UM0447
User manual

STLS2F01

Introduction

STLS processors are based on the Loongson CPU architecture licensed by STMicroelectronics from the Institute of Computing Technology (ICT), which is part of the Chinese Academy of Science. The STLS family is made up of 64-bit high-end processors that target applications requiring high levels of performances efficiency in terms of cost, power consumption and area.

The Loongson CPU architecture is MIPS III based and enables multiple levels of parallelism to boost computing performance. At instruction level, out-of-order execution and superscalar techniques allow STLS processors to schedule the execution of instructions for maximum throughput. At data level, STLS processors generate multiple results with single vector instructions using SIMD techniques.

STLS processors implement a superscalar, out-of-order execution pipeline with dynamic branch prediction and non-blocking cache. They also implement some fixed-point SIMD instructions by reusing the floating-point data paths.

Contents

List of tables

List of figures

1 STLS2F01 microprocessor architecture

1.1 STLS2F01 microprocessor architecture overview

The STLS2F01 processor is an enhanced version of its earlier STLS2E02 cousin, and a four-issue general-purpose RISC SoC. Its main architectural improvements include an integrated DDR2 Memory Controller and a 133MHz PCI-X interface. The STLS2F01 is fabricated with STMicroelectronics 90nm technology, and runs at a 1GHz or higher main clock frequency.

STLS2F01 employs out-of-order execution and aggressive memory hierarchy design to maximize pipeline efficiency.

Out-of-order execution is accomplished with a combination of register renaming, dynamic scheduling, and branch prediction techniques. The result is fewer pipeline stalls caused by WAR (write after read) and WAW (write after write) hazards, RAW (read after write) hazards, and control hazards. The STLS2F01 has a 64-entry physical register file for fixed- and floating-point register renaming, a 16-entry fixed-point reservation station, and a 16-entry floating-point reservation station that is responsible for out-of-order instruction issuing. 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) are used to record branch history information.

The STLS2F01 memory hierarchy is also engineered for high performance. There is a 64KB instruction cache, a 64KB data cache, and a 512KB level-two cache; all four-way set associative. The on-chip DDR2 memory controller implements the JESD79-2B standard and allows the STLS2F01 to achieve high memory bandwidth with low latency. The fully associative Translation Lookahead Buffer (TLB) has 64 entries, each mapping an odd and even page. A 24-entry memory access queue contains a content-addressable memory for dynamic memory disambiguation and allows the STLS2F01 to implement out-of-order memory access, non-blocking cache, load speculation, and store forwarding.

The STLS2F01 has two fixed-point functional units, two floating-point functional units, and one memory access unit. The floating-point units can also execute 32- or 64-bit fixed-point instructions and 8- or 16-bit SIMD fixed-point instructions through extension of the fmt field of the floating-point instructions.

The basic pipeline stages of the STLS2F01 include instruction fetch, pre-decode, decode, register rename, dispatch, issue, register read, execution, and commit. *Figure 1* shows major sections of the STLS2F01.

- **Fetch Stage:** The instruction cache and TLB are read, according to the contents of the program counter (PC). Four new instructions are sent to the instruction register (IR) if the instruction fetch is a TLB hit and a cache hit.
- **Pre-Decode Stage:** Branch instructions are found and their branch directions are dynamically predicted.
- **Decode Stage:** The four instructions in IR are decoded in the STLS2F01's internal format and sent to the register renaming module.
- **Register Rename Stage:** A new physical register is allocated for each logical destination register, and the logical source register is renamed according 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 to be sent to reservation stations and queues in next cycle.

- **Dispatch Stage:** Renamed instructions are dispatched to the fixed- or floating-point reservation station to be 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.
- **Issue Stage:** One instruction with all required operands ready is selected from the fixed- 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.
- **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.
- **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.
- **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.

1.2 Fetching and decoding

The STLS2F01 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 STLS2F01 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 the STLS2F01 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. STLS2F01 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.

1.3 Register renaming

The STLS2F01 implements two 64-entry physical register file for fixed-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.

1.4 Issuing and reading operands

Register renamed instructions are latched and then sent to the reservation station to be scheduled for execution. STLS2F01 has two independent group reservation stations. Fixed-point and memory instructions are sent to the fixed-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 fixed-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. STLS2F01 has one fixed-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 fixed-point register file has three write ports and seven read ports. The ALU1 fixed-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.

1.5 Execution and functional units

Instructions are sent to functional or memory units for execution after reading operations. STLS2F01 has two fixed-point functional units ALU1 and ALU2, and two floating-point functional units FALU1 and FALU2.

The ALU1 unit executes fixed-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 fixed-point addition, subtraction, logical, shift, comparison, multiplication, and division instructions. Fixed-point multiplication is fully pipelined and has a latency of four cycles. Fixed-point division uses the SRT algorithm and is not fully pipelined, the latency of fixed-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 instructions is two cycles. The latency of conversion instructions is four cycles. The latency of floating-point addition, subtraction, multiplication, multiplication and accumulation instructions is 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 instructions is six cycles. The division and square root instructions use the SRT algorithm and are not fully pipelined. The latency of single/double precision floating-point division instructions ranges from 4 to 10/17 cycles, while the latency of floating-point division instructions 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 execute C±(A×B) as a single instruction with no intermediate rounding. The standard floating-point add and floating-point multiply operations can be performed using this FMAF unit by making B=1 for addition and C=0 for multiplication. The STLS2F01's FALU1 and FALU2 floating point units both 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 multiplications, 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 fixed-point instruction through extension of the fmt field of the floating-point instructions.

