2 Random Register (1)

The *Random* register is a read-only register of which the lowest six bits index an entry in the TLB. This register decrements when any instruction graduates at that particular cycle, and its value ranges between an upper and a lower bound, as follows:

• The lower bound is set by the number of TLB entries reserved for exclusive use by the operating system (the content of the *Wired* register).

• The upper bound is set by the total number of TLB entries minus 1 (64 – 1 maximum).

The *Random* register specifies the entry in the TLB that is affected by the TLB Write Random instruction. The register does not need to be read for this purpose; however, the register is readable to verify proper operation of the processor.

To simplify testing, the *Random* register is set to the value of the upper bound upon system reset. This register is also set to the upper bound when the *Wired* register is written.

Figure 5-2 shows the format of the *Random* register;Table 5-3 describes the *Random* register fields.

Figure 5-2 Random Register

Table 5-3 Fields in the Random Register

Field	Description
Random	TLB random index
0	Reserved. Must be written as zeroes, and returns zeroes when read.

5.3 EntryLo0 (2), and EntryLo1 (3) Registers

The *EntryLo* register consists of two registers with identical formats:

• *EntryLo0* is used for even virtual pages.

• *EntryLo1* is used for odd virtual pages.

The *EntryLo0* and *EntryLo1* registers are read/write registers. They hold the physical page frame number (PFN) of the TLB entry for even and odd pages

respectively for TLB read and write operations.Figure 5-3 shows the format of these registers.

EntryLo0 and EntryLo1 Registers

63 62	61	60 34	33 6	5 3	2	1	0
0	E	0	PFN	C	D	V	G
2	1	27	28	3	1	1	1

Figure 5-3 Fields of the EntryLo0 and EntryLo1 Registers

The *PFN* fields of the *EntryLo0* and *EntryLo1* registers span bits 39:12 of the 40-bit physical address.

Two additional bits for the mapped space's *uncached attribute* can be loaded into bits 63:62 of the *EntryLo* register, which are then written into the TLB with a TLB Write. During the address cycle of processor double/single/partial-word read and write requests, and during the address cycle of processor *uncached accelerated* block write requests, the processor drives the uncached attribute on **SysAD[59:58]**. The same *EntryLo* registers are used for the 64-bit and 32-bit addressing modes. In both modes the registers are 64 bits wide, however when the MIPS III ISA is not enabled (32-bit User and Supervisor modes) only the lower 32 bits of the *EntryLo* registers are accessible.

Table 5-4 Description of EntryLo Registers' Fields

Field	Description
E	Non-executable. 1 means non-executable, 0 means executable.
PFN	Page frame number; the higher bits of the physical address.
C	Specifies the TLB page coherence attribute.
D	Dirty. If this bit is set, the page is marked as dirty and, therefore, writable. This bit is actually a write-protect bit that software can use to prevent alteration of data.
V	Valid. If this bit is set, it indicates that the TLB entry is valid; otherwise, a TLBL or TLBS invalid exception occurs.
G	Global. If this bit is set in both Lo0 and Lo1, then the processor ignores the ASID during TLB lookup.
0	Reserved. Must be written as zeroes, and returns zeroes when read.

MIPS III is disabled when the processor is in 32-bit Supervisor or User mode. Loading of the integer registers is limited to bits *31:0*, sign-extended through bits *63:32*. *EntryLo[33:31]* or *PFN[39:38]* can only be set to all zeroes or all ones. In 32-and 64-bit modes, the *UC* and *PFN* bits of both *EntryLo* registers are written into

the TLB. The *PFN* bits can be masked by setting bits in the *FrameMask* register (described in this chapter) but the *UC* bits cannot be masked or initialized in 32-bit User or Supervisor modes. In 32-bit Kernel mode, MIPS III is enabled and 64-bit operations are always available to program the *UC* bits.

There is only one *G* bit per TLB entry, and it is written with *EntryLo0[0]* and *EntryLo1[0]* on a TLB write.

5.4 Context (4)

The *Context* register is a read/write register containing the pointer to an entry in the page table entry (PTE) array; this array is an operating system data structure that stores virtual-to-physical address translations.

When there is a TLB miss, the CPU loads the TLB with the missing translation from the PTE array. Normally, the operating system uses the *Context* register to address the current page map which resides in the kernel-mapped segment, *kseg3*. The *Context* register duplicates some of the information provided in the *BadVAddr* register, but the information is arranged in a form that is more useful for a software TLB exception handler.

Figure 5-4 shows the format of the *Context* register;Table 5-5 describes the *Context* register fields.

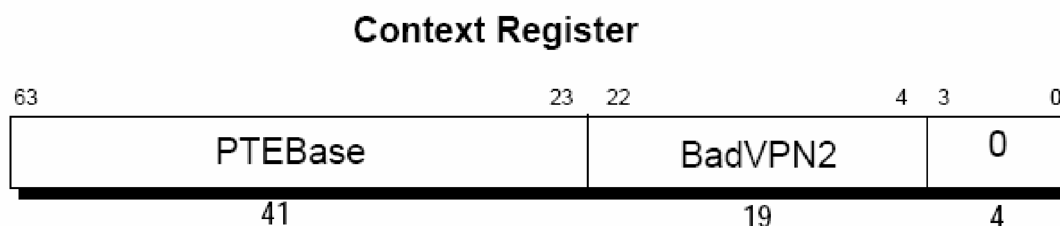


Figure 5-4 Context Register Format

Table 5-5 Context Register Fields

Field	Description
BadVPN2	This field is written by hardware on a miss. It contains the virtual page number (VPN) of the most recent virtual address that did not have a valid translation.
PTEBase	This field is a read/write field for use by the operating system. It is normally written with a value that allows the operating system to use the *Context* register as a pointer into the current PTE array in memory.
0	Reserved. Must be written as zeroes, and returns zeroes when read.

The 19-bit *BadVPN2* field contains bits 31:13 of the virtual address that caused the TLB miss; bit 12 is excluded because a single TLB entry maps to an even-odd page pair. For a 4-Kbyte page size, this format can directly address the pair-table of 8-byte PTEs. For other page and PTE sizes, shifting and masking this value produces the appropriate address.

5.5 PageMask Register(5)

The *PageMask* register is a read/write register used for reading from or writing to the TLB; it holds a comparison mask that sets the variable page size for each TLB entry, as shown in Table 5-6. Format of the register is shown inFigure 5-5.

TLB read and write operations use this register as either a source or a destination; when virtual addresses are presented for translation into physical address, the corresponding bits in the TLB identify which virtual address bits among bits 24:13 are used in the comparison. When the *Mask* field is not one of the values shown in Table 5-6, the operation of the TLB is undefined. The 0 field is reserved; it must be written as zeroes, and returns zeroes when read.

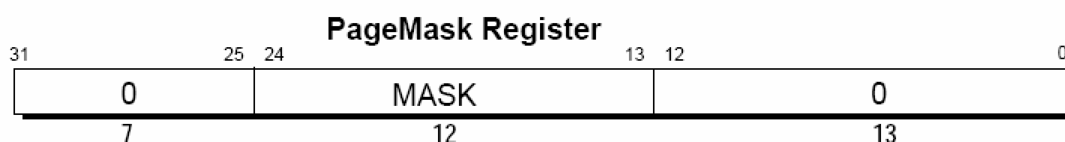


Figure 5-5 PageMask Register

Table 5-6 Mask Field Values for Page Sizes

Page Size	Bit											
(Mask)	24	23	22	21	20	19	18	17	16	15	14	13
4Kbytes	0	0	0	0	0	0	0	0	0	0	0	0
16 Kbytes	0	0	0	0	0	0	0	0	0	0	1	1
64 Kbytes	0	0	0	0	0	0	0	0	1	1	1	1
256 Kbytes	0	0	0	0	0	0	1	1	1	1	1	1
1 Mbytes	0	0	0	0	1	1	1	1	1	1	1	1
4 Mbytes	0	0	1	1	1	1	1	1	1	1	1	1
16M bytes	1	1	1	1	1	1	1	1	1	1	1	1

5.6 Wired Register (6)

The *Wired* register is a read/write register that specifies the boundary between the *wired* and *random* entries of the TLB as shown inFigure 5-6. Wired entries are fixed, nonreplaceable entries, which cannot be overwritten by a TLB write operation. Random entries can be overwritten.

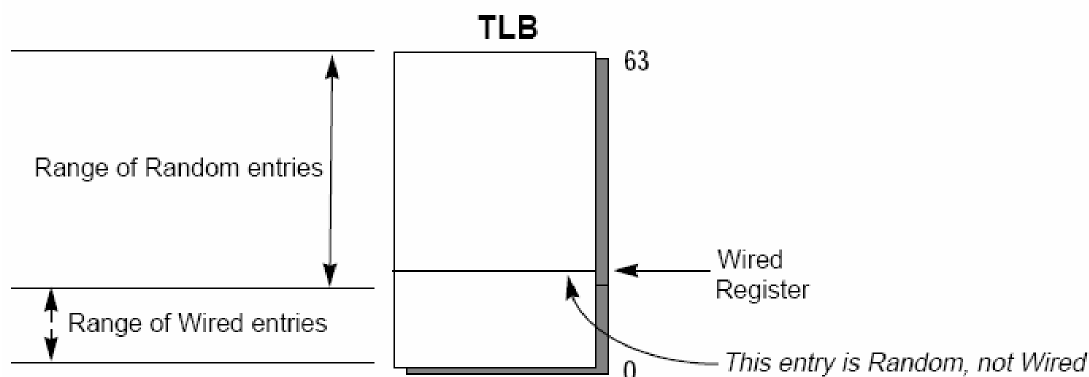


Figure 5-6 Wired Register Boundary

The *Wired* register is set to 0 upon system reset. Writing this register also sets the *Random* register to its upper bound value (see *Random* register, above).

Figure 5-7 Wired Register

Figure 5-7 shows the format of the *Wired* register;Table 5-7 describes the register fields.

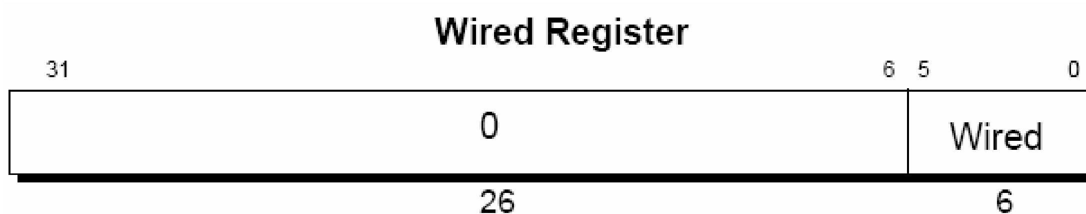


Figure 5-7 Wired Register

Table 5-7 Fields in the Wired Register Field

Field	Description
Wired	TLB Wired boundary
0	Reserved. Must be written as zeroes, and returns zeroes when read.

5.7 BadVAddr Register (8)

The Bad Virtual Address register (*BadVAddr*) is a read-only register that displays the most recent virtual address that caused either a TLB or Address Error

exception. The *BadVAddr* register remains unchanged during Soft Reset, NMI, or Cache Error exceptions. Otherwise, the architecture leaves this register undefined.

Figure 5-8 shows the format of the *BadVAddr* register.

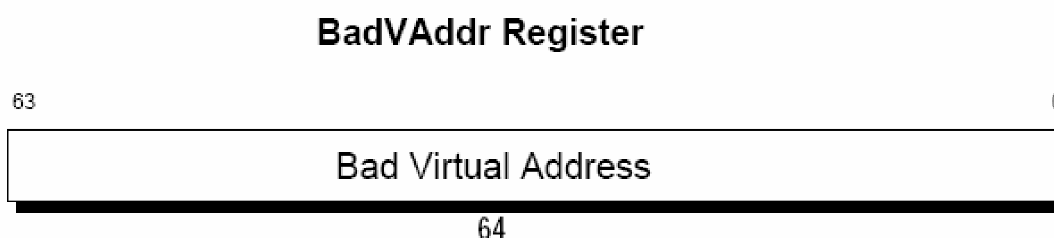


Figure 5-8 BadVAddr Register Format

5.8 Count and Compare Registers(9 and 11)

The *Count* and *Compare* registers are 32-bit read/write registers whose formats are shown inFigure 5-9.

The *Count* register acts as a real-time timer. Like the R4400 implementation, the Godson-2E *Count* register is incremented every *other* **PClk** cycle. However, unlike the R4400, the Godson-2E processor has no Timer Interrupt Enable boot-mode bit, so the only way to disable the timer interrupt is to negate the interrupt mask bit, *IM[7]*, in the *Status* register. This means the timer interrupt cannot be disabled without also disabling the *Performance Counter* interrupt, since they share *IM[7]*.

The *Compare* register can be programmed to generate an interrupt at a particular time, and is continually compared to the *Count* register. Whenever their values equal, the interrupt bit *IP[7]* in the *Cause* register is set. This interrupt bit is reset whenever the *Compare* register is written.

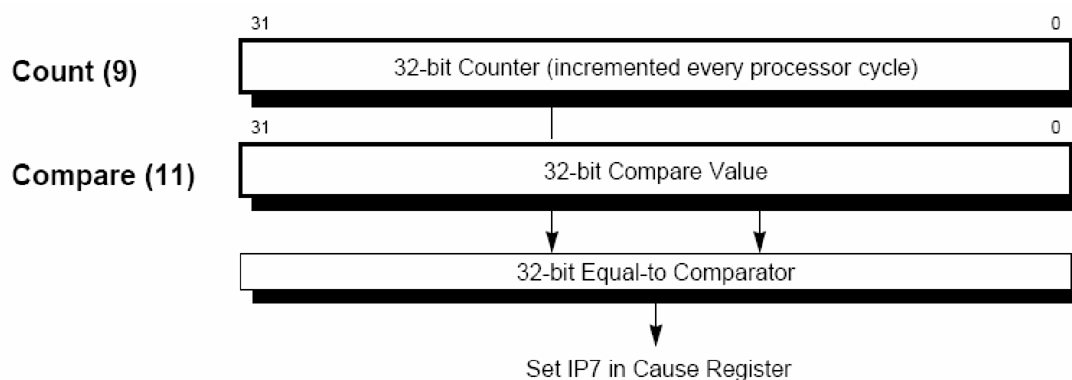


Figure 5-9 Count and Compare Registers

5.9 EntryHi Register (10)

The *EntryHi* register holds the high-order bits of a TLB entry for TLB read and write operations.

The *EntryHi* register is accessed by the TLB Probe, TLB Write Random, TLB Write Indexed, and TLB Read Indexed instructions.

Figure 5-10 shows the format of this register and Table 5-8 describes the register's fields.

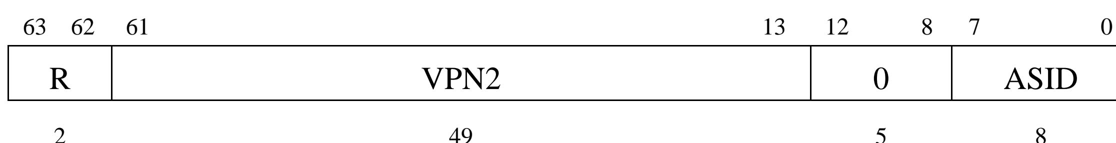


Figure 5-10 EntryHi Register

Table 5-8 EntryHi Register Fields

Field	Description
VPN2	Virtual page number divided by two (maps to two pages); upper bits of the virtual address
ASID	Address space ID field. An 8-bit field that lets multiple processes share the TLB; each process has a distinct mapping of otherwise identical virtual page numbers.
R	Region. (00 → user, 01 → supervisor, 11 → kernel) used to match vAddr63...62
0	Reserved. Must be written as zeroes, and returns zeroes when read.

In 64-bit addressing mode, the *VPN2* field contains bits 43:13 of the 44-bit virtual address.

In 32-bit addressing mode only the lower 32 bits of the *EntryHi* register are used, so the format remains the same as in the R4400 processor's 32-bit addressing mode. The *FILL* field is ignored on write and read as zeroes, as it was in the R4400 implementation.

When either a TLB refill, TLB invalid, or TLB modified exception occurs, the *EntryHi* register is loaded with the virtual page number (VPN2) and the ASID of the virtual address that did not have a matching TLB entry.

5.10 Status Register (12)

The *Status* register (SR) is a read/write register that contains the operating mode, interrupt enabling, and the diagnostic states of the processor. The following list

describes the more important *Status* register fields;Figure 5-11 shows the format of the entire register, including descriptions of the fields. Some of the important fields include:

• The 8-bit *Interrupt Mask* (*IM*) field controls the enabling of eight interrupt conditions. Interrupts must be enabled before they can be asserted, and the corresponding bits are set in both the *Interrupt Mask* field of the *Status* register and the *Interrupt Pending* field of the *Cause* register. For more information, refer to the *Interrupt Pending* (*IP*) field of the *Cause* register.

• The 4-bit *Coprocessor Usability* (*CU*) field controls the usability of 4 possible coprocessors. Regardless of the *CU0* bit setting, CP0 is always usable in Kernel mode.

Status Register

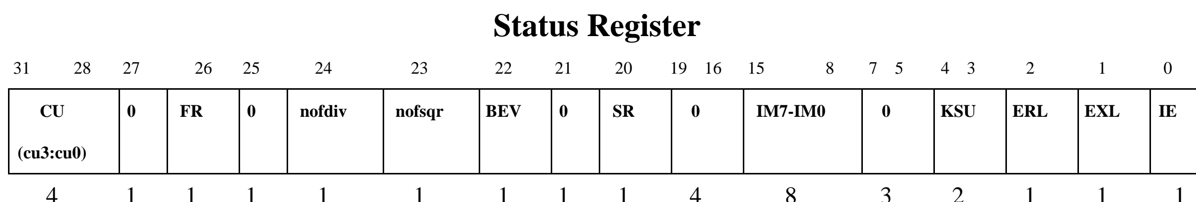


Figure 5-11 Status Register

Status Register Format

Figure 5-11 shows the format of the *Status* register.Table 5-9 describes the *Status* register fields.

Table 5-9 Fields in the Status Register

Field	Description
CU	Controls the usability of each of the four coprocessor units. CP0 is always usable when in Kernel mode, regardless of the setting of the *CU0* bit. 1 → usable 0 → unusable
0	Reserved 0.
FR	Enables additional floating-point registers 0 → 16 registers 1 → 32 registers
NOFDIV	Disable the floating-point division unit 1 - disable 0 - enable
NOFSQR	Disable the floating-point square-root unit 1 - disable 0 - enable
BEV	Controls the location of TLB refill and general exception vectors. 0 →normal

Field	Description
	1→bootstrap
SR	1→ Indicates a **Reset*** signal or NMI has caused a Soft Reset exception
IM	*Interrupt Mask*: controls the enabling of each of the external, internal, and software interrupts. An interrupt is taken if interrupts are enabled, and the corresponding bit is set in both the *Interrupt Mask* field of the *Status* register and the *Interrupt Pending* field of the *Cause* register. 0 → disabled 1→ enabled
KSU	Mode bits 11_2 → Undefined 10_2 → User 01_2 → Supervisor 00_2 → Kernel
ERL	Error Level; set by the processor when Reset, Soft Reset, NMI, or Cache Error exception are taken. 0 → normal 1 → error
EXL	Exception Level; set by the processor when any exception other than Reset, Soft Reset, NMI, or Cache Error exception are taken. 0 → normal 1 → exception
IE	Interrupt Enable 0 → disable all interrupts 1 → enables all interrupts

Status Register Modes and Access States

Fields of the *Status* register set the modes and access states described in the sections that follow.