1.6 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.

1.7 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. STLS2F01 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.

1.8 Memory subsystem

Memory references are issued out-of-order to the address calculation unit. The STLS2F01 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, STLS2F01 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. STLS2F01 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 STLS2F01 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 STLS2F01 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

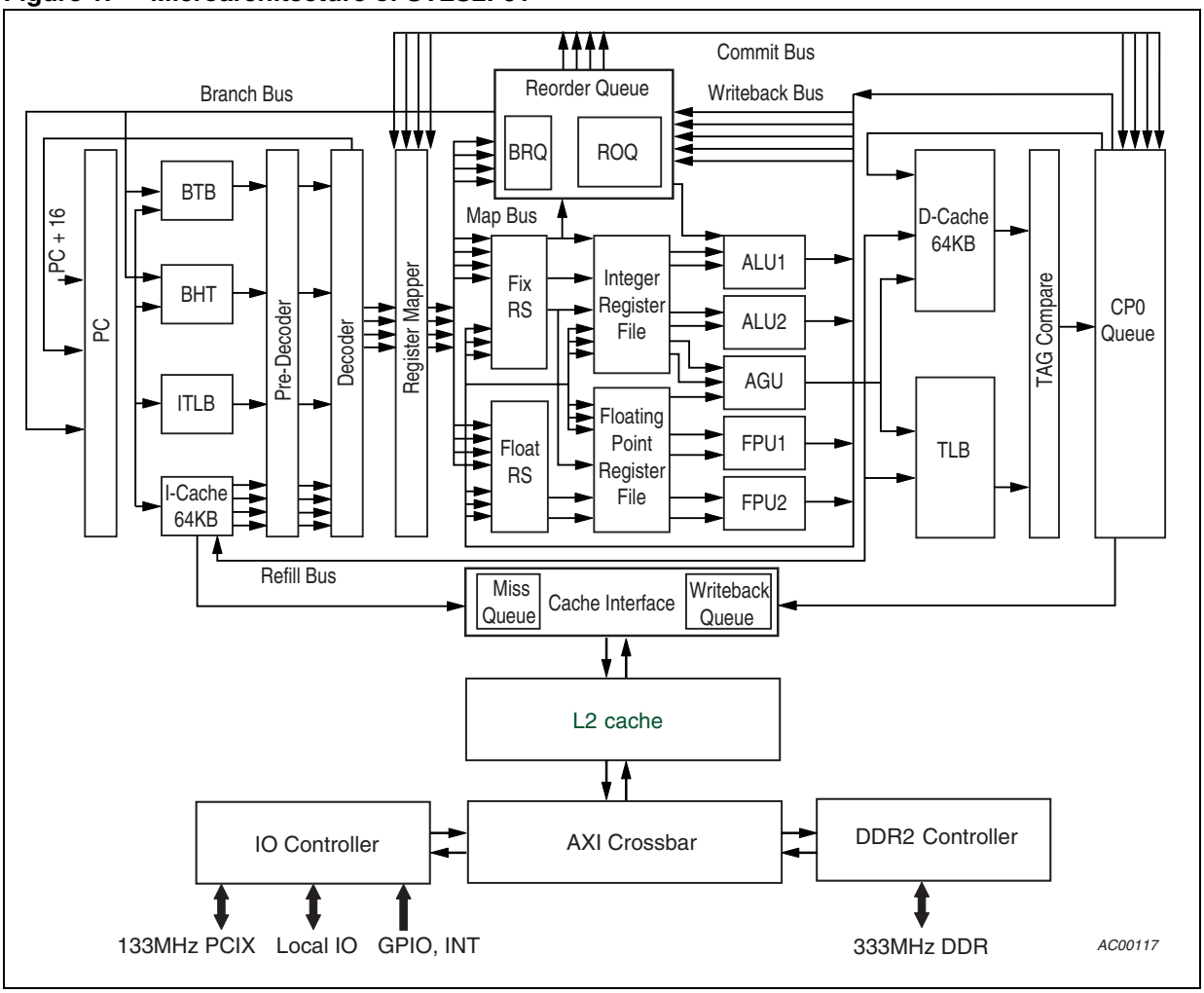
directly to lower level memory hierarchy. The fully pipelined L2 cache of STLS2F01 runs at the same frequency as the processor core and has an access latency of five cycles.

The on-chip DDR2 memory controller allows STLS2F01 to achieve high memory bandwidth with low latency. The STLS2F01 CPU support 4 physical memory bank at most (implemented by 4 chip select signal), with address bus of 18 bits (ddr2_a[14:0] and ddr2_bank[2:0]).The integrated DDR2 controller of STLS2F02 CPU supports the dynamic page management. The DDR2 controller decides the Open Page or Close Page all by the hardware for the memory access, but not by the software designer.

1.9 The STLS2F01 processor summary

The STLS2F01 processor is a 64-bit, four-issue RISC SOC microprocessor which is MIPS based. It implements the advanced out-of-order executions technologies (i.e. register renaming, dynamic scheduling, and branch prediction) and Cache technologies (i.e. non-blocking cache, load speculation, and store forwarding). It also integrates on-chip second-level cache, DDR2 memory controller and I/O controller to enhance the pipeline efficiency and I/O ability.

Figure 1. Microarchitecture of STLS2F01

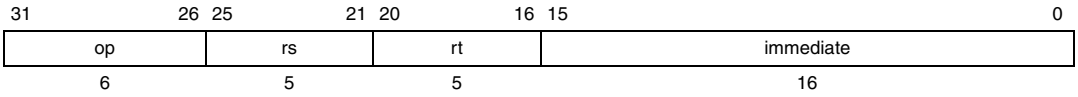


2 Instruction set overview

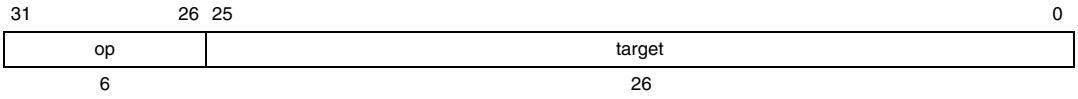
Each CPU instruction consists of a single 32-bit word, aligned on a word boundary. There are three instruction formats-immediate (I-type), jump (J-type), and register (R-type)-as shown in *Section 2.1*. The use of a small number of instruction formats simplifies instruction decoding, allowing the compiler to synthesize more complicated (and less frequently used) operations and addressing modes from these three formats as needed.

2.1 CPU instruction formats

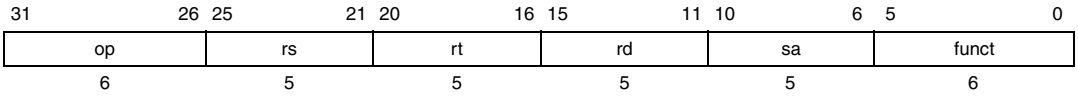
I-Type (immediate)



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

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. STLS2F01 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 high order 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. STLS2F01 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 9* .
  Coprocessor 1 instructions include float pointer instructions, multi-media extended instructions and Loongson-extended fixed pointer computational instructions. They all operate on float pointer registers. *Chapter 8* provides summary of these instructions and *Appendix B* 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.

Table 1 through *Table 9* lists all prior instructions except coprocessor 1 instructions.

Table 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

Table 1. CPU instruction set: load and store instructions (continued)

OpCode	Description	MIPS ISA
SDR	Store Doubleword Right	III
SC	Store Conditional	I
SCD	Store Conditional Double	III
SYNC	Sync	I

Table 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 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 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	STLS2F01 Multiply	STLS2F01
DMULTG	STLS2F01 Doubleword Multiply	STLS2F01
MULTUG	STLS2F01 Multiply unsigned	STLS2F01
DMULTUG	STLS2F01 Doubleword Multiply Unsigned	STLS2F01
DIVG	STLS2F01 Divide	STLS2F01
DDIVG	STLS2F01 Doubleword Divide	STLS2F01
DIVUG	STLS2F01 Divide unsigned	STLS2F01
DDIVUG	STLS2F01 Doubleword Divide Unsigned	STLS2F01
MODG	STLS2F01 Modulus	STLS2F01
DMODG	STLS2F01 Doubleword Modulus	STLS2F01
MODUG	STLS2F01 Modulus Unsigned	STLS2F01
DMODUG	STLS2F01 Doubleword Modulus Unsigned	STLS2F01

Table 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

Table 5. CPU instruction set: jump and branch instructions (continued)

Opcode	Description	MIPS ISA
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 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 7. CPU instruction set: special instructions

OpCode	Description	MIPS ISA
SYSCALL	System Call	I
BREAK	Break	I

Table 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 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 STLS2F01 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). [(a)] The primary TLB is the Joint TLB (JTLB). In addition, the STLS2F01 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 STLS2F01 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 one specific page size at the run time. In the future, STLS2F01 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 STLS2F01 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.

a. 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].

3.1.2 Instruction TLB

The STLS2F01 uses an 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 STLS2F01 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 STLS2F01 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 10* 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 10. 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

STLS2F01 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

STLS2F01 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 STLS2F01 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

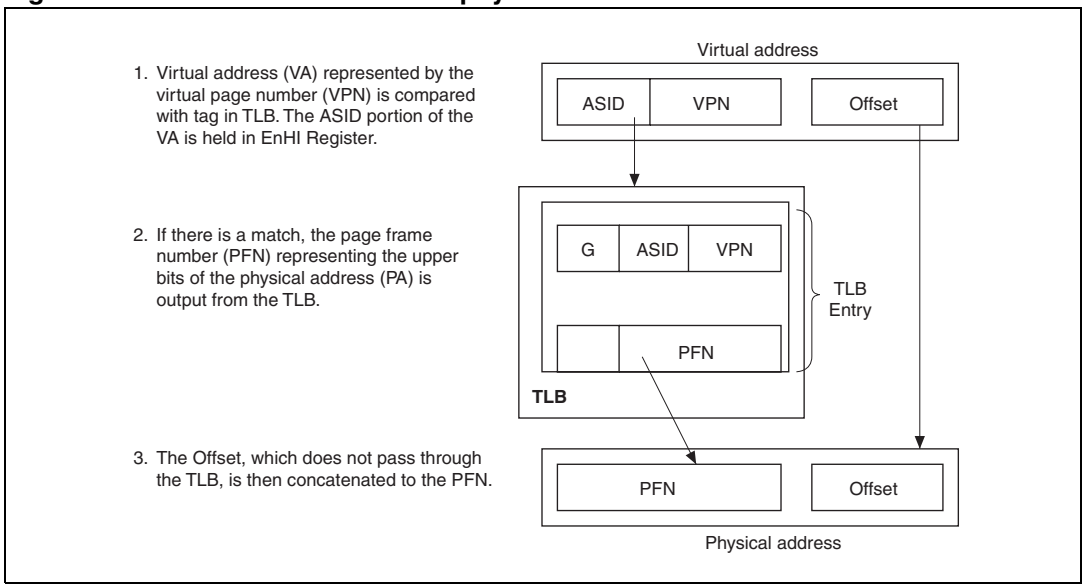
The STLS2F01 processor has three virtual address spaces: User address space, Supervisor address space and Kernel address space. Each space is 64-bit and consists of several discontinued segments. The maximum segment size is 1 terabyte(240).