Interrupt Enable: Interrupts are enabled when all of the following conditions are true:

- *IE* = 1
- *EXL* = 0
- *ERL* = 0

If these conditions are met, the settings of the *IM* bits enable the interrupt.

Operating Modes: The following CPU *Status* register bit settings are required for User, Kernel, and Supervisor modes.

• The processor is in User mode when *KSU* = 10_2, *EXL* = 0, and *ERL* = 0.

• The processor is in Supervisor mode when *KSU* = 01_2, *EXL* = 0, and *ERL* = 0.

• The processor is in Kernel mode when *KSU* = 00_2, or *EXL* = 1, or *ERL* = 1.

32- and 64-bit Modes: Godson-2E runs at 64-bit mode.

Kernel Address Space Accesses: Access to the kernel address space is allowed when the processor is in Kernel mode.

Supervisor Address Space Accesses: Access to the supervisor address space is allowed when the processor is in Kernel or Supervisor mode, as described above in the section titled Operating Modes.

User Address Space Accesses: Access to the user address space is allowed in any of the three operating modes.

Status Register Reset: The contents of the *Status* register are 0x30400004 at reset.

5.11 Cause Register(13)

The 32-bit read/write *Cause* register describes the cause of the most recent exception.

Figure 5-12 shows the fields of this register;Table 5-10 describes the *Cause* register fields. A 5-bit exception code (*ExcCode*) indicates one of the causes, as listed in Table 5-11.

All bits in the *Cause* register, with the exception of the *IP[1:0]* bits, are read-only; *IP[1:0]* are used for software interrupts.

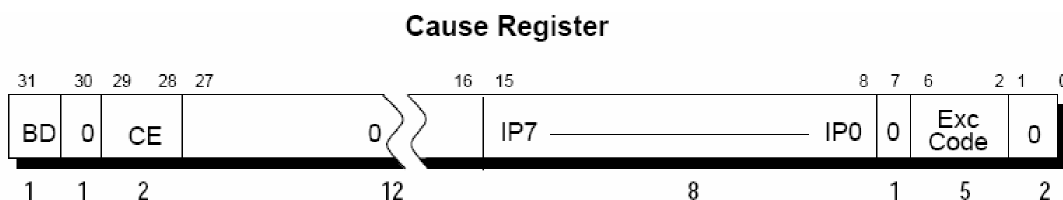


Figure 5-12 Cause Register Format

Table 5-10 Fields in the Cause Register

Field	Descriptions
BD	Indicates whether the last exception taken occurred in a branch delay slot. 1 → delay slot 0 → normal

CE	Coprocessor unit number referenced when a Coprocessor Unusable exception is taken. This bit is undefined for any other exception.
IP	Indicates an interrupt is pending. This bit remains unchanged for NMI, Soft Reset, and Cache Error exceptions. 1 → interrupt pending 0 → no interrupt
ExcCode	Exception code field (see Table 5-11)
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Table 5-11 Cause Register ExcCode Field

Exception Code Value	Mnemonic	Description
0	Int	Interrupt
1	Mod	TLB modification exception
2	TLBL	TLB exception (load or instruction fetch)
3	TLBS	TLB exception (store)
4	AdEL	Address error exception (load or instruction fetch)
5	AdES	Address error exception (store)
6	IBE	Bus error exception (instruction fetch)
7	DBE	Bus error exception (data reference: load or store)
8	Sys	Syscall exception
9	Bp	Breakpoint exception
10	RI	Reserved instruction exception
11	CpU	Coprocessor Unusable exception
12	Ov	Arithmetic Overflow exception
13	Tr	Trap exception
14	-	Reserved
15	FPE	Floating-Point exception
16—22	-	Reserved
23	WATCH	Reference to *WatchHi*/*WatchLo* address
24—30	-	Reserved
31	-	Reserved

5.12 Exception Program Counter (14)

The Exception Program Counter (*EPC*)† is a read/write register that contains the address at which processing resumes after an exception has been serviced.

For synchronous exceptions, the *EPC* register contains either:

• the virtual address of the instruction that was the direct cause of the exception, or

• the virtual address of the immediately preceding branch or jump instruction (when the instruction is in a branch delay slot, and the *Branch Delay* bit in the *Cause* register is set).

The processor does not write to the *EPC* register when the *EXL* bit in the *Status* register is set to 1.

Figure 5-13 shows the format of the *EPC* register.

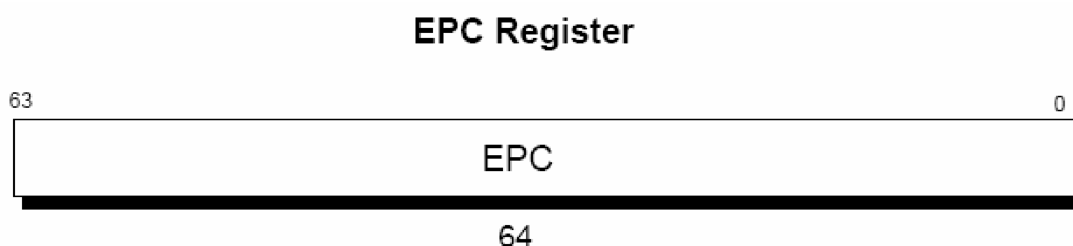


Figure 5-13 EPC Register Format

5.13 Processor Revision Identifier (PRID) Register

The 32-bit, read-only *Processor Revision Identifier* (*PRId*) register contains information identifying the implementation and revision level of the CPU and CP0. Figure 5-14 shows the format of the *PRId* register;Table 5-12 describes the *PRId* register fields.

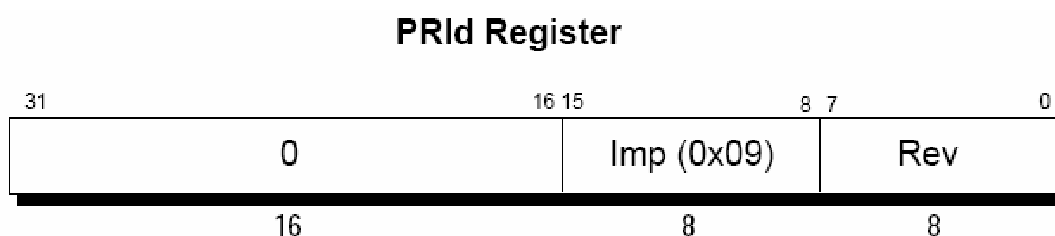


Figure 5-14 Processor Revision Identifier Register Format

Table 5-12 PRId Register Fields

Field	Description
Imp	Implementation number
Rev	Revision number
0	Reserved. Must be written as zeroes, and returns zeroes when read.

The low-order byte (bits 7:0) of the *PRId* register is interpreted as a revision number, and the high-order byte (bits 15:8) is interpreted as an implementation number. The implementation number of the Godson-2E processor is 0x63. The revision number is 0x02. The content of the high-order halfword (bits 31:16) of the

register are reserved.

The revision number can distinguish some chip revisions, however there is no guarantee that changes to the chip will necessarily be reflected in the *PRId* register, or that changes to the revision number necessarily reflect real chip changes. For this reason, software should not rely on the revision number in the *PRId* register to characterize the chip.

5.14 Config Register (16)

The *Config* register specifies various configuration options selected on Godson-2E processors;Table 5-13 lists these options.

Some configuration options, as defined by *Config* bits 31:6, are set by the hardware during reset and are included in the *Config* register as read-only status bits for the software to access. Other configuration options are read/write (as indicated by *Config* register bits 5:0) and controlled by software; on reset these fields are undefined.

Certain configurations have restrictions. The *Config* register should be initialized by software before caches are used. Caches should be written back to memory before line sizes are changed, and caches should be reinitialized after any change is made.

Figure 5-15 shows the format of the *Config* register;Table 5-13 describes the *Config* register fields.

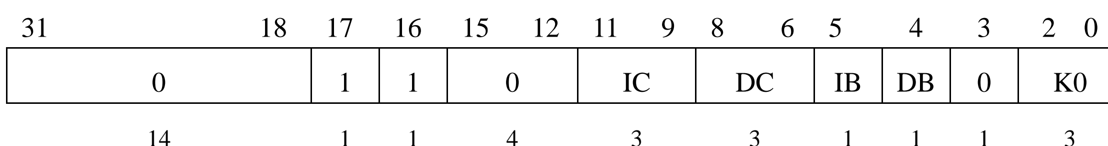


Figure 5-15 Config Register Format

Table 5-13 Fields in the Config Register

Field	Descriptions
0	Reserved. Must be written as zeroes, returns zeroes when read.
1	Reserved. Must be written as ones, returns ones when read.
IC	Primary I-cache Size (I-cache size = 2^{12+IC} bytes).
DC	Primary D-cache Size (D-cache size = 2^{12+DC} bytes).
IB	Primary I-cache line size 0 → 16 bytes 1 → 32 bytes In Godson-2E, this bit is set to 1.
DB	Primary D-cache line size 0 → 16 bytes

Field	Descriptions
	1 → 32 bytes In Godson-2E, this bit is set to 1.
K0	*kseg0* coherence algorithm.

5.15 Load Linked Address (LLAddr) Register (17)

The read/write *Load Linked Address* (*LLAddr*) register contains the physical address read by the most recent Load Linked instruction. It is not defined in Godson-2E.

5.16 Watch Register

The *Watch* register is a 64-bit read/write register which contains a virtual address of a doubleword in the virtual memory. If enabled, any attempt to read or write at this location causes a Watch exception. This feature is used for debugging.

Figure 5-16 describes the format of the Watch register.Table 5-14 describes the fields of the Watch register.

Figure 5-16 Watch Register Formats

Table 5-14 Watch Register Fields

Field	Description
VADDR	Bits 63:3 of the virtual address
R	Trap on load references if set to 1
W	Trap on store references if set to 1
0	Reserved. Must be written as zeroes, and returns zeroes when read.

5.17 Xcontext Register(20)

The read/write *XContext* register contains a pointer to an entry in the page table entry (PTE) array, an operating system data structure that stores virtual-to-physical address translations. When there is a TLB miss, the operating system software loads the TLB with the missing translation from the PTE array. The *XContext* register no longer shares the information provided in the *BadVAddr* register, as it did in the R4400.

The *XContext* register is for use with the XTLB refill handler, which loads TLB entries for references to a 64-bit address space, and is included solely for operating system use. The operating system sets the PTE base field in the register, as needed. Normally, the operating system uses the *Context* register to address the current page map, which resides in the kernel-mapped segment *kseg3*.

Figure 5-17 shows the format of the *XContext* register;Table 5-15 describes the *XContext* register fields.

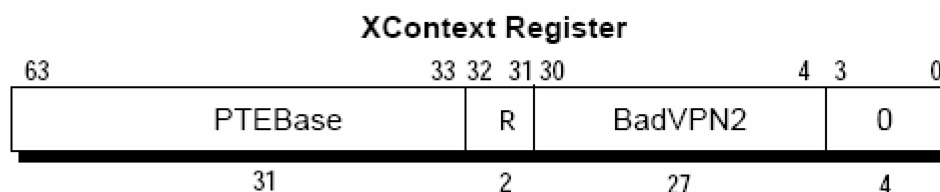


Figure 5-17 XContext Register Format

The 31-bit *BadVPN2* field holds bits 43:13 of the virtual address that caused the TLB miss; bit 12 is excluded because a single TLB entry maps to an even-odd page pair. For a 4-Kbyte page size, this format may be used directly to address the pairtable of 8-byte PTEs. For other page and PTE sizes, shifting and masking this value produces the appropriate address.

Table 5-15 Fields in the XContext Register

Field	Description
BadVPN2	The *Bad Virtual Page Number*/2 field is written by hardware on a miss. It contains the VPN of the most recent invalidly translated virtual address.
R	The *Region* field contains bits 63:62 of the virtual address. 00_2 = user 01_2 = supervisor 11_2 = kernel.
0	Reserved. Must be written as zeroes, and returns zeroes when read.
PTEBase	The *Page Table Entry Base* read/write field is normally written with a value that allows the operating system to use the *Context* register as a pointer into the current PTE array in memory.

5.18 Diagnostic Register(22)

CP0 register 22, the *Diagnostic* register, is a new 64-bit register for Godson-2E

specific diagnostic functions. (Since this register is designed for local use, the diagnostic functions are subject to change without notice.) Currently, this register helps handle the ITLB, BTB(branch target buffer) and RAS(return address stack).

Diagnostic Register

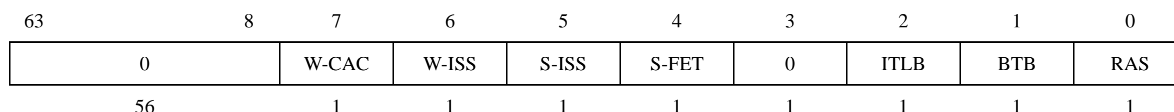


Figure 5-18 Diagnostic Register

Table 5-16 Diagnostic Register Fields

Field	Description
0	Reserved. Must be written as zeroes, and returns zeroes when read.
W-CAC	cancel the constraint on wait-cache operation
W-ISS	cancel the constraint on wait-issue operation
S-ISS	cancel the constraint on store-issue operation
S-FET	cancel the constraint on store-fetch operation
ITLB	write 1 to this bit to clear the ITLB
BTB	write 1 to this bit to clear the BTB
RAS	write 1 to this bit to disable the RAS

5.19 Performance Counter Registers (24,25)

The Godson-2E processor defines two performance counters, which are mapped into CP0 register 24 and 25. The associated control fields reside in CP0 register 24.

Each counter is a 32-bit read/write register and increments by one each time the countable event, specified in its associated control field, occurs. Each counter can independently count one type of event at a time.

Performance Counter Register 24

63 13	12 9	8 5	4	3	2	1	0
0	Event1	Event0	IE	U	S	K	EXL
51	1	1	1	1	1	1	1

Performance Counter Register 25

63 32	31 0
Counter1	Counter0
32	32

Figure 5-19 Performance Counter Registers Format

The counter asserts an interrupt, *IP[7]*, when its most significant bit (bit 31)

becomes one (the counter overflows) and the associated control field enables the interrupt . The counting continues after counter overflows whether or not an interrupt is signalled.

The format of the control registers are shown inFigure 5-19.Table 5-17 describes control fields format and Table 5-18 describes count enable bit definition. Table 5-19 and Table 5-20 describe events of counter 0 and counter 1 respectively.

Table 5-17 Control Fields Format

[12:9]	[8:5]	[4]	[3:0]
Event 1 select	Event 0 select	IP[7] interrupt enable	Count enable bits (K/S/U/EXL)

Table 5-18 Count Enable Bit Definition

Count Enable Bit	Count Qualifier(CP0 Status Register Fields)
K	KSU = 0 (Kernel mode), EXL = 0, ERL = 0
S	KSU = 1 (Supervisor mode), EXL = 0, ERL = 0
U	KSU = 2 (User mode), EXL = 0, ERL = 0
EXL	EXL = 1, ERL = 0

Table 5-19 Counter 0 Events

Event	Signal	Description
0000	Cycles	cycles
0001	Brbus.valid	Branch instruction
0010	Jrcount	JR instruction
0011	Jr31count	JR instruction with field rs=31
0100	Imemread.valid& imemread_allow	Primary instruction cache misses.
0101	Rissuebus0.valid	Alu1 op issued
0110	Rissuebus2.valid	Mem op issued
0111	Rissuebus3.valid	Falu1 op issued
1000	Cp0fwd.valid	CP0 queue forward loads
1001	Mreadreq.valid& Mreadreq_allow	Reads from main memory
1010	Fxqfull	Times of fix issue queue full
1011	Roqfull	Times of reorder queue full
1100	Cp0qfull	Times of CP0 queue full
1101	Exbus.ex & excode=34,35	Tlb Refill exception
1110	Exbus.ex &	Interrupt

Event	Signal	Description
	Excode=0	
1111	Exbus.ex & Excode=63	Internal Exception

Table 5-20 Counter 1 Events

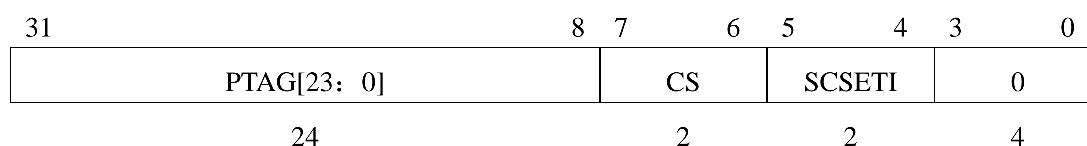
Event	Signal	Description
0000	Cmtbus?.valid	Commit ops
0001	Brbus.brerr	Branch Misprediction
0010	Jrmiss	JR Misprediction
0011	Jr31miss	JR with rs=31 Misprediction
0100	Dmemread.valid& Dmemread_allow	Primary Data cache misses
0101	Rissuebus1.valid	Alu2 op issued
0110	Rissuebus4.valid	Falu2 op issued
0111	Duncache_valid& Duncache_allow	Uncached Accesses
1000	Dmemref.op=store	Store ops
1001	Mwritereq.valid& Mwritereq_allow	Writes to main memory
1010	Ftqfull	Times of float pointer queue full
1011	Brqfull	Times of branch queue full
1100	Exbus.ex & Op==OP_TLBPI	Itlb misses
1101	Exbus.ex	Total exceptions
1110	Mispec	Load speculation misses
1111		

5.20 TagLo (28) and TagHi (29) Registers

The *TagLo* and *TagHi* registers are 32-bit read/write registers that hold the tag and state of primary cache or secondary cache. The *Tag* registers are written by the CACHE and MTC0 instructions.

Figure 5-20 shows the format of these registers for primary cache operations. Table 5-21 lists the field definitions of the *TagLo* and *TagHi* registers.

TagLo Register



TagHi Register

Figure 5-20 TagLo and TagHi Register (P-cache) Formats

Table 5-21 Cache Tag Register Fields

Field	Description
PTAG	Specifies the physical address bits 39:12
CS	Specifies the primary cache state
SCSETI	Specifies the set of secondary cache
0	Reserved. Must be written as zeroes, and returns zeroes when read.

5.21 ErrorEPC Register(30)

The *ErrorEPC* register is similar to the *EPC* register, except that *ErrorEPC* is used on ECC and parity error exceptions. It is also used to store the program counter (PC) on Reset, Soft Reset, and nonmaskable interrupt (NMI) exceptions.

The read/write *ErrorEPC* register contains the virtual address at which instruction processing can resume after servicing an error.Figure 5-21 shows the format of the *ErrorEPC* register.

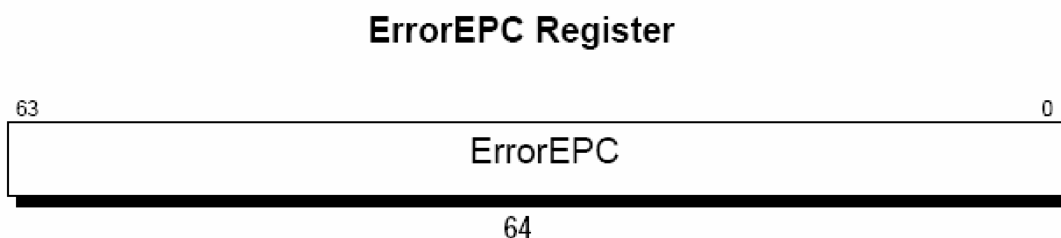


Figure 5-21 ErrorEPC Register Format

5.22 CP0 Instructions

Table 5-22 lists the CP0 instructions defined for the Godson-2E processor. Since they are implementation dependent, they are included here and not in the MIPS ISA manual.