3.2.7 Physical address space

Using a 36-bit address, the STLS2F01 processor physical address space encompasses 64 gigabytes.

3.2.8 Virtual-to-physical address translation

Figure 2. 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 2 shows 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. 64-bit mode virtual address translation

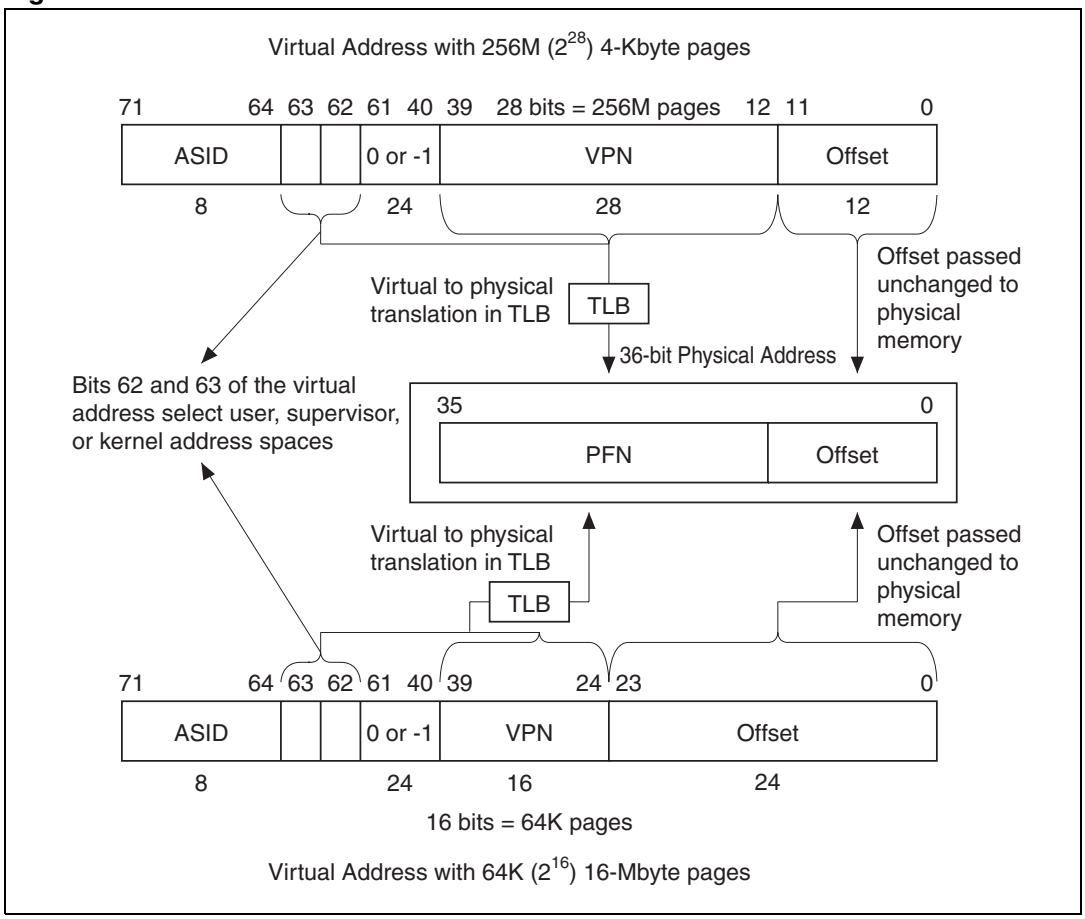


Figure 3 shows the 64-bit mode virtual-to-physical-address translation. This figure illustrates the two extremes in the range of possible page sizes: a 4-Kbyte page (12 bits) and a 16-Mbyte page (24 bits).

The top portion of *Figure 3* 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* 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.

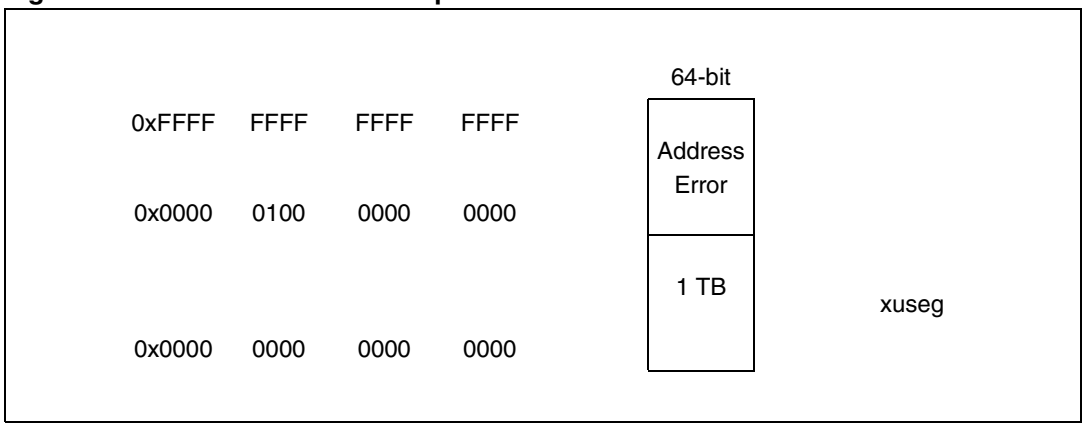
3.2.9 User address space

In User address space, a single, uniform virtual address space-labeled Extended User segment (xuseg), is available and its size is 1 terabyte (240 bytes) .

Figure 4 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 4. 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 STLS2F01 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 bit values:

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

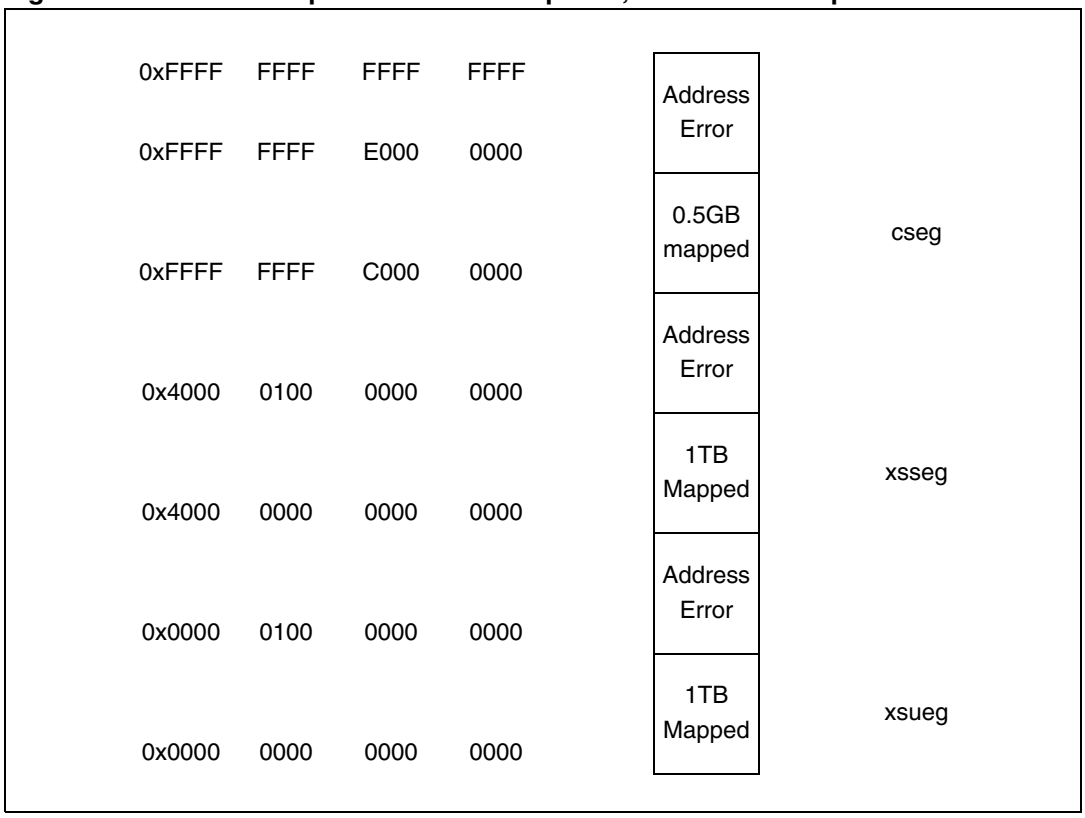
Figure 5 shows the User and Supervisor address spaces viewed from Supervisor mode.

64-bit supervisor, user space (xsuseg)

In Supervisor Mode when accessing User space and bits 63:62 of the virtual address are set to 002, 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.

Figure 5. 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 012, 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.