Table 5-22 CP0 Instructions

OpCode	Description

CACHE	Cache Operation
DMFC0	Doubleword Move From CP0
DMTC0	Doubleword Move To CP0
ERET	Exception Return
MFC0	Move From CP0
MTC0	Move To CP0
TLBP	Prove TLB for Matching Entry
TLBR	Read Indexed TLB Entry
TLBWI	Write Indexed TLB Entry
TLBWR	Write Random TLB Entry

Hazards

The processor detects most of the pipeline hazards in hardware, including CP0 hazards and load hazards. No NOP instructions are required to correct instruction sequences.

6 CPU Exceptions

This chapter describes the processor exceptions—a general view of the cause and return of an exception, exception vector locations, and the types of exceptions that are supported, including the cause, processing, and servicing of each exception.

6.1 Causing and Returning from an Exceptions

When the processor takes an exception, the *EXL* bit in the *Status* register is set to 1, which means the system is in Kernel mode. After saving the appropriate state, the exception handler typically changes the *KSU* bits in the *Status* register to Kernel mode and resets the *EXL* bit back to 0. When restoring the state and restarting, the handler restores the previous value of the *KSU* field and sets the *EXL* bit back to 1.

Returning from an exception also resets the *EXL* bit to 0 (see the ERET instruction in Appendix A).

6.2 Exception Vector Locations

The Cold Reset, Soft Reset, and NMI exceptions are always vectored to the dedicated Cold Reset exception vector at an uncached and unmapped address. Addresses for all other exceptions are a combination of a *vector offset* and a *base address*.

The boot-time vectors (when *BEV* = 1 in the *Status* register) are at uncached and unmapped addresses. During normal operation (when *BEV* = 0) the regular exceptions have vectors in cached address spaces; Cache Error is always at an uncached address so that cache error handling can bypass a suspect cache.

The exception vector assignments for the Godson-2E processor shown in Table 6-1。

Table 6-1 Exception Vector Addresses

BEV	Exception Type	Exception Vector Address
	Cold Reset/Soft Reset/ NMI	0xFFFFFFFF BFC00000
BEV = 0	TLB Refill (EXL=0)	0xFFFFFFFF 80000000
	XTLB Refill (EXL=0)	0xFFFFFFFF 80000000
	Cache Error	0xFFFFFFFF A0000100
	Others	0xFFFFFFFF 80000180
BEV = 1	TLB Refill (EXL=0)	0xFFFFFFFF BFC00200
	XTLB Refill (EXL=0)	0xFFFFFFFF BFC00200
	Cache Error	0xFFFFFFFF BFC00300
	Others	0xFFFFFFFF BFC00380

6.3 TLB Refill Vector Selection

In all present implementations of the MIPS III ISA, there are two TLB refill exception vectors:

• one for references to 32-bit address space (TLB Refill)

• one for references to 64-bit address space (XTLB Refill)

Table 6-1 lists the exception vector addresses.

The TLB refill vector selection is based on the address space of the address (*user*, *supervisor*, or *kernel*) that caused the TLB miss, and the value of the corresponding extended addressing bit in the *Status* register (*UX*, *SX*, or *KX*). The current operating mode of the processor is not important except that it plays a part in specifying in which address space an address resides. The *Context* and *Xcontext* registers are entirely separate page-table-pointer registers that point to and refill from two separate page tables, however these two registers share *BadVPN2* fields (see Chapter 6 for more information). For all TLB exceptions (Refill, Invalid, TLBL or TLBS), the *BadVPN2* fields of both registers are loaded as they were in the R4400.

In contrast to the R10000, the R4400 processor selects the vector based on the current operating mode of the processor (*user*, *supervisor*, or *kernel*) and the value of the corresponding extended addressing bit in the *Status* register (*UX*, *SX* or *KX*). In addition, the *Context* and *XContext* registers are not implemented as entirely separate registers; the *PTEbase* fields are shared. A miss to a particular address goes through either TLB Refill or XTLB Refill, depending on the source of the reference. There can be only be a single page table unless the refill handlers execute address-deciphering and page table selection in software.

NOTE: Refills for the 0.5 Gbyte supervisor mapped region, *sseg/ksseg*, are controlled by the value of *KX* rather than *SX*. This simplifies control of the processor when supervisor mode is not being used.

6.4 Priority of Exceptions

The remainder of this chapter describes exceptions in the order of their priority shown in Table 6-2 (with certain of the exceptions, such as the TLB exceptions and Instruction/Data exceptions, grouped together for convenience). While more than one exception can occur for a single instruction, only the exception with the highest priority is reported. Some exceptions are not caused by the instruction executed at the

time, and some exceptions may be deferred. See the individual description of each exception in this chapter for more detail.

Table 6-2 Exception Priority Order

Exception Priority Order
Cold Reset (highest priority)
Soft Reset
Nonmaskable Interrupt (NMI)‡
Cache error — Instruction cache* Cache error — Data cache* Cache error — Secondary cache* Cache error — System interface*
Address error — Instruction fetch
TLB refill — Instruction fetch
TLB invalid — Instruction fetch
Bus error — Instruction fetch
Integer overflow, Trap, System Call, Breakpoint, Reserved Instruction, Coprocessor Unusable, or Floating-Point Exception
Address error — Data access
TLB refill — Data access
TLB invalid — Data access
TLB modified — Data write
Watch*
Bus error — Data access
Interrupt (lowest priority)*

Generally speaking, the exceptions described in the following sections are handled (“processed”) by hardware; these exceptions are then serviced by software.

6.5 Cold Reset Exception

Cause

The Cold Reset exception is taken for a power-on or “cold” reset; it occurs when the **SysGnt*** signal is asserted while the **SysReset*** signal is also asserted.† This exception is not maskable.

Processing

The CPU provides a special interrupt vector for this exception:

- location 0xBFC0 0000 in 32-bit mode
- location 0xFFFF FFFF BFC0 0000 in 64-bit mode

The Cold Reset vector resides in unmapped and uncached CPU address space, so the hardware need not initialize the TLB or the cache to process this exception. It also means the processor can fetch and execute instructions while the caches and virtual memory are in an undefined state.

The contents of all registers in the CPU are undefined when this exception occurs, except for the following register fields:

• In the *Status* register, *SR* and *TS* are cleared to 0, and *ERL* and *BEV* are set to 1. All other bits are undefined.

• *Config* register is initialized with the boot mode bits read from the serial input.

• The *Random* register is initialized to the value of its upper bound.

• The *Wired* register is initialized to 0.

• The *EW* bit in the *CacheErr* register is cleared.

• The *ErrorEPC* register gets the PC.

• The *FrameMask* register is set to 0.

• Branch prediction bits are set to 0.

• *Performance Counter* register *Event* field is set to 0.

• All pending cache errors, delayed watch exceptions, and external interrupts are cleared.

Servicing

The Cold Reset exception is serviced by:

• initializing all processor registers, coprocessor registers, caches, and the memory system

• performing diagnostic tests

• bootstrapping the operating system

6.6 Soft Reset Exception

Cause

The Soft Reset exception occurs in response to a Soft Reset.

A Soft Reset exception is not maskable.

The processor differentiates between a Cold Reset and a Soft Reset as follows:

• A Cold Reset occurs when the **SysGnt*** signal is asserted while the **SysReset*** signal is also asserted.

• A Soft Reset occurs if the **SysGnt*** signal remains negated when a **SysReset***

signal is asserted.

In Godson-2E processor, there is no way for software to differentiate between a Soft Reset exception and an NMI exception.

Processing

When a Soft Reset exception occurs, the *SR* bit of the *Status* register is set, distinguishing this exception from a Cold Reset exception.

When a Soft Reset is detected, the processor initializes minimum processor state. This allows the processor to fetch and execute the instructions of the exception handler, which in turn dumps the current architectural state to external logic. Hardware state that loses architectural state is not initialized unless it is necessary to execute instructions from unmapped uncached space that reads the registers, TLB, and cache contents.

The Soft Reset can begin on an arbitrary cycle boundary and can abort multicycle operations in progress, so it may alter machine state. Hence, caches, memory, or other processor states can be inconsistent: data cache blocks may stay at the refill state and any cached loads/stores to these blocks will hang the processor. Therefore, CacheOps should be used to dump the cache contents.

After the processor state is read out, the processor should be reset with a Cold Reset sequence.

A Soft Reset exception preserves the contents of all registers, except for:

- *ErrorEPC* register, which contains the PC
- *ERL* bit of the *Status* register, which is set to 1
- *SR* bit of the *Status* register, which is set to 1 on Soft Reset or an NMI; 0 for a Cold Reset
- *BEV* bit of the *Status* register, which is set to 1
- *TS* bit of the *Status* register, which is set to 0
- PC is set to the reset vector 0xFFFF FFFF BFC0 0000
- clears any pending Cache Error exceptions

Servicing

A Soft Reset exception is intended to quickly reinitialize a previously operating processor after a fatal error.

It is not normally possible to continue program execution after returning from

this exception, since a **SysReset*** signal can be accepted anytime.

6.7 NMI Exception

Cause

The NMI exception is caused by assertion of the **SysNMI*** signal.

An NMI exception is not maskable.

In Godson-2E processor, there is no way for software to differentiate between a Soft Reset exception and an NMI exception.

Processing

When an NMI exception occurs, the *SR* bit of the *Status* register is set, distinguishing this exception from a Cold Reset exception.

An exception caused by an NMI is taken at the instruction boundary. It does not abort any state machines, preserving the state of the processor for diagnosis. The *Cause* register remains unchanged and the system jumps to the NMI exception handler (see Table 6-1).

An NMI exception preserves the contents of all registers, except for:

- *ErrorEPC* register, which contains the PC
- *ERL* bit of the *Status* register, which is set to 1
- *SR* bit of the *Status* register, which is set to 1 on Soft Reset or an NMI; 0 for a Cold Reset
- *BEV* bit of the *Status* register, which is set to 1
- *TS* bit of the *Status* register, which is set to 0
- PC is set to the reset vector 0xFFFF FFFF BFC0 0000
- clears any pending Cache Error exceptions

Servicing

The NMI can be used for purposes other than resetting the processor while preserving cache and memory contents. For example, the system might use an NMI to cause an immediate, controlled shutdown when it detects an impending power failure.

It is not normally possible to continue program execution after returning from this exception, since an NMI can occur during another error exception.

6.8 Address Error Exception

Cause

The Address Error exception occurs when an attempt is made to execute one of the following:

• reference to an illegal address space

• reference the supervisor address space from User mode

• reference the kernel address space from User or Supervisor mode

• load or store a doubleword that is not aligned on a doubleword boundary

• load, fetch, or store a word that is not aligned on a word boundary

• load or store a halfword that is not aligned on a halfword boundary

This exception is not maskable.

Processing

The common exception vector is used for this exception. The *AdEL* or *AdES* code in the *Cause* register is set, indicating whether the instruction caused the exception with an instruction reference, load operation, or store operation shown by the *EPC* register and *BD* bit in the *Cause* register.

When this exception occurs, the *BadVAddr* register retains the virtual address that was not properly aligned or that referenced protected address space. The contents of the *VPN* field of the *Context*, *XContext*, and *EntryHi* registers are undefined, as are the contents of the *EntryLo* register.

The *EPC* register contains the address of the instruction that caused the exception, unless this instruction is in a branch delay slot. If it is in a branch delay slot, the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set as indication.

Servicing

The process executing at the time is handed a UNIX SIGSEGV (segmentation violation) signal. This error is usually fatal to the process incurring the exception.

6.9 TLB Exceptions

Three types of TLB exceptions can occur:

• TLB Refill occurs when there is no TLB entry that matches an attempted

reference to a mapped address space.

• TLB Invalid occurs when a virtual address reference matches a TLB entry that is marked invalid.

• TLB Modified occurs when a store operation virtual address reference to memory matches a TLB entry which is marked valid but is not dirty (the entry is not writable).

The following three sections describe these TLB exceptions.

NOTE: TLB Refill vector selection is also described earlier in this chapter, in the section titled, TLB Refill Vector Selection.

6.10 TLB Refill Exceptions

Cause

The TLB refill exception occurs when there is no TLB entry to match a reference to a mapped address space. This exception is not maskable.

Processing

There are two special exception vectors for this exception; one for references to 32-bit address spaces, and one for references to 64-bit address spaces. The *UX, SX,*and *KX* bits of the *Status* register determine whether the user, supervisor or kernel address spaces referenced are 32-bit or 64-bit spaces; the TLB refill vector is selected based upon the address space of the address causing the TLB miss (user, supervisor, or kernel mode address space), together with the value of the corresponding extended addressing bit in the *Status* register (*UX*, *SX*, or *KX*). The current operating mode of the processor is not important except that it plays a part in specifying in which space an address resides. An address is in *user* space if it is in *useg*, *suseg*, *kuseg*, *xuseg*, *xsuseg*, or *xkuseg* (see the description of virtual address spaces in Chapter 16). An address is in *supervisor* space if it is in *sseg*, *ksseg*, *xsseg* or *xksseg*, and an address is in *kernel* space if it is in either *kseg3* or *xkseg*. *Kseg0*, *kseg1*, and kernel physical spaces (*xkphys*) are kernel spaces but are not mapped.

All references use these vectors when the *EXL* bit is set to 0 in the *Status* register. This exception sets the *TLBL* or *TLBS* code in the *ExcCode* field of the *Cause* register. This code indicates whether the instruction, as shown by the *EPC* register and the *BD* bit in the *Cause* register, caused the miss by an instruction reference, load operation, or store operation.

When this exception occurs, the *BadVAddr*, *Context*, *XContext* and *EntryHi* registers hold the virtual address that failed address translation. The *EntryHi* register also contains the ASID from which the translation fault occurred. The *Random* register normally contains a valid location in which to place the replacement TLB entry. The contents of the *EntryLo* register are undefined. The *EPC* register contains the address of the instruction that caused the exception, unless this instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set.

Servicing

To service this exception, the contents of the *Context* or *XContext* register are used as a virtual address to fetch memory locations containing the physical page frame and access control bits for a pair of TLB entries. The two entries are placed into the E*ntryLo0/EntryLo1* register; the *EntryHi* and *EntryLo* registers are written into the TLB.

It is possible that the virtual address used to obtain the physical address and access control information is on a page that is not resident in the TLB. This condition is processed by allowing a TLB refill exception in the TLB refill handler. This second exception goes to the common exception vector because the *EXL* bit of the *Status* register is set.

6.11 TLB Invalid Exception

Cause

The TLB invalid exception occurs when a virtual address reference matches a TLB entry that is marked invalid (TLB valid bit cleared). This exception is not maskable.

Processing

The common exception vector is used for this exception. The *TLBL* or *TLBS* code in the *ExcCode* field of the *Cause* register is set. This indicates whether the instruction, as shown by the *EPC* register and *BD* bit in the *Cause* register, caused the miss by an instruction reference, load operation, or store operation.

When this exception occurs, the *BadVAddr*, *Context*, *XContext* and *EntryHi* registers contain the virtual address that failed address translation. The *EntryHi*

register also contains the ASID from which the translation fault occurred. The *Random* register normally contains a valid location in which to put the replacement TLB entry. The contents of the *EntryLo* registers are undefined.

The *EPC* register contains the address of the instruction that caused the exception unless this instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set.

Servicing

A TLB entry is typically marked invalid when one of the following is true:

• a virtual address does not exist

• the virtual address exists, but is not in main memory (a page fault)

• a trap is desired on any reference to the page (for example, to maintain a reference bit)

After servicing the cause of a TLB Invalid exception, the TLB entry is located with TLBP (TLB Probe), and replaced by an entry with that entry's *Valid* bit set.

6.12 TLB Modified Exception

Cause

The TLB modified exception occurs when a store operation virtual address reference to memory matches a TLB entry that is marked valid but is not dirty and therefore is not writable. This exception is not maskable.

Processing

The common exception vector is used for this exception, and the *Mod* code in the *Cause* register is set.

When this exception occurs, the *BadVAddr*, *Context*, *XContext* and *EntryHi* registers contain the virtual address that failed address translation. The *EntryHi* register also contains the ASID from which the translation fault occurred. The contents of the *EntryLo* register are undefined.

The *EPC* register contains the address of the instruction that caused the exception unless that instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set.

Servicing

The kernel uses the failed virtual address or virtual page number to identify the corresponding access control information. The page identified may or may not permit write accesses; if writes are not permitted, a write protection violation occurs.

If write accesses are permitted, the page frame is marked dirty/writable by the kernel in its own data structures. The TLBP instruction places the index of the TLB entry that must be altered into the *Index* register. The *EntryLo* register is loaded with a word containing the physical page frame and access control bits (with the *D* bit set), and the *EntryHi* and *EntryLo* registers are written into the TLB.

6.13 Bus Error Exception

Cause

A Bus Error exception occurs when a processor block read, upgrade, or double/single/partial-word read request receives an external ERR completion response, or a processor double/single/partial-word read request receives an external ACK completion response where the associated external double/single/partial-word data response contains an uncorrectable error. This exception is not maskable.

Processing

The common interrupt vector is used for a Bus Error exception. The *IBE* or *DBE* code in the *ExcCode* field of the *Cause* register is set, signifying whether the instruction (as indicated by the *EPC* register and *BD* bit in the *Cause* register) caused the exception by an instruction reference, load operation, or store operation.

The *EPC* register contains the address of the instruction that caused the exception, unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set.

Servicing

The physical address at which the fault occurred can be computed from information available in the CP0 registers.

• If the *IBE* code in the *Cause* register is set (indicating an instruction fetch reference), the instruction that caused the exception is located at the virtual address

contained in the *EPC* register (or 4+ the contents of the *EPC* register if the *BD* bit of the *Cause* register is set).

• If the *DBE* code is set (indicating a load or store reference), the instruction that caused the exception is located at the virtual address contained in the *EPC* register (or 4+ the contents of the *EPC* register if the *BD* bit of the *Cause* register is set).

The virtual address of the load and store reference can then be obtained by interpreting the instruction. The physical address can be obtained by using the TLBP instruction and reading the *EntryLo* registers to compute the physical page number. The process executing at the time of this exception is handed a UNIX SIGBUS (bus error) signal, which is usually fatal.

6.14 Integer Overflow Exception

Cause

An Integer Overflow exception occurs when an ADD, ADDI, SUB, DADD, DADDI or DSUB instruction results in a 2's complement overflow. This exception is not maskable.

Processing

The common exception vector is used for this exception, and the *OV* code in the *Cause* register is set.

The *EPC* register contains the address of the instruction that caused the exception unless the instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set.

Servicing

The process executing at the time of the exception is handed a UNIX SIGFPE/ FPE_INTOVF_TRAP (floating-point exception/integer overflow) signal. This error is usually fatal to the current process.

6.15 Trap Exception

Cause

The Trap exception occurs when a TGE, TGEU, TLT, TLTU, TEQ, TNE, TGEI, TGEUI, TLTI, TLTUI, TEQI, or TNEI instruction results in a TRUE condition. This

exception is not maskable.

Processing

The common exception vector is used for this exception, and the *Tr* code in the *Cause* register is set.

The *EPC* register contains the address of the instruction causing the exception unless the instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set.

Servicing

The process executing at the time of a Trap exception is handed a UNIX SIGFPE/FPE_INTOVF_TRAP (floating-point exception/integer overflow) signal. This error is usually fatal.