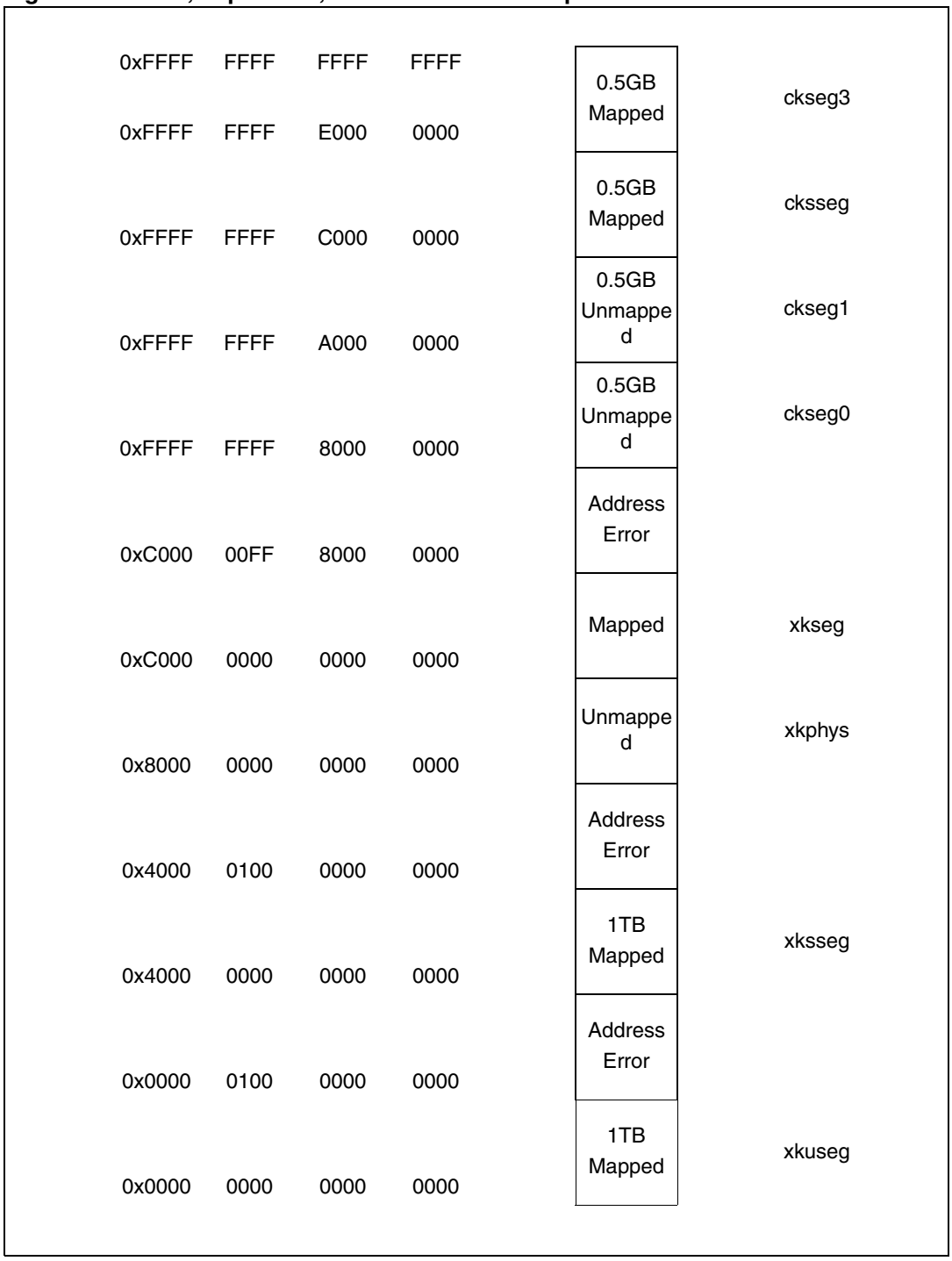
64-bit supervisor, separate supervisor space (csseg)

In Supervisor space, when bits 63:62 of the virtual address are set to 112, 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

Figure 6. 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 6* 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 002, 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 012, 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 102, the xkphys virtual address space is selected; it is a set of eight 236-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 *Figure 3*.

64-bit kernel, kernel space (xkseg)

In Kernel space, when bits 63:62 of the 64-bit virtual address are 112, 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 112, 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.

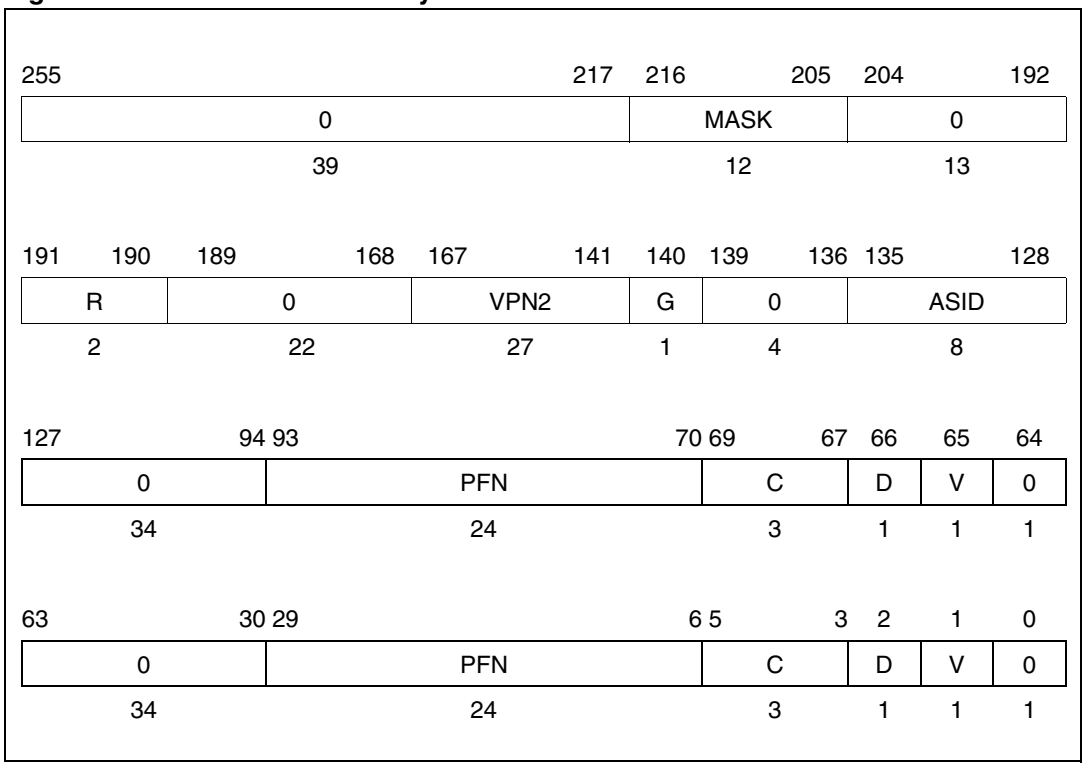
3.3 System control coprocessor

The System Control Coprocessor (CP0) supports memory management, address translation, exception handling, and other privileged operations. STLS2F01 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

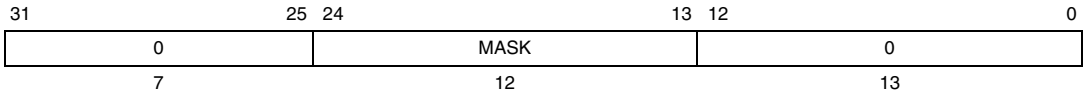
Figure 7 shows the TLB entry formats. Each field of an entry has a corresponding field in the EntryHi, EntryLo0, EntryLo1, or PageMask registers.

Figure 7. 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.

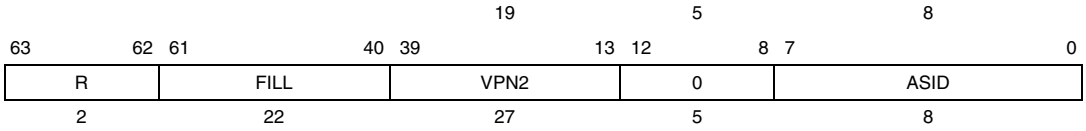
EntryHi register format



Mask.. ..... Page comparison mask

0..... ..... Reserved. Must be written as zeros, and return zereos when read

PageMask register format



VPN2.. Virtual page number divided by two (maps to two pages).

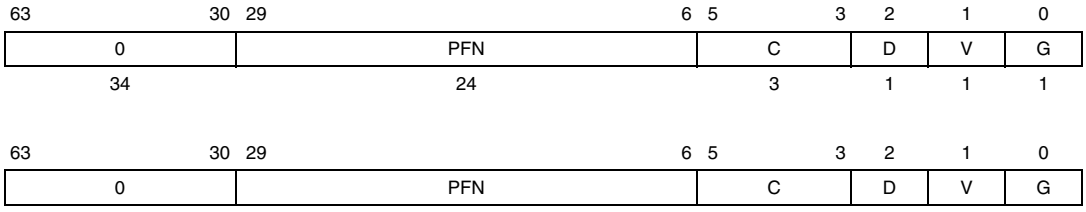
ASID Address space ID field. An 8-bit field that lets multiple processes share the TLB; each process has a distinct mapping of otherwise identical virtual page numbers,

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 zeros, and return zeros when read.

EntryLo0 and EntryLo1 register formats



PFNPage frame number, the upper bits of the physical address.

C Specifies the TLB page coherency 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 TLB 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 zeros, and return zeros when read

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 11* shows the coherency attributes selected by the ***C*** bits.

Table 11. 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

Figure 4 lists the CP0 registers used by the MMU, *Chapter 6* provides complete description of each CP0 registers.