6.16 System Call Exception

Cause

A System Call exception occurs during an attempt to execute the SYSCALL instruction. This exception is not maskable.

Processing

The common exception vector is used for this exception, and the *Sys* code in the *Cause* register is set.

The *EPC* register contains the address of the SYSCALL instruction unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction.

If the SYSCALL instruction is in a branch delay slot, the *BD* bit of the *Status* register is set; otherwise this bit is cleared.

Servicing

When the System Call exception occurs, control is transferred to the applicable system routine. Additional distinctions can be made by analyzing the *Code* field of the SYSCALL instruction (bits 25:6), and loading the contents of the instruction whose address the EPC register contains.

To resume execution, the *EPC* register must be altered so that the SYSCALL instruction does not re-execute; this is accomplished by adding a value of 4 to the *EPC* register (*EPC* register + 4) before returning.

If a SYSCALL instruction is in a branch delay slot, a more complicated algorithm, beyond the scope of this description, may be required.

6.17 Breakpoint Exception

Cause

A Breakpoint exception occurs when an attempt is made to execute the BREAK instruction. This exception is not maskable.

Processing

The common exception vector is used for this exception, and the *BP* code in the *Cause* register is set.

The *EPC* register contains the address of the BREAK instruction unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction.

If the BREAK instruction is in a branch delay slot, the *BD* bit of the *Status* register is set, otherwise the bit is cleared.

Servicing

When the Breakpoint exception occurs, control is transferred to the applicable system routine. Additional distinctions can be made by analyzing the *Code* field of the BREAK instruction (bits 25:6), and loading the contents of the instruction whose address the EPC register contains. A value of 4 must be added to the contents of the EPC register (EPC register + 4) to locate the instruction if it resides in a branch delay slot.

To resume execution, the EPC register must be altered so that the BREAK instruction does not re-execute; this is accomplished by adding a value of 4 to the EPC register (EPC register + 4) before returning.

If a BREAK instruction is in a branch delay slot, interpretation of the branch instruction is required to resume execution.

6.18 Reserved Instruction Exception

Cause

The Reserved Instruction exception occurs when one of the following conditions occurs:

• an attempt is made to execute an instruction with an undefined major opcode (bits 31:26)

• an attempt is made to execute a SPECIAL instruction with an undefined minor opcode (bits 5:0)

• an attempt is made to execute a REGIMM instruction with an undefined minor opcode (bits 20:16)

• an attempt is made to execute 64-bit operations in 32-bit mode when in User or Supervisor modes

• an attempt is made to execute a COP1X when the MIPS IV ISA is not enabled

64-bit operations are always valid in Kernel mode regardless of the value of the *KX* bit in the *Status* register.

This exception is not maskable.

Processing

The common exception vector is used for this exception, and the *RI* code in the *Cause* register is set.

The *EPC* register contains the address of the reserved instruction unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction.

Servicing

No instructions in the MIPS ISA are currently interpreted. The process executing at the time of this exception is handed a UNIX SIGILL/ILL_RESOP_FAULT (illegal instruction/reserved operand fault) signal. This error is usually fatal.

6.19 Coprocessor Unusable Exception

Cause

The Coprocessor Unusable exception occurs when an attempt is made to execute a coprocessor instruction for either:

• a corresponding coprocessor unit (CP1 or CP2) that has not been marked usable, or

• CP0 instructions, when the unit has not been marked usable and the process executes in either User or Supervisor mode.

This exception is not maskable.

Processing

The common exception vector is used for this exception, and the *CpU* code in the *Cause* register is set. The contents of the *Coprocessor Usage Error* field of the coprocessor *Control* register indicate which of the four coprocessors was referenced. The *EPC* register contains the address of the unusable coprocessor instruction unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction.

Servicing

The coprocessor unit to which an attempted reference was made is identified by the Coprocessor Usage Error field, which results in one of the following situations:

• If the process is entitled access to the coprocessor, the coprocessor is marked usable and the corresponding user state is restored to the coprocessor.

• If the process is entitled access to the coprocessor, but the coprocessor does not exist or has failed, interpretation of the coprocessor instruction is possible.

• If the *BD* bit is set in the *Cause* register, the branch instruction must be interpreted; then the coprocessor instruction can be emulated and execution resumed with the *EPC* register advanced past the coprocessor instruction.

• If the process is not entitled access to the coprocessor, the process executing at the time is handed a UNIX SIGILL/ILL_PRIVIN_FAULT (illegal instruction/privileged instruction fault) signal. This error is usually fatal.

6.20 Floating-Point Exception

Cause

The Floating-Point exception is used by the floating-point coprocessor. This exception is not maskable.

Processing

The common exception vector is used for this exception, and the *FPE* code in the *Cause* register is set.

The contents of the *Floating-Point Control/Status* register indicate the cause of this exception.

Servicing

This exception is cleared by clearing the appropriate bit in the *Floating-Point Control/Status* register.

6.21 Watch Exception

Cause

A Watch exception occurs when a load or store instruction references the physical address specified in the *WatchLo*/*WatchHi* System Control Coprocessor (CP0) registers. The *WatchLo* register specifies whether a load or store initiated this exception.

A Watch exception violates the rules of a precise exception in the following way:

If the load or store reference which triggered the Watch exception has a cacheable address and misses in the data cache, the line will then be read from memory into the secondary cache if necessary, and refilled from the secondary cache into the data cache. In all other cases, cache state is not affected by an instruction which takes a Watch exception.

The CACHE instruction never causes a Watch exception.

The Watch exception is postponed if either the *EXL* or *ERL* bit is set in the *Status* register. If either bit is set, the instruction referencing the *WatchLo*/*WatchHi* address is executed and the exception is delayed until the delay condition is cleared; that is, until *ERL* and *EXL* both are cleared (set to 0). The *EPC* contains the address of the next unexecuted instruction.

A delayed Watch exception is cleared by system reset or by writing a value to the *WatchLo* register.†

Watch is maskable by setting the *EXL* or *ERL* bits in the *Status* register.

Processing

The common exception vector is used for this exception, and the *Watch* code in

the *Cause* register is set.

Servicing

The Watch exception is a debugging aid; typically the exception handler transfers control to a debugger, allowing the user to examine the situation.

To continue program execution, the Watch exception must be disabled to execute the faulting instruction. The Watch exception must then be reenabled. The faulting instruction can be executed either by interpretation or by setting breakpoints.

6.22 Interrupt Exception

Cause

The Interrupt exception occurs when one of the eight interrupt conditions is asserted. The significance of these interrupts is dependent upon the specific system implementation.

Each of the eight interrupts can be masked by clearing the corresponding bit in the *Interrupt-Mask (IM)* field of the *Status* register, and all of the eight interrupts can be masked at once by clearing the *IE* bit of the *Status* register.

Processing

The common exception vector is used for this exception, and the *Int* code in the *Cause* register is set.

The *IP* field of the *Cause* register indicates current interrupt requests. It is possible that more than one of the bits can be simultaneously set (or even *no* bits may be set) if the interrupt is asserted and then deasserted before this register is read.

On Cold Reset, an R4400 processor can be configured with *IP[7]* either as a sixth external interrupt, or as an internal interrupt set when the *Count* register equals the *Compare* register. There is no such option on the R10000 processor; *IP[7]* is always an internal interrupt that is set when one of the following occurs:

- the *Count* register is equal to the *Compare* register
- either one of the two performance counters overflows

Software needs to poll each source to determine the cause of the interrupt (which could come from more than one source at a time). For instance, writing a value to the *Compare* register clears the timer interrupt but it may not clear *IP[7]* if one of the performance counters is simultaneously overflowing. Performance counter interrupts

can be disabled individually without affecting the timer interrupt, but there is no way to disable the timer interrupt without disabling the performance counter interrupt.

Servicing

If the interrupt is caused by one of the two software-generated exceptions (described in Chapter 6, the section titled “Software Interrupts”), the interrupt condition is cleared by setting the corresponding *Cause* register bit, *IP[1:0],* to 0.

Software interrupts are imprecise. Once the software interrupt is enabled, program execution may continue for several instructions before the exception is taken. Timer interrupts are cleared by writing to the *Compare* register. The Performance Counter interrupt is cleared by writing a 0 to bit 31, the overflow bit, of the counter.

Cold Reset and Soft Reset exceptions clear all the outstanding external interrupt requests**,** *IP[2]* to *IP[6]*.

If the interrupt is hardware-generated, the interrupt condition is cleared by correcting the condition causing the interrupt pin to be asserted.

7 Floating-Point Unit

This section describes the floating-point unit (FPU) of the Godson-2E processor, including the programming model, instruction set and formats, instruction pipeline, and exceptions. The FPU, with associated system software, fully conforms to the requirements of ANSI/IEEE Standard 754–1985, IEEE Standard for Binary Floating-Point Arithmetic. In addition, the Godson-2E's FPU can execute SIMD fixed-point multimedia instructions.

7.1 Overview

The FPU operates as a coprocessor for the CPU (it is assigned coprocessor label *CP1*), and extends the CPU instruction set to perform arithmetic operations on floating-point values.

The Floating-Point unit consists of the following functional units:

- **FALU1 unit**
- **FALU2 unit**

The FALU2 unit performs floating-point multiply-add, multiply, addition, subtraction, divide and square-root operations. The FALU1 unit also performs floating-point multiply-add, multiply, addition, subtraction operations and other floating-point operations. In addition, the Godson-2E FPU can perform PS (paired-single) and fixed-point multimedia instructions. Figure 7-1 illustrates the organization of the functional units in Godson-2E's architecture.

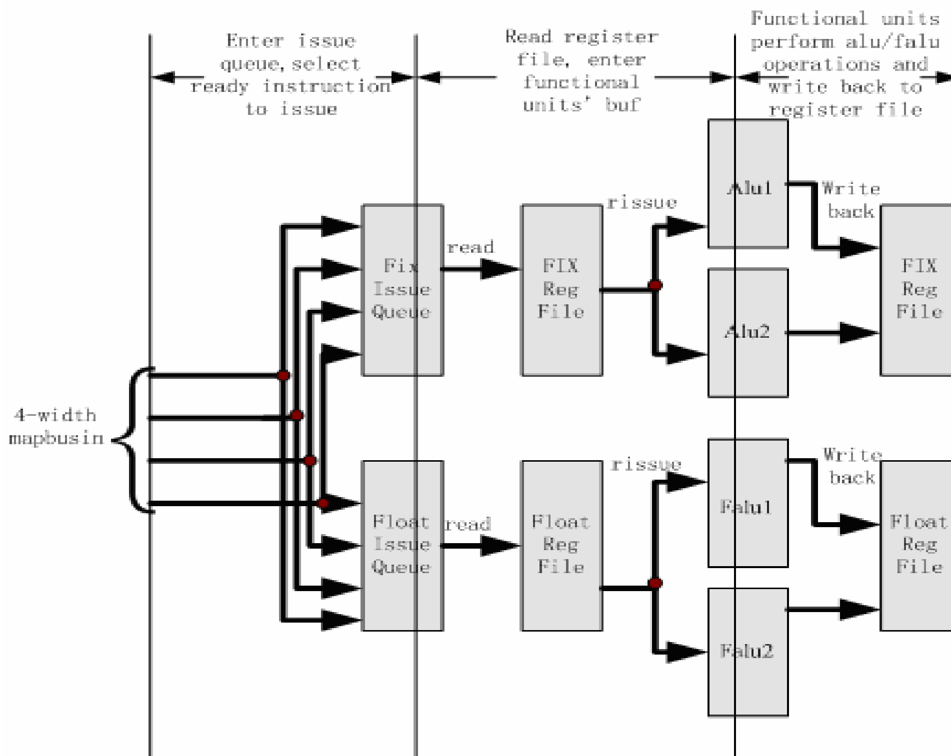


Figure 7-1 The organization of the functional units in Godson-2E's architecture

The floating-point queue can issue one instruction to the FALU1 unit and one instruction to the FALU2 unit each cycle. The FALU1 and FALU2 unit each have three dedicated read ports and one dedicated write port in the floating-point register file.

7.2 FPU Programming Model

This section describes the set of FPU registers and their data organization. The FPU registers include Floating-Point General Purpose registers (FGRs) and two control registers: Control/Status and Implementation/Revision.

7.2.1 Floating-Point Registers

The Floating-Point Unit is the hardware implementation of Coprocessor 1 in the MIPS IV Instruction Set Architecture. The MIPS IV ISA defines 32 logical floating-point general registers (FGRs), Each FGR is 64 bits wide and can hold either 32-bit single-precision or 64-bit double-precision values. The hardware actually contains 64 physical 64-bit registers in the Floating-Point Register File, from which the 32 logical registers are taken.

FP instructions use a 5-bit logical number to select an individual FGR. These

logical numbers are mapped to physical registers by the rename unit (regmap), before the Floating-Point Unit executes them. Physical registers are selected using 6-bit addresses.

The FR bit (26) in the Status register determines the number of logical floating-point registers available to the program, and it alters the operation of single precision load/store instructions.

- FR is reset to 0 for compatibility with earlier MIPS I and MIPS II ISAs, and instructions use only the 16 physical even-numbered floating-point registers (32 logical registers). Each logical register is 32 bits wide.
- FR is set to 1 for normal MIPS III and MIPS IV operations, and all 32 of the 64-bit logical registers are available.

7.2.2 Floating-Point Control Registers

The MIPS IV ISA permits up to 32 control registers to be defined for each coprocessor, but the Godson-2E's Floating-Point Unit uses only two:

- Control register 0, the FP Implementation and Revision register
- Control register 31, the Floating-Point Status register (FSR)

The control registers (FCRs) can only be accessed by move operations. The Implementation/Revision register (FCR0) holds revision information about the FPU, and the Control/Status register (FCR31) controls and monitors exceptions, holds the result of compare operations, and establishes rounding modes.

Implementation and Revision Register, (FCR0)

The read-only Implementation and Revision register (FCR0) specifies the implementation and revision number of the FPU. This information can determine the coprocessor revision and performance level, and can also be used by diagnostic software.

Table 7-1 describes the Implementation and Revision register (FCR0) fields.

Table 7-1 FCR0 Fields

Field	Description
Imp[15:8]	Implementation number (0x05)
Rev[7:0]	Revision number in the form of y.x (0x01)
0[31:16]	Reserved. Must be written as zeroes, and returns zeroes when read.

Control/Status Register (FCR31)

The Control/Status register (FCR31) contains control and status information that can be accessed by instructions in either Kernel or User mode. FCR31 also controls the arithmetic rounding mode and enables User mode traps, as well as identifying any exceptions that may have occurred in the most recently executed instruction, along with any exceptions that may have occurred without being trapped.

Figure 7-2 shows the format of the Control/Status register, and Table 7-2 describes the Control/Status register fields.

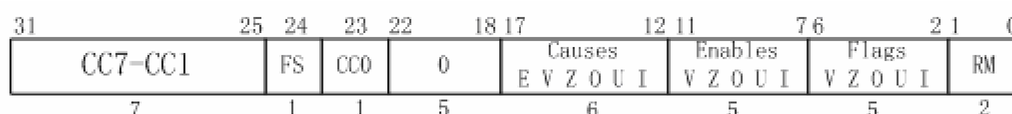


Figure 7-2 FP Control/Status Register Bit Assignments

Table 7-2 Control/Status Register Fields

Field	Description
CC7-CC1	Condition bits 7-1. CC1 is set when the high part of paired-single compare operation is true
FS	When set, denormalized results are flushed to 0 instead of causing an unimplemented operation exception.
CC0	Condition bit. See description of Control/Status register Condition bit.
Cause	Cause bits. See description of Control/Status register Cause bits.
Enables	Enable bits. See description of Control/Status register Enable bits.
Flags	Flag bits. See description of Control/Status register Flag bits.
RM	Rounding mode bits. See description of Control/Status register Rounding Mode Control bits.

Control/Status Register Condition Bit

When a floating-point Compare operation takes place, the result is stored at bit 23, the Condition bit, to save or restore the state of the condition line. The CC0 bit is set to 1 if the condition is true; the bit is cleared to 0 if the condition is false. Bit 23 is affected only by compare and Move Control to FPU instructions.

Control/Status Register Cause Bits

Bits 17:12 in the Control/Status register contain Cause bits, as shown in Figure 7-2, which reflect the results of the most recently executed instruction. The Cause bits are a logical extension of the CP0 Cause register; they identify the exceptions raised by the last floating-point operation and raise an interrupt or exception if the

corresponding enable bit is set. If more than one exception occurs on a single instruction, each appropriate bit is set.

The Cause bits are written by each floating-point operation (but not by load, store, or move operations). The Unimplemented Operation (E) bit is set to a 1 if software emulation is required, otherwise it remains 0. The other bits are set to 0 or 1 to indicate the occurrence or non-occurrence (respectively) of an IEEE 754 exception.

When a floating-point exception is taken, no results are stored, and the only state affected is the Cause bit.

Control/Status Register Enable Bits

A floating-point exception is generated any time a Cause bit and the corresponding Enable bit are set. A floating-point operation that sets an enabled Cause bit forces an immediate exception, as does setting both Cause and Enable bits with CTC1.

There is no enable for Unimplemented Operation (E). Setting Unimplemented Operation always generates a floating-point exception.

Before returning from a floating-point exception, software must first clear the enabled Cause bits with a CTC1 instruction to prevent a repeat of the interrupt. Thus, User mode programs can never observe enabled Cause bits set; if this information is required in a User mode handler, it must be passed somewhere other than the Status register.

For a floating-point operation that sets only unenabled Cause bits, no exception occurs and the default result defined by IEEE 754 is stored. In this case, the exceptions that were caused by the immediately previous floating-point operation can be determined by reading the Cause field.

Control/Status Register Flag Bits

The Flag bits are cumulative and indicate that an exception was raised by an operation that was executed since they were explicitly reset. Flag bits are set to 1 if an IEEE 754 exception is raised, otherwise they remain unchanged. The Flag bits are never cleared as a side effect of floating-point operations; however, they can be set or cleared by writing a new value into the Status register, using a Move To Coprocessor Control instruction.

When a floating-point exception is taken, the flag bits are not set by the hardware;

floating-point exception software is responsible for setting these bits before invoking a user handler.

Control/Status Register Rounding Mode Control Bits

Bits 1 and 0 in the Control/Status register constitute the Rounding Mode (RM) field. As shown in Table 7-3, these bits specify the rounding mode that the FPU uses for all floating-point operations.

Table 7-3 Rounding Mode Bit Decoding

Rounding Mode RM(1:0)	Mnemonic	Description
0	RN	Round result to nearest representable value; round to value with least-significant bit 0 when the two nearest representable values are equally near.
1	RZ	Round toward 0: round to value closest to and not greater in magnitude than the infinitely precise result.
2	RP	Round toward +∞: round to value closest to and not less than the infinitely precise result.
3	RM	Round toward –∞: round to value closest to and not greater than the infinitely precise result.

7.3 FPU Instruction Set Overview

All FPU instructions are 32 bits long, aligned on a word boundary. The Godson-2E FPU not only performs floating-point instructions defined by MIPS standard, but also adds some special instructions like multimedia and PS operations to enhance Godson-2E CPU’s overall performance. These special instructions use same opcode as floating-point instructions but extend the fmt field to define these new instructions. The Godson-2E FPU instruction set can be divided into the following groups according to different formats:

- **Single- or double-precision floating-point instructions (fmt =16, 17)**. These instructions include multiply-add, add，sub，conversion，move，compare and branch instructions. Table 7-4 lists these floating-point instructions.
- **Paired-single (PS) floating-point instructions (fmt =11).** PS instructions can perform a pair of single-precision floating-point operations simultaneously, which include multiply-add, add, sub, mul, abs, neg, move and compare instructions. Table 7-5 lists the details.

- **Multimedia instructions (fmt =12~31).** The media extensions for the Godson-2E Architecture were designed to enhance performance of advanced media and communication applications. The Godson-2E multimedia instructions support parallel operations on byte, half-word, and word data elements, and double-word integer data type.
- **Word or double-word Fixed-point instructions (fmt =12~31).** These instructions are a subset of MIPS fixed-point instructions. They perform fixed-point operations but share floating-point registers and data path. In some sense they can be taken as a part of multimedia instructions.

Table 7-4 floating-point instructions in Godson-2E FPU

MADD	ADD	ROUND.L	MFC1	CVT.S	BC1F	C.F	C.SF
MSUB	SUB	TRUNC.L	MTC1	CVT.D	BC1T	C.UN	C.NGLE
NMADD	MUL	CEIL.L	DMFC1		BC1FL	C.EQ	C.SEQ
NMSUB	DIV	FLOOR.L	DMTC1		BC1TL	C.UEQ	C.NGL
	SQRT	ROUND.W	CFC1	CVT.W		C.OLT	C.LT
	ABS	TRUNC.W	CTC1	CVT.L		C.ULT	C.NGE
	MOV	CEIL.W				C.OLE	C.LE
	NEG	FLOOR.W				C.ULE	C.NGT

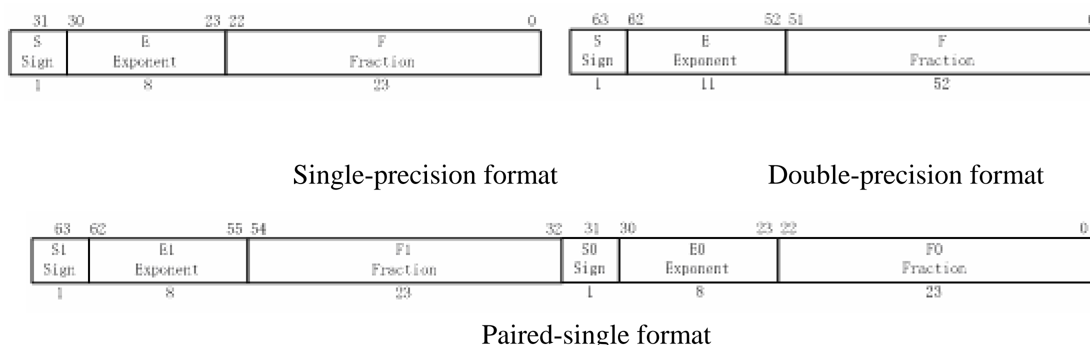
Table 7-5 Paired-single (PS) instructions in Godson-2E FPU

Fmt / OP	Fmt=11
ADD	Add.ps
MADD	MADD.ps
MSUB	MSUB.ps
NMADD	NMADD.ps
NMSUB	NMSUB.ps
SUB	Sub.ps
NEG	Neg.ps
ABS	Abs.ps
C.F	C.F.ps
C.UN	C.UN.ps
C.EQ	C.EQ.ps
C.UEQ	C.UEQ.ps
C.OLT	C.OLT.ps
C.ULT	C.ULT.ps
C.OLE	C.OLE.ps
C.ULE	C.ULE.ps
C.SF	C.SF.ps
C.NGLE	C.NGLE.ps
C.SEQ	C.SEQ.ps
C.NGL	C.NGL.ps
C.LT	C.LT.ps
C.NGE	C.NGE.ps
C.LE	C.LE.ps
C.NGT	C.NGT.ps
MUL	MUL.ps
MOV	MOV.ps

7.4 FPU Formats

7.4.1 Floating-Point Format

The FPU performs both 32-bit (single-precision) and 64-bit (double-precision) IEEE standard floating-point operations. The 32-bit single-precision format has a 24-bit signed-magnitude fraction field (f+s) and an 8-bit exponent (e); The 64-bit double-precision format has a 53-bit signed-magnitude fraction field (f+s) and an 11-bit exponent; The 64-bit paired-single format constraints two single-precision floating-point format. as shown in Figure 7-3 respectively.



Single-precision format Double-precision format

Paired-single format

Figure 7-3 Floating-Point Format

As shown in the above figures, numbers in floating-point format are composed of three fields:

- sign field, s
- biased exponent, e = E + bias
- fraction, $f = .b_1b_2....b_{p-1}$

The range of the unbiased exponent E includes every integer between the two values E_{min} and E_{max} inclusive, together with two other reserved values:

- E_{min} - 1 (to encode ± 0 and denormalized numbers)
- E_{max} +1 (to encode $\pm\infty$ and NaNs [Not a Number])

For single- and double-precision formats, each representable nonzero numerical value has just one encoding. For single- and double-precision formats, the value of a number, v, is determined by the equations shown in Table 7-6.

Table 7-6 Equations for Calculating Values in Single and Double-Precision Floating-Point Format

NO.	Equation
(1)	if $E = E_{max}+1$ and $f\neq0$, then v = NaN, regardless of s
(2)	if $E = E_{max}+1$ and f = 0, then $v= (-1)^s\infty$
(3)	if $E_{min}\leq E\leq E_{max}$, then $v = (-1)^s 2^E(1.f)$
(4)	if $E = E_{min}-1$ and $f\neq0$, then $v= (-1)^s 2^{Emin}(0 .f)$
(5)	if $E = E_{min}-1$ and f = 0, then $v = (-1)^s 0$

For all floating-point formats, if v is NaN, the most-significant bit of f determines whether the value is a signaling or quiet NaN: v is a signaling NaN if the most-significant bit of f is set, otherwise, v is a quiet NaN.

Table 7-7 defines the values for the format parameters; minimum and maximum floating-point values are given in Table 7-8.

Table 7-7 Floating-Point Format Parameter Values

Parameter	Format	
	Single	Double
E_{max}	+127	+1203
E_{min}	-126	-1022
Exponent bias	+127	+1023
Exponent width in bits	8	11
Integer bit	Hidden	Hidden
f (Fraction width in bits)	24	53
Format width in bits	32	64

Table 7-8 Minimum and Maximum Floating-Point Values

Type	Value
Float Minimum	1.40129846e–45
Float Minimum Norm	1.17549435e–38
Float Maximum	3.40282347e+38
Double Minimum	4.9406564584124654e–324
Double Minimum Norm	2.2250738585072014e–308
Double Maximum	1.7976931348623157e+308

7.4.2 Multimedia Format

The Multimedia technology introduces new packed data types, each 64 bits long. The data elements can be:

- eight packed, consecutive 8-bit bytes

- four packed, consecutive 16-bit half-words
- two packed, consecutive 32-bit words
- one 64-bit double-word

The 64 bits are numbered 0 through 63. Bit 0 is the least significant bit (LSB), and bit 63 is the most significant bit (MSB). The low-order bits are the lower part of the data element and the high-order bits are the upper part of the data element. For example, a word contains 16 bits numbered 0 through 15, the byte containing bits 0-7 of the word is called the low byte, and the byte containing bits 8-15 is called the high byte.

The packed integers are held in two formats, unsigned and signed. For example, Figure 7-4 illustrates packed unsigned half-word format and Figure 7-5 illustrates packed signed half-word format.

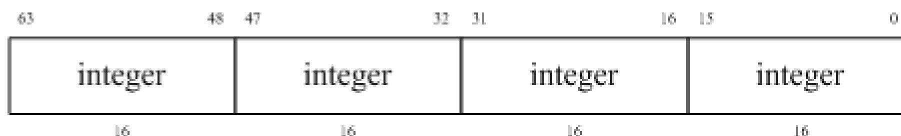


Figure 7-4 packed unsigned half-word format

Figure 7-5 packed signed half-word format

7.5 FPU Instruction Pipeline Overview

The FPU provides an instruction pipeline that parallels the CPU instruction pipeline. It shares the same ten-stage pipeline architecture with the CPU. Each FPU instruction is implemented in one of the two floating-point functional units: FALU1 unit or FALU2 unit. The FALU2 unit performs multiply-add, mul, add, sub, div and sqrt instructions, and the FALU1 performs multiply-add, mul, sub, add and also all other FPU instructions.

Each FALU unit can receive one instruction every cycle, and output one result to the floating-point register file. In FALU1 unit, the floating-point multiply-add, multiply, add, sub operations have 6-cycle execution latency; the conversion between integer and float operations have 4-cycle execution latency; all other operations in FALU1 (include cvt.d.s, cvt.s.d) have 2-cycle execution latency. It means, for

example, if the RAW dependency exists in current floating-point add instruction and next floating-point instruction, the next instruction will wait at least 7 cycles without forward before it can be executed. The FALU1 unit is fully pipelined, so it never need give stall signal to the front pipeline stage. But there is possibility that two instructions with different execution cycles will be output at the same cycle, in this case the instruction whose execution latency is short will be output to the result bus firstly.

The FALU2 unit performs floating-point multiply-add, multiply, add, sub, divide and square-root operations. In FALU2 unit, the floating-point multiply-add, multiply, add, sub operations have 6-cycle execution latency; the floating-point division operation has 4~16 cycle execution latency; the floating-point square root operation has 4~31 cycle execution latency. The floating-point division and square root operations are not pipelined, so if there are two division or sqrt instructions in the FALU2 unit, the FALU2 unit will give a stall signal to the front pipeline stage and can't receive more instructions till the division or sqrt instruction is written back.

7.6 FPU Exceptions

This section describes FPU floating-point exceptions. A floating-point exception occurs whenever the FPU cannot handle either the operands or the results of a floating-point operation in its normal way. The FPU responds by generating an exception to initiate a software trap or by setting a status flag.

The FP Control/Status register contains an Enable bit for each exception type; exception Enable bits determine whether an exception will cause the FPU to initiate a trap or set a status flag.

- If a trap is taken, the FPU remains in the state found at the beginning of the operation and a software exception handling routine executes.
- If no trap is taken, an appropriate value is written into the FPU destination register and execution continues.

The FPU supports the five IEEE Standard 754 exceptions:

- Inexact (I)
- Underflow (U)
- Overflow (O)
- Division by Zero (Z)
- Invalid Operation (V)

The FPU adds a sixth exception type, Unimplemented Operation (E), to use when the FPU cannot implement the standard MIPS floating-point architecture, including cases in which the FPU cannot determine the correct exception behavior. This exception indicates the use of a software implementation. The Unimplemented Operation exception has no Enable or Flag bit; whenever this exception occurs, an unimplemented exception trap is taken (if the FPU interrupt input to the CPU is enabled).

Each of the five IEEE Standard 754 exceptions (V, Z, O, U, I) is associated with a trap under user control, and is enabled by setting one of the five Enable bits. When an exception occurs, the corresponding Cause bit is set. If the corresponding Enable bit is not set, the Flag bit is also set. If the corresponding Enable bit is set, the Flag bit is not set and the FPU generates an interrupt to the CPU. Subsequent exception processing allows a trap to be taken.

When no exception trap is signaled, floating-point coprocessor takes a default action, providing a substitute value for the exception-causing result of the floating-point operation. The particular default action taken depends upon the type of exception. Table 7-9 lists the default action taken by the FPU for each of the IEEE exceptions.

Table 7-9 Default FPU Exception Actions

Field	Description	Rounding Mode	Default action
I	Inexact exception	Any	Supply a rounded result
U	Underflow exception	RN	Modify underflow values to 0 with the sign of the intermediate result
		RZ	Modify underflow values to 0 with the sign of the intermediate result
		RP	Modify positive underflows to the format's smallest positive finite number; modify negative underflows to -0
		RM	Modify negative underflows to the format's smallest negative finite number; modify positive underflows to 0
O	Overflow exception	RN	Modify overflow values to ∞ with the sign of the intermediate result
		RZ	Modify overflow values to the format's largest finite number with the sign of the intermediate result
		RP	Modify negative overflows to the format's most negative finite number; modify positive overflows to +∞
		RM	Modify positive overflows to the format's largest finite number; modify negative overflows to –∞
Z	Division by zero	Any	Supply a properly signed ∞
V	Invalid operation	Any	Supply a quiet Not a Number (NaN)

The following describes the conditions that cause the FPU to generate each of its exceptions, and details the FPU response to each exception-causing condition.

Inexact Exception (I)

The FPU generates the inexact exception if one of the following occurs:

- the rounded result of an operation is not exact, or
- the rounded result of an operation overflows, or the rounded result of an operation underflows and both the Underflow and Inexact Enable bits are not set and the FS bit is set.

Trap Enabled Results: If inexact exception traps are enabled, the result register is not modified and the source registers are preserved. Since this mode of execution can impact performance, inexact exception traps should be enabled only when necessary.

Trap Disabled Results: The rounded or overflowed result is delivered to the destination register if no other software trap occurs.

Invalid Operation Exception (V)

The Invalid Operation exception is signaled if one or both of the operands are invalid for an implemented operation. When the exception occurs without a trap, the MIPS ISA defines the result as a quiet Not a Number (NaN). The invalid operations are:

- Addition or subtraction: magnitude subtraction of infinities, such as: $(+\infty) + (-\infty)$ or $(-\infty) - (-\infty)$
- Multiplication: 0 times ∞, with any signs
- Division: 0/0, or ∞/∞, with any signs
- Comparison of predicates involving<or>without **?**, when the operands are unordered
- Comparison or a Convert From Floating-point Operation on a signaling NaN.
- Any arithmetic operation on a signaling NaN. A move (MOV) operation is not considered to be an arithmetic operation, but absolute value (ABS) and negate (NEG) are considered to be arithmetic operations and cause this exception if one or both operands is a signaling NaN.
- Square root: $\sqrt{x}$, where x is less than zero.

Software can simulate the Invalid Operation exception for other operations that are invalid for the given source operands. Examples of these operations include IEEE Standard 754-specified functions implemented in software, such as Remainder: x REM y, where y is 0 or x is infinite; conversion of a floating-point number to a decimal format whose value causes an overflow, is infinity, or is NaN; and transcendental functions, such as ln (–5) or cos–1(3).

Trap Enabled Results: The original operand values are undisturbed.

Trap Disabled Results: A quiet NaN is delivered to the destination register if no other software trap occurs.

Division-by-Zero Exception (Z)

The Division-by-Zero exception is signaled on an implemented divide operation if the divisor is zero and the dividend is a finite nonzero number. Software can simulate this exception for other operations that produce a signed infinity, such as ln(0), sec（Π/2), csc(0), or 0^{-1}.

Trap Enabled Results: The result register is not modified, and the source registers are preserved.

Trap Disabled Results: The result, when no trap occurs, is a correctly signed infinity.

Overflow Exception (O)

The Overflow exception is signaled when the magnitude of the rounded floating-point result, with an unbounded exponent range, is larger than the largest finite number of the destination format. (This exception also sets the inexact exception and Flag bits.)

Trap Enabled Results: The result register is not modified, and the source registers are preserved.

Trap Disabled Results: The result, when no trap occurs, is determined by the rounding mode and the sign of the intermediate result.

Underflow Exception (U)

Two related events contribute to the Underflow exception:

- Creation of a tiny nonzero result between $\pm 2^{Emin}$ which can cause some later exception because it is so tiny
- Extraordinary loss of accuracy during the approximation of such tiny numbers by denormalized numbers.

IEEE Standard 754 allows a variety of ways to detect these events, but requires they be detected the same way for all operations.

Tininess can be detected by one of the following methods:

- After rounding (when a nonzero result, computed as though the exponent range were unbounded, would lie strictly between $\pm 2^{Emin}$)
- Before rounding (when a nonzero result, computed as though the exponent range and the precision were unbounded, would lie strictly between ±

2^{Emin}).

The MIPS architecture requires that tininess be detected after rounding.

Loss of accuracy can be detected by one of the following methods:

- Denormalization loss (when the delivered result differs from what would have been computed if the exponent range were unbounded)
- Inexact result (when the delivered result differs from what would have been computed if the exponent range and precision were both unbounded).

The MIPS architecture requires that loss of accuracy be detected as an inexact result.

Trap Enabled Results: If Underflow or Inexact traps are enabled, or if the FS bit is not set, then an Unimplemented exception (E) is generated, and the result register is not modified.

Trap Disabled Results: If Underflow and Inexact traps are not enabled and the FS bit is set, the result is determined by the rounding mode and the sign of the intermediate result .

Unimplemented Instruction Exception (E)

Any attempt to execute an instruction with an operation code or format code that has been reserved for future definition sets the Unimplemented bit in the Cause field in the FPU Control/Status register and traps. The operand and destination registers remain undisturbed and the instruction is emulated in software. Any of the IEEE Standard 754 exceptions can arise from the emulated operation, and these exceptions in turn are simulated.

The Unimplemented Instruction exception can also be signaled when unusual operands or result conditions are detected that the implemented hardware cannot handle properly.

These include:

- Denormalized operand, except for Compare instruction
- Quiet Not a Number operand, except for Compare instruction
- Denormalized result or Underflow, when either Underflow or Inexact Enable bits are set or the FS bit is not set.

NOTE: Denormalized and NaN operands are only trapped if the instruction is a convert or computational operation. Moves do not trap if their operands are either denormalized or NaNs.

Trap Enabled Results: The original operand values are undisturbed.

Trap Disabled Results: This trap cannot be disabled.

8 Privileged Instruction

Table 8-1 lists those privileged instructions of Godson-2E.

Table 8-1 Godson-2E Privileged Instructions

OpCode	Description
CACHE	Cache Operation
DMFC0	Doubleword Move From CP0
DMTC0	Doubleword Move To CP0
ERET	Exception Return
MFC0	Move From CP0
MTC0	Move To CP0
TLBP	Prove TLB for Matching Entry
TLBR	Read Indexed TLB Entry
TLBWI	Write Indexed TLB Entry
TLBWR	Write Random TLB Entry

8.1 CP0 Move Instructions

The Godson-2E processor implements Coprocessor 0 move instructions, MTC0, MFC0, DMTC0 and DMFC0. The exact operations of CP0 move instructions on 32/64-bit CP0 registers are summarized Table 8-2.

Table 8-2 CP0 Move Instructions

Instruction	CP0 Register Size	Operation
MFC0 rt, rd	32	rt <- $rd_{31..0}$
MTC0 rt, rd	32	rd <- $rt_{31..0}$
DMFC0 rt, rd	64	rt <- $rd_{63..0}$
DMTC0 rt,rd	64	rd <- $rt_{63..0}$

8.1.1 DMFC0 Instruction

Doubleword Move From System Control Coprocessor

31 26	25 21	20 16	15 11	10 6 5 0
COP0 010000	DMF 00001	rt	rd	0 00000000000

Format: DMFC0 rt, rd

Description: The contents of coprocessor register rd of the CP0 are loaded into general register rt. This operation is defined for theGodson-2E operating in kernel

mode. Execution of thes instruction in user or supervisor mode causes a coprocessor unusable exception. All 64-bits of the general register destination are written from the coprocessor register source.

Operation: GPR[rt] <- CPR[rd]

Exceptions: Coprocessor unusable exception

8.1.2 DMTC0 Instruction

Doubleword Move To System Control Coprocessor

31 26	25 21	20 16	15 11	10 6 5 0
COP0 010000	DMT 00101	rt	rd	0 00000000000
6	5	5	5	11

Format: DMTC0 rt, rd

Description: The contents of general register rt are loaded into coprocessor register rd of the CP0. This operation is defined for the Godson-2E operating in kernel mode. Execution of this instruction in user or supervisor mode causes a coprocessor unusable exception. All 64-bits of the coprocessor register destination are written from the general register source.

Operation: GPR[rd] <- CPR[rt]

Exceptions: Coprocessor unusable exception

8.1.3 MFC0 Instruction

Move From System Control Coprocessor

31 26	25 21	20 16	15 11	10 6 5 0
COP0 010000	MF 00000	rt	rd	0 00000000000
6	5	5	5	11

Format: MFC0 rt, rd

Description: The contents of coprocessor register rd of the CP0 are loaded into general register rt.

Operation: GPR[rt] <- CPR[rd]

Exceptions: Coprocessor unusable exception

8.1.4 MTC0 Instruction

Move To System Control Coprocessor

31 26	25 21	20 16	15 11	10 6 5 0
COP0 010000	MT 00100	rt	rd	0 00000000000
6	5	5	5	11

Format: MTC0 rt, rd

Description: The contents of general register rt are loaded into coprocessor register rd of the CP0.

Operation: GPR[rd] <- CPR[rt]

Exceptions: Coprocessor unusable exception

8.1.5 Usable CP0 Move Instruction in User Mode

When users use DMFC0 or MFC0 to read the coprocessor 0 register No.24 or No.25 in order to get the performance information of the Godson-2E processor, this excution doesn't cause a coprocessor unusable exception.

8.2 TLB Access Instructions

The Godson-2E processor implements TLB instructions, TLBP, TLBI, TLBWI and TLBWR.

8.2.1 TLBP Instruction

Probe TLB For Matching Entry

31 26	25	24 6	5 0
COP0 010000	CO 1	0 0000000000000000000	TLBP 001000
6	1	19	6

Format: TLBP

Description: The *Index* register is loaded with the address of the TLB entry whose contents match the contents of the *EntryHi* register. If no TLB entry matches, the high-order bit of the Index register is set to 0x80000000.

Operation:

Index<-1||0^{25}||undefined^{6}

for I in 0..TLBEntries-1

if(TLB[i]$_{171..141}$and not(0^{15}||TLB[i]$_{216..205}$))

=EntryHi$_{43..13}$) and not(0^{15}||TLB[i]$_{216..205}$)) and

TLB[i]$_{140}$ or (TLB[i]$_{135..128}$=EntryHi$_{7..0}$)) then

Index<=0^{26}||i$_{5..0}$

endif

endfor

Exceptions: Coprocessor unusable exception

8.2.2 TLBR Instruction

Read Indexed TLB Entry

31 26	25	24 6	5 0
COP0 010000	CO 1	0 0000000000000000000	TLBR 000001
6	1	19	6

Format: TLBR

Description: The G bit (which controls ASID matching) read from the TLB is written into both of the *EntryLo0* and *EntryLo1* registers. The *EntryHi* and *EntryLo* registers are loaded with the contents of the TLB entry pointed at by the contents of the TLB Index register. TLBR can be executed in mapped spaces.

Operation:

PageMask<-TLB[Index$_{5..0}$]$_{255..192}$

EntryHi<- TLB[Index$_{5..0}$]$_{191..128}$ and not TLB[Index$_{5..0}$]$_{255..192}$

EntryLo1<- TLB[Index$_{5..0}$]$_{127..65}$|| TLB[Index$_{5..0}$]$_{140}$

EntryLo0<- TLB[Index$_{5..0}$]$_{63..1}$|| TLB[Index$_{5..0}$]$_{140}$

Exceptions: Coprocessor unusable exception

8.2.3 TLBWI Instruction

Write Indexed TLB Entry

31 26	25	24 6	5 0
COP0 010000	CO 1	0 0000000000000000000	TLBWI 000010
6	1	19	6

Format: TLBWI

Description: The *G* bit of the TLB is written with the logical AND of the *G* bits in the *EntryLo0* and *EntryLo1* registers. The TLB entry pointed at by the contents of the TLB *Index* register is loaded with the contents of the *EntryHi* and *EntryLo* registers. The operation is invalid (and the results are unspecified) if the contents of the TLB *Index* register are greater than the number of TLB entries in the processor.

Operation:

TLB[$\text{Index}_{5..0}$]<-PageMask||(EntryHi and not PageMask)||EntryLo1||EntryLo0

Exceptions: Coprocessor unusable exception

8.2.4 TLBWR Instruction

Write Random TLB Entry

31 26	25	24 6	5 0
COP0 010000	CO 1	0 0000000000000000000	TLBWR 000110
6	1	19	6

Format: TLBWR

Description: The G bit of the TLB is written with the logical AND of the *G* bits in the *EntryLo0* and *EntryLo1* registers. The TLB entry pointed at by the contents of the TLB *Random* register is loaded with the contents of the *EntryHi* and *EntryLo* registers.

Operation:

TLB[$\text{Random}_{5..0}$]<-PageMask||(EntryHi and not PageMask)||EntryLo1||EntryLo0

Exceptions: Coprocessor unusable exception

8.3 ERET Instruction

Exception Return

31 26	25	24 6	5 0
COP0 010000	CO 1	0 0000000000000000000	ERET 011000
6	1	19	6

Format: ERET

Description: ERET is the instruction for returning from an interrupt, exception, or error trap. Unlike a branch or jump instruction, ERET does not execute the next instruction. ERET must not itself be placed in a branch delay slot. If the processor is servicing an error trap (*SR2* = 1), then load the PC from the *ErrorEPC* and clear the *ERL* bit of the *Status* register (*SR2*). Otherwise (*SR2* = 0), load the PC from the *EPC*, and clear the *EXL* bit of the *Status* register (SR1). An ERET executed between a LL and SC also causes the SC to fail. If there is no exception (*EXL*=0 and *ERL*=0 in the *Status* register), execution of an ERET instruction is meaningless. Execution of an ERET when *ERL*=0, regardless of the state of *EXL*, sets *EXL* to 0 and a jump is taken to the address presently held in the *EPC* register, even when there is no exception.

Operation:

If SR_2=1 then

PC<-ErrorEPC

SR<-$SR_{31..3}$||0||$SR_{1..0}$

else

PC<-EPC

SR<-$SR_{31..2}$||0||SR_0

Endif

LLbit<-0

Exceptions: Coprocessor unusable exception

8.4 CACHE Instruction

31 26	25 21	20 16	15 11	10 6	5 0
COP0 010000	base	op	offset		
6	5	5	16		

Format: CACHE op, offset(base)

Description: The 16 bit *offset* is sign-extended and added to the contents of general register *base* to form a CacheOp virtual address (VA). The VA is translated to a physical address (PA) through the TLB, and the 5-bit opcode (decoded in Table 8-3) specifies a cache operation for that address, together with the affected cache. Operation of this instruction on any combination not listed in the tables below is undefined. The operation of this instruction on uncached addresses is also undefined.

Table 8-3 CACHE Instruction Op Field Encoding

Op Field	CACHE Instruction Variation	Target Cache
00000	Index Invalidate	(I)
00001	Index WriteBack Invalidate	(D)
00101	Index Load Tag	(D)
01001	Index Store Tag	(D)
10001	Hit Invalidate	(D)
10101	Hit WriteBack Invalidate	(D)
11001	Index Load Data	(D)
11101	Index Store Data	(D)
00011	Index WriteBack Invalidate	(S)
00111	Index Load Tag	(S)
01011	Index Store Tag	(S)
10011	Hit Invalidate	(S)
10111	Hit WriteBack Invalidate	(S)
11011	Index Load Data	(S)
11111	Index Store Data	(S)

Operation:

vAddr<-$((offset_{15})^{48}||offset_{15..0})$+GPR[base]

(pAddr,uncached)<-AddressTranslation(vAddr,DATA)

CacheOp(op,vAddr,pAddr)

Exceptions: Coprocessor unusable exception

8.4.1 Index Invalidate (I)

Index Invalidate (I) sets four blocks of four ways in the primary instruction cache to *Invalid*. VA[13:5] defines the line address to be invalidated. The invalidation takes place by writing the primary instruction cache state bit to 0 (*Invalid*).

8.4.2 Index WriteBack Invalidate (D)

Index WriteBack Invalidate (D) sets a block in the primary data cache to *Invalid*. VA[13:5] defines the address and VA[1:0] defines the way to be invalidated. The invalidation takes place by writing the primary data cache state bit to 00 (*Invalid*). If this block in the primary data cache is dirty, it must be written back to the secondary cache.

8.4.3 Index WriteBack Invalidate (S)

Index WriteBack Invalidate (S) instruction sets a block in the secondary cache to *Invalid* and writes back any dirty data to the System interface unit. This operation extends to any blocks in the primary data or instruction caches which are subsets of the secondary cache block. The CACHE instruction physical address, PA[16:5], defines the address and PA[1:0] defines the way to be invalidated.

The invalidation occurs in the following sequence:

1. The processor reads the STag and State bits from the secondary cache tag array. If State = 00 (*Invalid*) no further activity takes place. If there is a valid entry, then the STag is used to interrogate the primary instruction and data caches.
2. The processor reads each subset block from the primary instruction cache. If ITag = STag and IState = 1 (*Valid*) then the block is invalidated by writing the IState bit to 0 (*Invalid*).
3. Read each subset block from the primary data cache. If DTag = STag and DState is not equal to 00 (*Invalid*), then write the DState bits = 00 (*Invalid*). If the original block is *Dirty*, also write this block back to the secondary cache.
4. Set the state of the secondary cache block to 00 (*Invalid*). If the secondary cache block's original State bits were 11 (*Dirty*), the block is written back to the system interface unit.

8.4.4 Index Load Tag (D)

Index Load Tag (D) reads the primary data cache tag fields into the CP0 *TagLo* and *TagHi* registers. VA[13:5] defines the address and VA[1:0] defines the way of the tag to be read.

The following mapping defines the operation:

TagLo[5:4] = SCWay

TagLo[7:6] = State bits

TagLo[31:8] = Tag[35:12]

TagHi[3:0] = Tag[39:36]

TagHi[31:29] = StateMod bits

All other CP0 *TagLo* and *TagHi* bits are set to 0.

8.4.5 Index Load Tag (S)

Index Load Tag (S) reads the secondary cache tag fields into the CP0 *TagLo* and *TagHi* registers. The PA[16:5] defines the address and PA[1:0] defines the way to be read.

The following mapping defines the operation:

TagLo[11:10] = State bits

TagLo[31:13] = Tag[35:17]

TagHi[3:0] = Tag[39:36]

All other CP0 *TagLo* and *TagHi* bits are set to 0.

8.4.6 Index Store Tag (D)

Index Store Tag (D) stores the CP0 *TagLo* and *TagHi* registers into the primary data cache tag array. VA[13:5] defines the address and VA[1:0] defines the way of the tag to be written.

The following mapping defines the operation:

SCWay = TagLo[5:4]

State bits = TagLo[7:6]

Tag[35:12] = TagLo[31:8]

Tag[39:36] = TagHi[3:0]

8.4.7 Index Store Tag (S)

Index Store Tag (S) stores fields from the CP0 *TagLo* and *TagHi* registers into the secondary cache tag array. The PA[13:5] defines the address and PA[1:0] defines the way to be read.

The following mapping defines the operation:

State bits = TagLo[11:10]

Tag[35:17] = TagLo[31:13]

Tag[39:36] = TagHi[3:0]

8.4.8 Hit Invalidate (D)

Hit Invalidate (D) invalidates an entry in the data cache which matches the PA of the CACHE instruction. Both ways tags at VA[13:5] are read from the data cache. If the DState is not equal to 00 (*Invalid*) and the PA of the CACHE instruction matches the DTag from the data cache tag array, then the State bits are written to 00 (*Invalid*).

8.4.9 Hit Invalidate (S)

Hit Invalidate (S) invalidates all entries in the secondary, primary instruction, and primary data caches which match the PA of the CACHE instruction.

The following sequence takes place:

1. The processor reads the Tags from four ways of the secondary cache at the address pointed to by the PA of the CACHE instruction. If the tag entry's Stag matches the CACHE instruction PA, and the State of the entry is not equal to 00 (*Invalid*), then a Hit has occurred in that entry. If there is no Hit, the CACHE instruction completes.
2. The processor reads each subset block from the primary instruction cache. If ITag = STag and IState = 1 (*Valid*) then the block is invalidated by writing the IState bit to 0 (*Invalid*).
3. Read each subset block from the primary data cache. If DTag = STag and DState is not equal to 00 (*Invalid*), then write the DState bits = 00 (*Invalid*).
4. The processor sets the tag array entry of the secondary cache block which was hit to State = 00 (*Invalid*) and Tag = PA of CACHE instruction.

8.4.10 Hit WriteBack Invalidate (D)

Hit Writeback Invalidate (D) invalidates an entry in the primary data cache which matches the PA of the CACHE instruction. In addition, it writes back to the secondary cache any *dirty* data found in the primary data cache. Four way DTags at VA[13:5] are read from the data cache. If the DState is not equal to 00 (*Invalid*) and PA of the CACHE instruction matches the DTag, then the DState bits of the entry are set to 00 (*Invalid*).

8.4.11 Hit WriteBack Invalidate (S)

Hit Writeback Invalidate (S) checks for a block which matches the CACHE instruction PA in the secondary cache, invalidates it, and writes back any dirty data to the System interface unit. This operation extends to any blocks in the primary data or instruction caches which are subsets of the secondary cache block.

The operation takes place in the following sequence:

1. The processor reads the STag and State bits from four ways of the secondary tag array. If the PA of the CACHE instruction matches the STag, and the State does not equal 00 (*Invalid*), a hit has occurred. If there is a hit, the STag is used to interrogate the primary caches. If there is not a hit, the instruction ends.
2. The processor reads each subset block from the primary instruction cache. If there is a match then invalidate the block by writing the IState bit to 0 (*Invalid*).
3. Read each subset block from the primary data cache. If there is a match then write the DState bits = 00 (*Invalid*), and the DState parity bit = 0. If the original State of any subset block is dirty, also write it back to the secondary cache.
4. Set the state of the secondary cache block to 00 (*Invalid*). If the secondary cache block's original State bits were 11 (*Dirty*), the block is written back to the system interface unit.

8.4.12 Index Load Data (D)

Index Load Data (D) loads a doubleword of data into CP0 *TagHi* and *TagLo*. The address of the target doubleword is VA[13:3] of the CACHE instruction. The way of

the target doubleword is VA[1:0] of the CACHE instruction.

8.4.13 Index Load Data (S)

Index Load Data (S) loads a doubleword of data into CP0 *TagHi* and *TagLo*. The address of the target doubleword is VA[16:3] of the CACHE instruction. The way of the target doubleword is VA[1:0] of the CACHE instruction.

8.4.14 Index Store Data (D)

Index Store Data (D) stores a doubleword of data into the data cache from the CP0 *TagHi* and *TagLo* registers. The address where this doubleword will be written is defined by VA[13:3] of the CACHE instruction. The way is defined by VA[1:0]. The data doubleword comes from CP0 *TagHi* and *TagLo*.

8.4.15 Index Store Data (S)

Index Store Data (S) stores a doubleword of data into the second cache from the CP0 *TagHi* and *TagLo* registers. The address where this doubleword will be written is defined by VA[16:3] of the CACHE instruction. The way is defined by VA[1:0]. The data doubleword comes from CP0 *TagHi* and *TagLo*.

9 DDR SDRAM Control Interface

The built-in memory controller of Godson-2E processor fully conforms to DDR SDRAM industry standard JESD79C. All memory read/writes are implemented according to JESD79C specification.

9.1 DDR SDRAM Controller Functional Overview

Godson-2E processor supports a maximum of 4 physical memory banks (there are 4 DDR SDRAM chip selects). Address bus is 15 bits width, i.e., 13 bits of column row address and 2 bits of logical bank address.

Godson-2E processor supports memory chip type specified in JESD79C, see table 9-1.

Table 9-1 DDR SDRAM chip supported

BITS	Density	Org.	Row Addr.	Col Addr.
0000	64Mb	16Mb X 4	DA[11:0]	DA[9:0]
	128Mb	16Mb X 8		
0001	64Mb	8Mb X 8	DA[11:0]	DA[8:0]
	128Mb	8Mb X 16		
0010	64Mb	4Mb X 16	DA[11:0]	DA[7:0]
0011	128Mb	32Mb X 4	DA[11:0]	DA[11],DA[9:0]
0100	256Mb	64Mb X 4	DA[12:0]	DA[11],DA[9:0]
	512Mb	64Mb X 8		
0101	256Mb	32Mb X 8	DA[12:0]	DA[9:0]
	512Mb	32Mb X 16		
0110	256Mb	16Mb X 16	DA[12:0]	DA[8:0]
0111	512Mb	128Mb X 4	DA[12:0]	DA[12:11],DA[9:0]
1000	1Gb	256Mb X 4	DA[13:0]	DA[12:11],DA[9:0]
1001	1Gb	128Mb X 8	DA[13:0]	DA[11],DA[9:0]
1010	1Gb	64Mb X 16	DA[13:0]	DA[9:0]

The only memory read/write requests accepted by the built-in memory controller are those generated from inside the processor or memory request of external device. In all memory operation, the controller runs in slave state.

The built-in memory controller implements a dynamic page management strategy. For any memory access, the selection of Open Page or Close Page strategy is handled by the hardware, which is transparent to software designers.

9.2 DDR SDRAM Read Protocol

DDR SDRAM read protocol is described in figure 9-1. In the figure, the COMMAND signal is made up of RAS, CAS and WE. For read, RAS=1, CAS=0 and WE=1; ADDRESS signal consist of MCS, MBA and MSA.

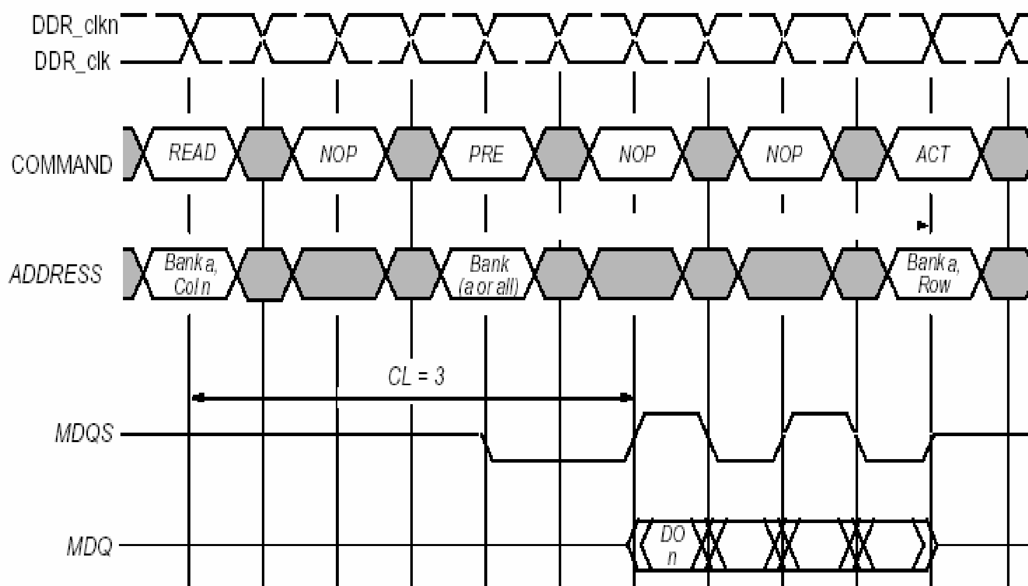


Figure 9-1 DDR SDRAM read protocol

9.3 DDR SDRAM Write Protocol

DDR SDRAM write protocol is described in figure 9-2. In the figure, the COMMAND signal is made up of RAS, CAS and WE. For write, RAS=1, CAS=0 and WE=0; ADDRESS signal consist of MCS, MBA and MSA. The bytes to write are masked by MDM, which is different to read operation.

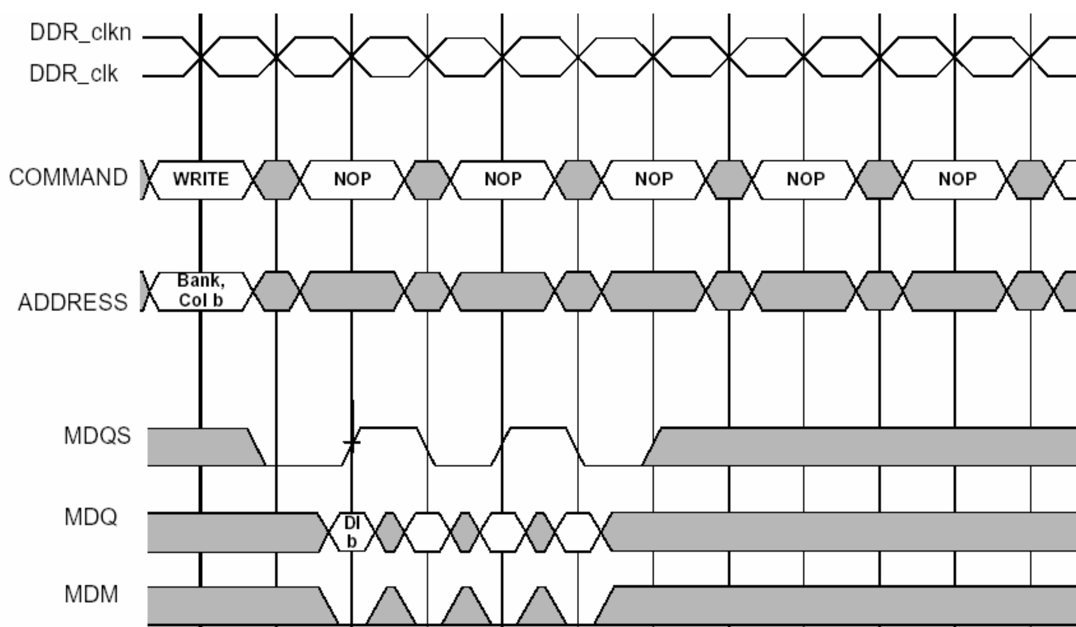


Figure 9-2 DDR SDRAM write protocol

9.4 DDR SDRAM Configuration

The system may be installed with different types of DDR SDRAM chips; therefore, configuration is needed after power-on reset. In JESD79C, detailed configuration procedures are specified.

In the design of Godson-2E processor, DDR SDRAM configuration is dealt with after the initialization of system board, before memory is needed. The configuration is done with writing 32-bit config register at physical address 40’h1FF0008. Table 9-2 list each bit’s meaning.

Table 9-2 DDR SDRAM configuration register

Bit	Field Name	Access	Description
31：30	Undefined	R/W	Undefined
29	DIMM_dic	R/W	Memory module presented in DIMM_slot0: 0: no, 1: yes
28:27	DIMM_MODULE_NUM	R/W	No. of module in DIMM0/DIMM1: 2’b00: DIMM1: 1; DIMM0: 1 2’b01: DIMM1: 1; DIMM0: 2 2’b10: DIMM1: 2; DIMM0: 1 2’b11: DIMM1: 2; DIMM0: 2
26	IS_SEQ	R/W	Burst mode intra-block sequencing: 1’b0: sequential 1’b1: interleaved
25:22	DDR Type	R/W	See table 9-1.
21:10	tREF	R/W	SDRAM refresh interval count (100MHz):

Bit	Field Name	Access	Description
			780 7.8us 1560 15.6us SDRAM refresh interval count (133MHz): 1040 7.8us 2080 15.6us SDRAM refresh interval count (166MHz); 1300 7.8us 2600 15.6us
9	TRCD	R/W	Cycles between row address valid and column address valid 1’b0 2 cycles (DDR100) 1’b1 3 cycles (DDR266, DDR333）
8:7	TRFC	R/W	Cycles between AUTO_REFRESH and ACTIVE 2’b00 Null 2’b01 8 cylces （DDR100） 2’b10 10 cycles（DDR266） 2’b11 12 cycles（DDR333）
6	TRAS	R/W	Cycles between ACTIVE and PRECHARGE 1’b0 5 cycles（DDR100） 1’b1 7 cycles（DDR266、DDR333）
5:4	TCAS	R/W	Cycles between read request and arrival of the first data: 2’b00 1.5 cycles 2’b01 2 cycles 2’b10 2.5 cycles 2’b11 3 cycles
3	TWR	R/W	Cycles between last data written and PRECHARGE: 1’b0 2 cycles (DDR100) 1’b1 3 cycles (DDR266, DDR333)
2	TRP	R/W	Cycles of PRECHARGE command duration 1’b0 2 cycles (DDR100) 1’b1 3 cycles (DDR266,DDR333)
1:0	TRC	R/W	Cycles between ACTIVE command and ACTIVE/AUTO_REFRESH command 2’b00 Null 2’b01 7 cycles (DDR100) 2’b10 9 cycles (DDR266) 2’b11 10cycles (DDR333) Note: The sum of PRECHARGE, RAS and CAS latency are equivalent to TRC, thus it was ignored by the controller.

9.5 DDR SDRAM Sampling Mode Configuration

The return of valid data from memory is asynchronously marked by DQS, which means sampling clock is not synchronized to data. Usually, a memory controller needs a DLL (Delay-locked loop) to do this, i.e., through adjusting the phase of sampling clock to find a suitable sampling point. Using DLL to solve this would not only required a DLL IP, but also arouse the problem of synchronize the sampling clock to processor core clock. Because the sampling clock is adjusted by DLL, a mechanism of inter clock domain data transfer like FIFO is needed.

In Godson-2E processor, the memory clock output is divided from the core clock and forms a 1:6, 1:8, 1:10 or 1:12 relationship. Regarding the core clock is a high multiple of memory clock, we propose a flexible and efficient strategy using core clock to sample data directly. Sample point is set by software, and the clock is the processor core clock. This design can save both DLL IP and the delay caused by intra clock domain data transfer. Following is an example of the proposed strategy where the ratio of DDR SDRAM clock frequency to the core's is 1:10.

As shown in figure 3-3, core clock frequency is 10 times of memory clock frequency. DQS*_dly is latched from DQS*, and sample point is generated via counting how many times DQS*_dly go valid. The counting number is set by software, thus achieved the goal of software tuning.

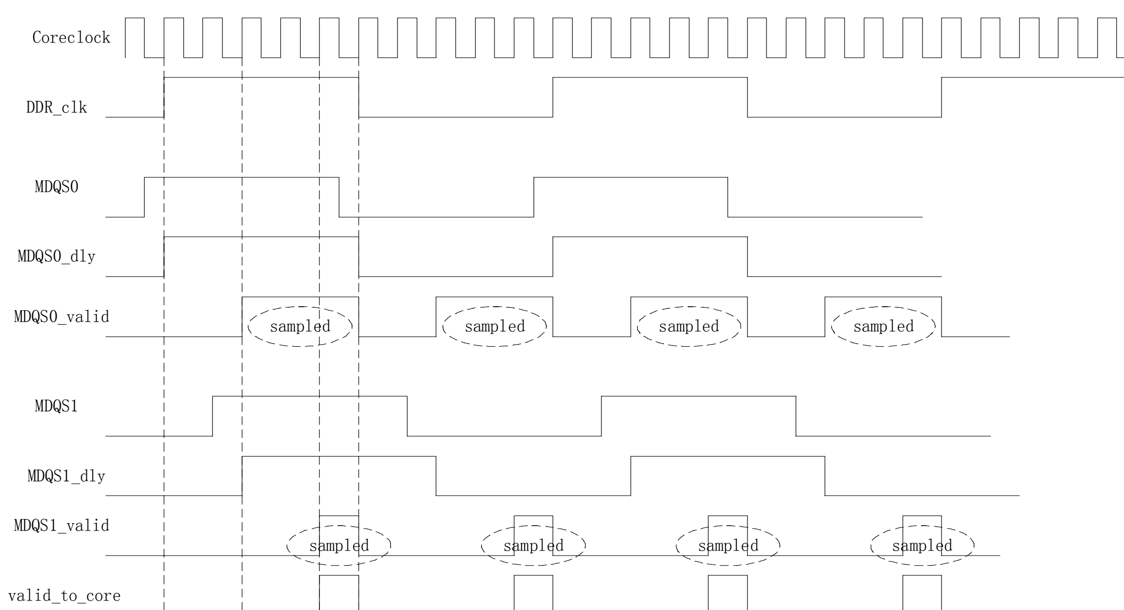


Figure 9-3 DDR SDRAM sampling mode when memory to core ratio is 1:10

In Godson-2E processor, the configuration of memory read sample point is carried out after the system board finished initialization, just before using memory. It's done by writing 2 bits of valid parameter into 32-bit register at physical address 40'h1FF00030. The parameter (sample_point)is located on the least significant two bits of the register, see table 9-3 for details.

Table 9-3 Sample point configuration register

Sample point register	Description
2'b00	Sample at 1 clock after DQS valid.
2'b01	Sample at 2 clocks after DQS valid.
2'b10	Sample at 3 clocks after DQS valid.
2'b11	Invalid

In Godson-2E, the frequency ratio of memory controller clock output to core frequency is 1:6, 1:8, 1:10 and 1:12. For general case, sample point is set by default value, i.e., when the ratio is 1:6, sample point register is 2'b00, sampling at 1 clock after DQS valid; when the ratio is 1:8 and 1:10, sample point register is 2'b01, sampling 2 clocks later; when the ratio is 1:12, sample point register is 2'b10, 3 clocks later.

10 Performance Tuning

This chapter describes some architecture impacts of Godson-2E on software and ways to make efficient software for godson-2E. The Godson-2E architecture, like all other RISC architectures, depends on careful attention of data alignment and instruction scheduling to achieve high performance.

10.1 User instruction Latency and Repeat Rate

Table 10-1 shows the latencies and repeat rates for all user instructions executed in ALU1/2, MEM, FALU1/2 functional units, kernel mode instructions and control instructions are not included.

Table 10-1 Latencies and Repeat Rates for User Instructions

Instrutions	unit	latency	repeat rate
Integer operations			
Add/sub/logical/shift/lui/cmp	ALU1/2	2	1
Trap/branch	ALU1	2	1
MF/MT HI/LO	ALU1/2	2	1
(D)MULT(U)	ALU2	5	2(split)
(D)MULT(U)G	ALU2	5	1
(D)DIV(U)	ALU2	2-38	1-37
(D)DIV(U)G	ALU2	2-38	1-37
(D)MOD(U)G	ALU2	2-38	1-37
Load	MEM	5	1
Store	MEM	-	1
Floating-point operations			
(D)MTC1/(D)MFC1	MEM	5	1
Abs/Neg/C.cond/Bc1t/Bc1f/ Move/Cvt*	FALU1	3	1
Round/Trunc/Ceil/ Floor/Cvt*	FALU1	5	1
Add/Sub/Mul/Madd/Msub/ Nmadd/Nmsub	FALU1/2	7	1
Div.s	FALU2	5-11	4-10
Div.d	FALU2	5-18	4-17
Sqrt.s	FALU2	5-17	4-16
Sqrt.d	FALU2	5-32	4-31
Lwc1,Ldc1	MEM	5	1
Swc1,Sdc1	MEM	-	1

Please note the following about table 10-1:

- The latency of an execution pipeline is the number of cycles between the time an instruction is issued and the time a dependent instruction(which uses its result as an operand) can be issued.
- The repeat rate of the pipeline is the number of cycles that occur between the issuance of one instruction and the issuance of the next instruction to the same execution unit.
- The latency of DIV* operations depends on the operand. It can be estimated as:

  (lz(a) < lz(b))?(lz(b)-lz(a))/2 + 4 – ez(c) / 2 : 1

  for a/b=c, lz: leading zero, ez: trailing zero.
- The repeat rate for load/store does not include load-link and store-conditional. LL/SC are wait-issue operations, that is, they are not issued until they come to the head of reorder queue and the cp0queue is empty.
- There is no special usage limit for HI/LO register, they are treated just the same as other general purpose registers.
- CTC1/CFC1 is not included in this table. They are serialized like many other control instructions.
- Multimedia instructions are not included in this table. Because they are implemented by extending the FORMAT field of normal floating-point instructions, we can easily deduce the function unit and latency of them.

10.2 Instruction extensions

Godson-2E implements several instruction extensions.

- Fix point multiply and division that write only one result into general-purpose registers. Including 12 instructions:

  (D)MULTG, (D)MULTUG, (D)DIVG

  (D)DIVUG, (D)MODG , (D)MODUG

  Multiply and divisions of standard MIPS instruction set write two special registers (HI/LO) for one operation, which is hard to implement in RISC pipelines. To use the results one has to use additional instructions to fetch it from HI/LO into general-purpose register. What's more, many MIPS processors have limits on the usage of these instructions due to pipeline problems. Our new instructions should be both faster and easier to use.

- Multimedia instruction extension

  Documented in other manuals.

- Fix-point operations using floating-point data path

  When running integer programs the floating-point data path is often idle, these instructions intend to provide a way to utilize them.

10.3 Instruction Stream

The following sections describe considerations for the instruction stream.

10.3.1 Instruction alignment

Every cycle Godson-2E can fetch four instructions from any word-aligned address within a cache line. Basic block of frequently executed branch targets should avoid crossing the cache line boundary by proper alignment. The branch instruction among the four instruction fetched will affect the output. If the first instruction is a taken branch, then the last two instructions are useless. If the last instruction is a branch, then even if the branch is taken the processor has to wait for its delay slot instructions in the next cache line. If there were two branch instructions in this bundle, it would take 2 cycles for the decoder to handle them, because only one branch can be decoded each cycle.

10.3.2 Branch handling

In godson-2E processors, an unexpected change in I-stream address will result in about 10 lost cycles. "Unexpected" may mean any branch-taken or may mean a miss-predicted branch. In current godson-2E implementation, even a correctly predicted taken branch will be slower (waste one cycle because BTB would not give correct next PC for conditional branches) than straight-line code.

Compilers should follow these rules to minimize unexpected branches:

- Godson-2E branch prediction schemes are different from any other high performance processors, and they vary a bit for different revisions. Based on execution profiles, compilers should physically rearrange code so that it has matching behavior.
- Make basic blocks as big as possible. A good goal is 20 instructions on average between branch-taken. This requires unrolling loops so that they contain at least 20 instructions, and putting subroutines of less than 20

instructions directly in line. It also requires using execution profiles to rearrange code so that the frequent case of a conditional branch falls through. For very high-performance loops, it will be profitable to move instructions across conditional branches to fill otherwise wasted instruction issue slots, even if the instructions moved will not always do useful work. Note that using the Conditional Move instructions can sometimes avoid breaking up basic blocks (not yet implemented in current revision-Godson-2EC, but they will show up in godson-2ED).

- In an if-then-else construct whose execution profile is skewed even slightly away from 50%-50% (51-49 is enough), put the infrequent case completely out of line, so that the frequent case encounters *zero* branch-takens, and the infrequent case encounters *two* branch-takens. If the infrequent case is rare (5%), put it far enough away that it never comes into the I-cache. If the infrequent case is extremely rare (error message code), put it on a page of rarely executed code and expect that page *never* to be paged in.

Section 2.1 gives out a brief description of the fetch-decode unit. We can see that the branch prediction scheme is composed of:

- Static prediction. For branch likely instructions and jump instructions.
- Gshare predictor. A 9-bit GHR plus 4K-entry PHT. For conditional branches.
- BTB. 16-entry fully associative. Used for predicting target PC of jump register instructions.
- RAS. 4-entry. Used for predicting target PC of function return instruction (jr31).

There are several notes for software considerations.

Be very careful to use branch likely instructions on godson-2E processors. Branch likely instructions may be very useful for simple statically scheduled in-order scalar processors, but not as useful for modern high performance processors. The branch prediction hardware in modern high performance processors is so sophisticated that they can often correctly predict the direction for more than 90% branches (E.g., current godson-2E processor can correctly predict the direction of 85-100% conditional branches, with an average of 95%). In this case compiler should not use branch likely instructions without a very high confidence about the prediction.

In fact we have found that gcc(version 3.3) often does better job with –mno-branch-likely option.

The fetch-decode unit is split into three pipeline stages, and branch destinations are calculated in the third stage. Taken branches have two cycle bubbles, that is, if a branch at PC is fetched at cycle 0, cycle 1 will fetch PC+16, cycle 2 will fetch PC+32, correct destination will be given to fetch unit at cycle 3. Minimize taken branches will help.

The BTB in godson-2E is used purely for jump register instructions (jr with exception of jr31, and jalr).

Destinations of jr31 instructions are predicted via a four-entry return address stack. Efficient prediction of function returns relies on software follow the convention of using jr31 as the function return instruction.

10.3.3 Improving Instruction Stream density

Compilers should try to use profiles to make sure that almost 100% of the bytes brought into the instruction cache are actually executed. This requires alignments of branch targets and putting rarely executed code out of line.

10.3.4 Instruction scheduling

Godson-2E has an instruction window to perform dynamic instruction scheduling. But since the window size and other resources are limited, it is not perfected. Compiler can help here. Modern compilers often have models to learn CPU's capability and they can act well upon given information.

"Result latency" is defined as the number of CPU cycles that must elapse between an instruction that writes a result register and one that uses that register, if execution-time stalls are to be avoided. Thus, with a latency of zero, the instruction writes a result register and the instruction that uses that register can be multiple-issued in the *same* cycle. With a latency of 2, if the writing instruction is issued at cycle N, the reading instruction can issue no earlier than cycle N+2.

Latency is implementation specific. Most godson-2E instructions have non-zero latency. Compilers should schedule code so that a result is not used too soon, at least in frequently executed code (inner loops, as identified by execution profiles). In general, this will require unrolling loops and inlining short procedures.

Compilers should try to schedule code to match the above latency rules and also

to match the multiple-issue rules. If doing both is impractical for a particular sequence of code, the latency rules are more important.

10.4 Memory accesses

The execution of load and store instructions can greatly affect performance. These instructions are executed quickly if the required memory block is contained in the primary data cache; they will be a bit more slow if data are in L2 cache; otherwise they will be significant delayed waiting for the access to the main memory. Out-of-order execution and non-blocking caches reduce the performance loss due to these delays, however.

Current revisions of Godson-2E have not directly provided prefetch instructions, but one can use load-to-zero-register to achieve some kinds of prefetch effects. To reduce overhead such instructions won't raise exceptions upon illegal addresses.

Compiler should try hard to eliminate unnecessary memory accesses. Memory access latency is quite long in current godson-2E processors (even a cache-hit operation takes 5 cycle in godson-2E), and the reorder queues are not big enough to tolerate all of it.

Software should pay enough attention to data alignment. Aggregates (arrays, some records, subroutine stack frames) should be allocated on cache line aligned boundaries to take advantage of cache line aligned data paths, and to decrease the number of cache fills. Items within aggregates that are forced to be unaligned (records, common blocks) should generate compile-time warning messages. Users must be educated that the warning message means that they are taking a big performance hit. Compiled code for parameters should assume that the parameters are aligned. Frequently used scalars should reside in registers.

10.5 Other Tips

- Utilize all floating-point registers. Godson-2E has 32 64-bit floating-point registers, while O32 ABI exposes only 16 to the user. Use N32 or N64 ABI should help.
- Use performance counters. Godson-2E’s performance counter can be used to monitor real-time performance characters of programs. Compilers and software writers can analysis the results to improve their code.

AppendixA Godson new integer instructions

1. MULT.G — Multiply Word (Godson2)

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 011111	rs	rt	rd	0 00000	MULT.G 011000
6	5	5	10		6

- **Format:**

  MULT.G rd, rs, rt

- **Purpose:**

  To multiply 32-bit signed intergers.

- **Description:**

  rd ← rs * rt

  The 32-bit word value in GPR *rt* is multiplied by the 32-bit value in GPR *rs*, treating both operands as signed values, to produce a 64-bit result. The low-order 32-bit word of the result is placed into special register rd.

  No arithmetic exception occurs under any circumstances.

- **Operation:**

  prod ← $GPR[rs]_{31..0}$ * $GPR[rt]_{31..0}$

  rd ← sign_extend($prod_{31..0}$)

- **Exceptions:**

  None

2. MULTU.G — Multiply Unsigned Word (Godson2)

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 011111	rs	rt	rd	0 00000	MULTU.G 011001
6	5	5	10		6

- **Format:**

MULTU.G rd, rs, rt

- **Purpose:**

To multiply 32-bit unsigned intergers.

- **Description:**

rd ← rs * rt

The 32-bit word value in GPR *rt* is multiplied by the 32-bit value in GPR *rs*, treating both operands as unsigned values, to produce a 64-bit result. The low-order 32-bit word of the result is placed into special register rd.

No arithmetic exception occurs under any circumstances.

- **Operation:**

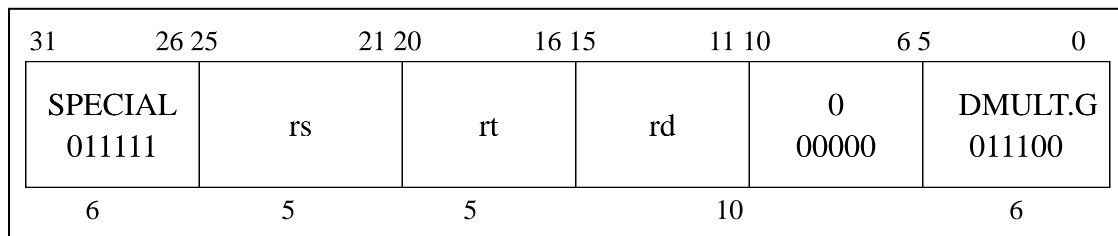
prod ← (0 || $GPR[rs]_{31..0}$) * (0 || $GPR[rt]_{31..0}$)

rd ← sign_extend($prod_{31..0}$)

- **Exceptions:**

None

3. DMULT.G — DoubleWord Multiply (Godson2)



31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 011111	rs	rt	rd	0 00000	DMULT.G 011100
6	5	5	10		6

- **Format:**

  DMULT.G rd, rs, rt

- **Purpose:**

  To multiply 64-bit signed intergers.

- **Description:**

  rd ← rs * rt

  The 64-bit word value in GPR *rt* is multiplied by the 64-bit value in GPR *rs*, treating both operands as signed values, to produce a 128-bit result. The low-order 64-bit word of the result is placed into special register rd.

  No arithmetic exception occurs under any circumstances.

- **Operation:**

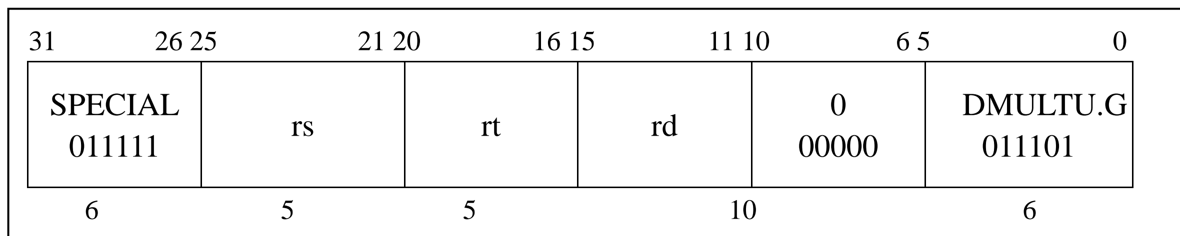
  prod ← GPR[rs] * GPR[rt]

  rd ← $prod_{63..0}$

- **Exceptions:**

  None

4. DMULTU.G—Doubleword Multiply Unsigned (Godson2)



- **Format:**

  DMULT.G rd, rs, rt

- **Purpose:**

  To multiply 64-bit unsigned intergers.

- **Description:**

  rd ← rs * rt

  The 64-bit word value in GPR *rt* is multiplied by the 64-bit value in GPR *rs*, treating both operands as unsigned values, to produce a 128-bit result. The low-order 64-bit word of the result is placed into special register rd.

  No arithmetic exception occurs under any circumstances.

- **Operation:**

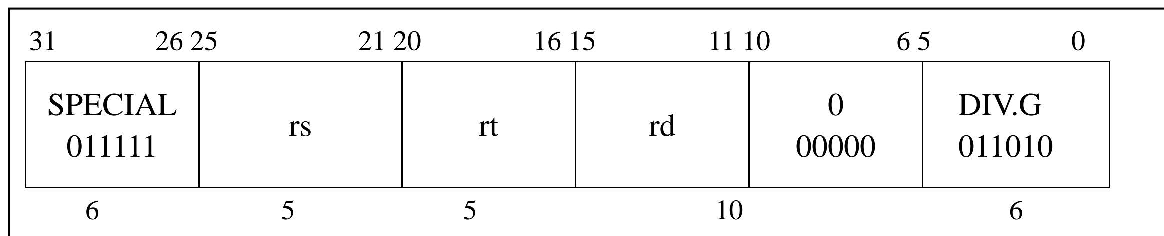
  prod ← (0 || GPR[rs]) * (0 || GPR[rt])

  rd ← $prod_{63..0}$

- **Exceptions:**

  None

5. DIV.G —Divide Word (Godson2)



- **Format:**

  DIV.G rd, rs, rt

- **Purpose:**

  To divide 32-bit signed intergers.

- **Description:**

  rd ← rs / rt

  The 32-bit word value in GPR *rs* is divided by the 32-bit value in GPR *rt,* treating both operands as signed values. The 32-bit quotient is placed into special register rd.

  No arithmetic exception occurs under any circumstances.

- **Operation:**

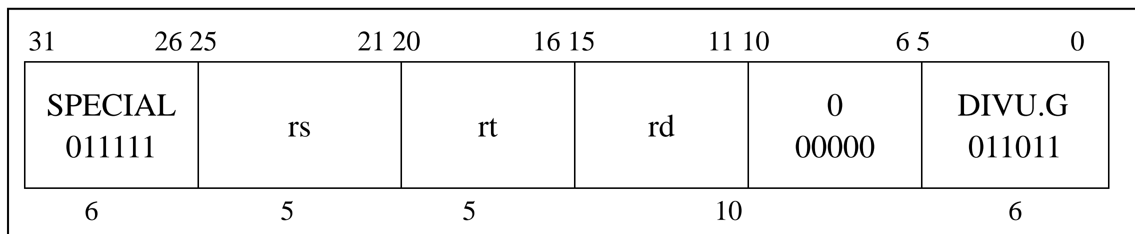
  $q \leftarrow GPR[rs]_{31..0}$ div $GPR[rt]_{31..0}$

  $LO \leftarrow sign_extend(q_{31..0})$

- **Exceptions:**

  None

6. DIVU.G — Divide Unsigned Word (Godson2)



31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 011111	rs	rt	rd	0 00000	DIVU.G 011011
6	5	5	10		6

- **Format:**

DIVU.G rd, rs, rt

- **Purpose:**

To divide 32-bit unsigned intergers.

- **Description:**

rd ← rs / rt

The 32-bit word value in GPR rs is divided by the 32-bit value in GPR rt, treating both operands as unsigned values. The 32-bit quotient is placed into special register rd.

No arithmetic exception occurs under any circumstances.

- **Operation:**

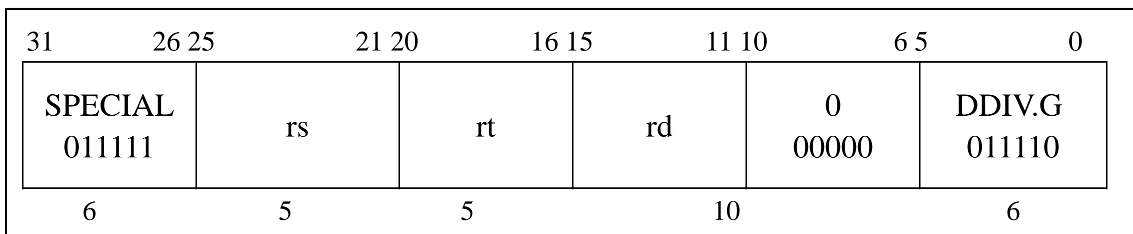
$q \leftarrow (0 \,||\, GPR[rs]_{31..0})$ div $(0 \,||\, GPR[rt]_{31..0})$

$rd \leftarrow$ sign_extend($q_{31..0}$)

- **Exceptions:**

Reserved Instruction

7. DDIV.G — Doubleword Divide (Godson2)



- **Format:**

  DDIV.G rd,rs, rt

- **Purpose:**

  To divide 64-bit signed intergers.

- **Description:**

  rd ← rs / rt

  The 64-bit doubleword in GPR *rs* is divided by the 64-bit doubleword in GPR *rt,* treating both operands as signed values. The 64-bit quotient is placed into special register rd.

  No arithmetic exception occurs under any circumstances.

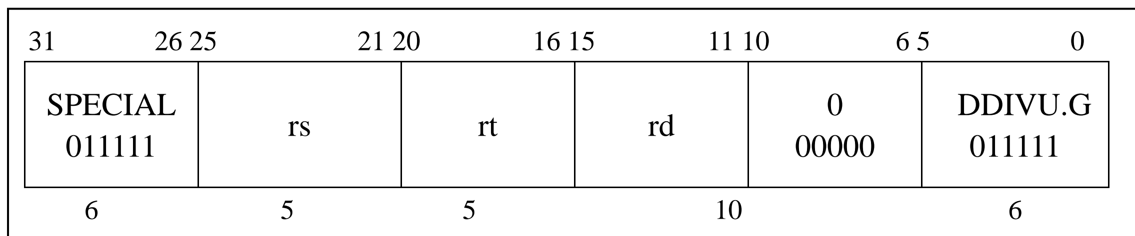
- **Operation:**

  rd ← GPR[rs] div GPR[rt]

- **Exceptions:**

  None

8. DDIVU.G — Doubleword Divide Unsigned(Godson2)



- **Format:**

DDIVU.G rd, rs, rt

- **Purpose:**

To divide 64-bit unsigned intergers.

- **Description:**

rd ← rs / rt

The 64-bit doubleword in GPR *rs* is divided by the 64-bit doubleword in GPR *rt,* treating both operands as unsigned values. The 64-bit quotient is placed into special register rd.

No arithmetic exception occurs under any circumstances.

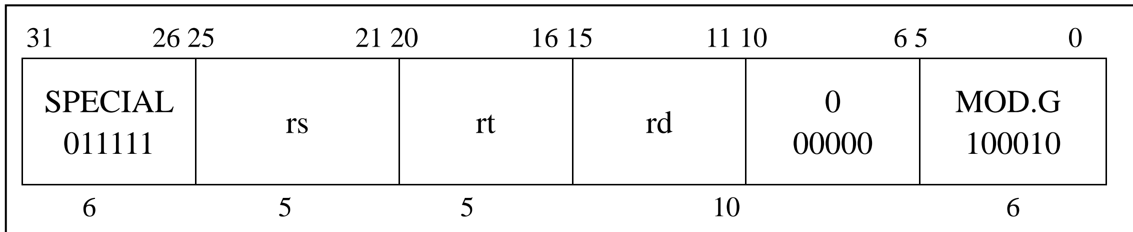
- **Operation:**

rd ← (0 || GPR[rs]) div (0 || GPR[rt])

- **Exceptions:**

None

9. MOD.G —MOD Word (Godson2)



- **Format:**

  MOD.G rd, rs, rt

- **Purpose:**

  To mod 32-bit signed intergers.

- **Description:**

  rd ← rs % rt

  The 32-bit word value in GPR *rs* is divided by the 32-bit value in GPR *rt,* treating both operands as signed values. The 32-bit remainder is placed into special register rd.

  No arithmetic exception occurs under any circumstances.

- **Operation:**

  $q \leftarrow GPR[rs]_{31..0} \bmod GPR[rt]_{31..0}$

  $HI \leftarrow sign_extend(q_{31..0})$

- **Exceptions:**

  None

10.MODU.G — Mod Unsigned Word (Godson2)

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 011111	rs	rt	rd	0 00000	MODU.G 100011
6	5	5	10		6

- **Format:**

  MODU.G rd, rs, rt

- **Purpose:**

  To mod 32-bit unsigned intergers.

- **Description:**

  rd ← rs % rt

  The 32-bit word value in GPR rs is divided by the 32-bit value in GPR rt, treating both operands as unsigned values. The 32-bit remainder is placed into special register rd.

  No arithmetic exception occurs under any circumstances.

- **Operation:**

  q ← (0 || GPR[rs]$_{31..0}$) mod (0 || GPR[rt]$_{31..0}$)

  rd ← sign_extend(q$_{31..0}$)

- **Exceptions:**

  Reserved Instruction

11.DMOD.G — Doubleword Mod (Godson2)

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 011111	rs	rt	rd	0 00000	DMOD.G 100110
6	5	5	10		6

- **Format:**

  DMOD.G rd, rs, rt

- **Purpose:**

  To mod 64-bit signed intergers.

- **Description:**

  rd ← rs % rt

  The 64-bit doubleword in GPR *rs* is divided by the 64-bit doubleword in GPR *rt,* treating both operands as signed values. The 64-bit remainder is placed into special register rd.

  No arithmetic exception occurs under any circumstances.

- **Operation:**

  rd ← GPR[rs] mod GPR[rt]

- **Exceptions:**

  None

12.DMODU.G — Doubleword Mod Unsigned(Godson2)

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 011111	rs	rt	rd	0 00000	DMODU.G 100111
6	5	5	10		6

- **Format:**

  DMODU.G rd, rs, rt

- **Purpose:**

  To mod 64-bit unsigned intergers.

- **Description:**

  rd ← rs % rt

  The 64-bit doubleword in GPR *rs* is divided by the 64-bit doubleword in GPR *rt,* treating both operands as unsigned values. The 64-bit remainder is placed into special register *rd*.

  No arithmetic exception occurs under any circumstances.

- **Operation:**

  rd ← (0 || GPR[rs]) mod (0 || GPR[rt])

- **Exceptions:**

  None

AppendixB Godson new float-point instructions

1. MADD.fmt— Floating-Point Multiply Add

31 — 26	25 — 21	20 — 16	15 — 11	10 — 6	5 — 0
COP1 010001	Fmt	ft	fs	fd	MADD 011000
6	5	5	5	5	6

- **Format:**

  MADD.S fd, fs, ft

  MADD.D fd, fs, ft

- **Purpose:**

  To perform a combined multiply-then-add of FP values.

- **Description:**

  fd ← ((fs * ft) + fd)

  The value in FPR fs is multiplied by the value in FPR ft to produce a product. The value in FPR fd is added to the product. The result sum is calculated to infinite precision, rounded according to the current rounding mode in FCSR, and placed into FPR fd. The operands and result are values in format fmt.

- **Operation:**

  vfd ← ValueFPR(fd, fmt)

  vfs ← ValueFPR(fs, fmt)

  vft ← ValueFPR(ft, fmt)

  StoreFPR(fd, fmt, vfd + vfs * vft)

- **Exceptions:**

  Coprocessor Unusable

  Reserved Instruction

  Floating-Point

  Inexact Unimplemented Operation   Unimplemented Operation

  Invalid Operation Overflow   Overflow

  Underflow

2. MSUB.fmt— Floating-Point Multiply Subtract

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	Fmt	ft	fs	fd	MSUB 011001
6	5	5	5	5	6

- **Format:**

  MSUB.S fd, fs, ft

  MSUB.D fd, fs, ft

- **Purpose:**

  To perform a combined multiply-then-subtract of FP values.

- **Description:**

  fd ← (fs * ft) - fd

  The value in FPR fs is multiplied by the value in FPR ft to produce an intermediate product. The value in FPR fd is subtracted from the product. The subtraction result is calculated to infinite precision, rounded according to the current rounding mode in FCSR, and placed into FPR fd. The operands and result are values in format fmt.

- **Operation:**

  vfd ← ValueFPR(fd, fmt)

  vfs ← ValueFPR(fs, fmt)

  vft ← ValueFPR(ft, fmt)

  StoreFPR(fd, fmt, (vfs * vft)-vfd)

- **Exceptions:**

  Coprocessor Unusable

  Reserved Instruction

  Floating-Point

  Inexact Unimplemented Operation Unimplemented Operation

  Invalid Operation Overflow Overflow

  Underflow

3. NMADD.fmt— Floating-Point Negative Multiply Add

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	Fmt	ft	fs	fd	NMADD 011010
6	5	5	5	5	6

- **Format:**

NMADD.S fd, fs, ft

NMADD.D fd, fs, ft

- **Purpose:**

To negate a combined multiply-then-add of FP values.

- **Description:**

fd ←- ((fs * ft) + fd)

The value in FPR fs is multiplied by the value in FPR ft to produce an intermediate product. The value in FPR fd is added to the product. The result sum is calculated to infinite precision, rounded according to the current rounding mode in FCSR, negated by changing the sign bit, and placed into FPR fd. The operands and result are values in format fmt.

- **Operation:**

vfd ← ValueFPR(fd, fmt)

vfs ← ValueFPR(fs, fmt)

vft ← ValueFPR(ft, fmt)

StoreFPR(fd, fmt, -(vfd + vfs * vft))

- **Exceptions:**

Coprocessor Unusable

Reserved Instruction

Floating-Point

Inexact Unimplemented Operation Unimplemented Operation

Invalid Operation Overflow Overflow

Underflow

4. NMSUB.fmt— Floating-Point Negative Multiply Subtract

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	Fmt	ft	fs	fd	NMSUB 011011
6	5	5	5	5	6

- **Format:**

NMSUB.S fd, fs, ft

NMSUB.D fd, fs, ft

- **Purpose:**

To negate a combined multiply-then-subtract of FP values.

- **Description:**

fd ← -((fs * ft) - fd)

The value in FPR fs is multiplied by the value in FPR ft to produce an intermediate product. The value in FPR fd is subtracted from the product. The result is calculated to infinite precision, rounded according to the current rounding mode in FCSR, negated by changing the sign bit, and placed into FPR fd. The operands and result are values in format fmt.

- **Operation:**

vfd ← ValueFPR(fd, fmt)

vfs ← ValueFPR(fs, fmt)

vft ← ValueFPR(ft, fmt)

StoreFPR(fd, fmt, -((vfs * vft)-vfd))

- **Exceptions:**

Coprocessor Unusable

Reserved Instruction

Floating-Point

Inexact Unimplemented Operation   Unimplemented Operation

Invalid Operation Overflow   Overflow

Underflow