Table 12. 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

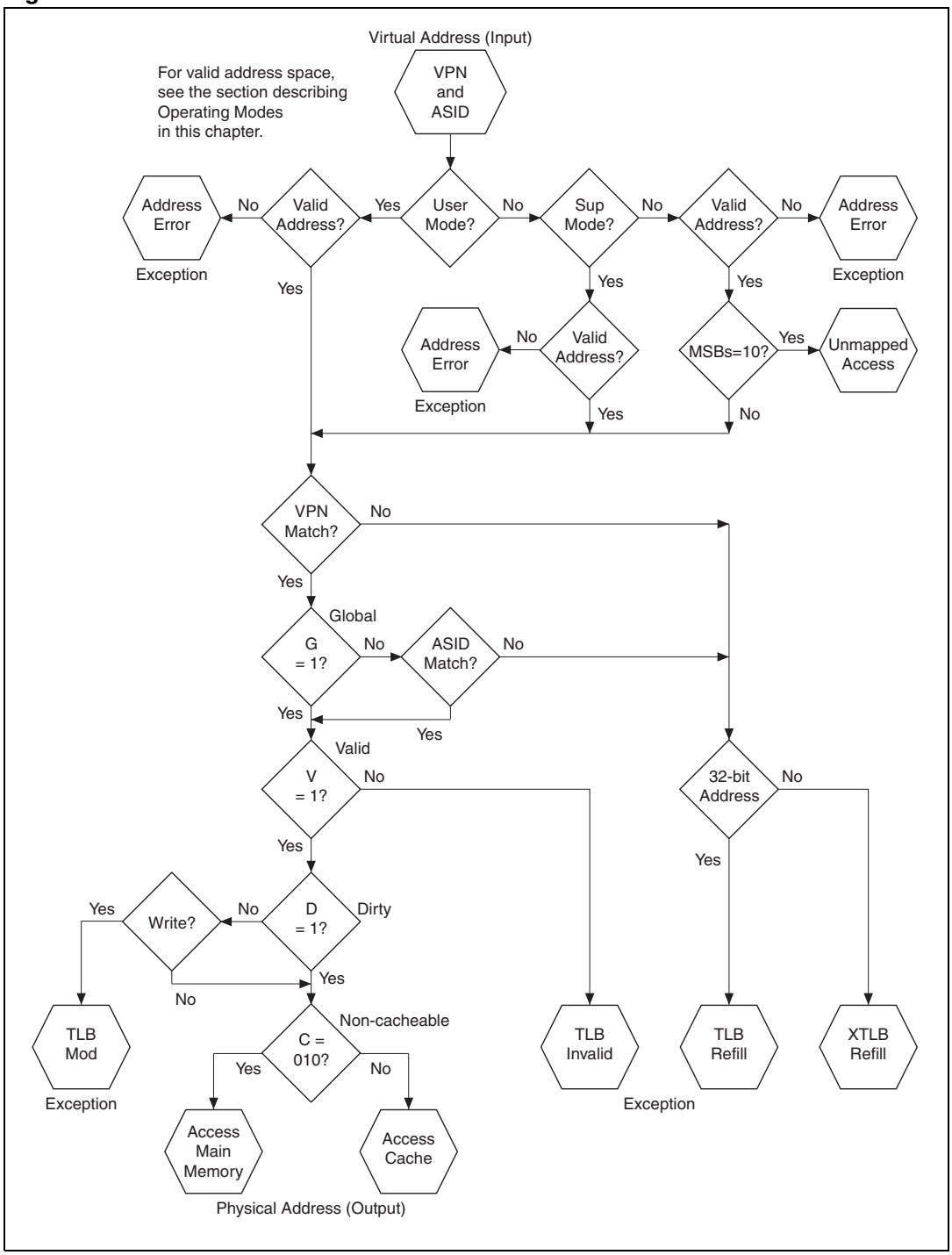
During virtual-to-physical address translation, the CPU compares the 8-bit ASID (if the Global bit, G, is not set) of the virtual address to the ASID of the TLB entry to see if there is a match. And the highest 15-to-27 bits (depending upon the page size) of the virtual address are compared to the contents 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 8 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 0102, the physical address that is retrieved accesses main memory, bypassing the cache.

Figure 8. TLB address translation



3.3.5 TLB instructions

Figure 5 lists the instructions that the CPU provides for working with the TLB.

Table 13. 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 on demand 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 STLS2F01 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 an 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 STLS2F01 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 STLS2F01 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 STLS2F01 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 STLS2F01 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 14 shows the attributes for the three caches.

Table 14. Attributes for the three caches

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

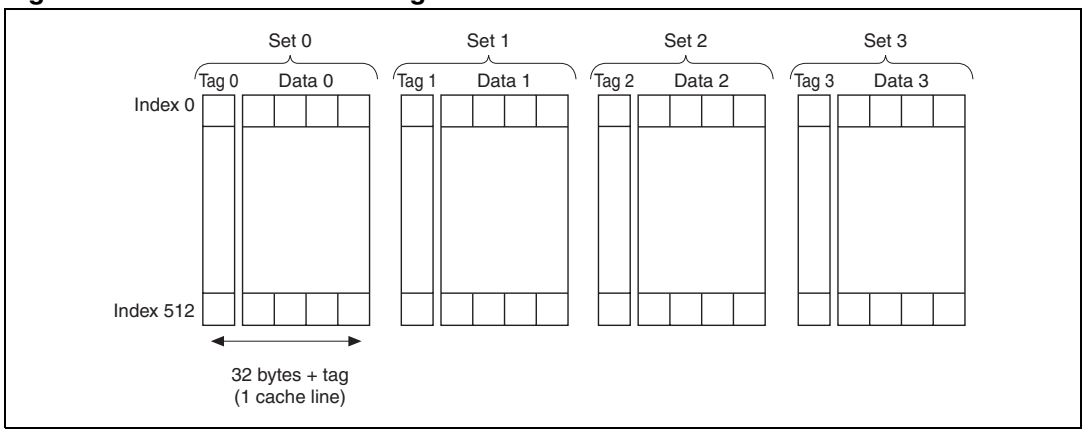
The primary instruction cache is 64 Kbytes in size and implements a 4-way set associative architecture. Line size is 32-bytes, or eight instructions. The 128-bit read path allows the STLS2F01 to fetch four instructions per clock cycle which are passed to the superscalar dispatch unit.

4.2.1 Instruction cache organization

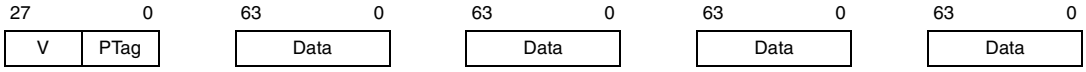
The instruction cache is organized as shown in *Figure 9*. The cache is 4-way set associative and contains 512 indexed locations. Each time the cache is indexed, the tag and data portion of each set are accessed. Each of the four tag addresses are compared against the translated portion of the virtual address to determine which set contains the correct data.

When the instruction cache is indexed, each of the four sets shown in *Figure 9* returns a single cache line. Each cache line consists of 32 bytes of data, a 28-bit physical tag address, and 1 valid bit. Paragraph *Instruction cache line format* shows the instruction cache line format.

Figure 9. Instruction cache organization



Instruction cache line format

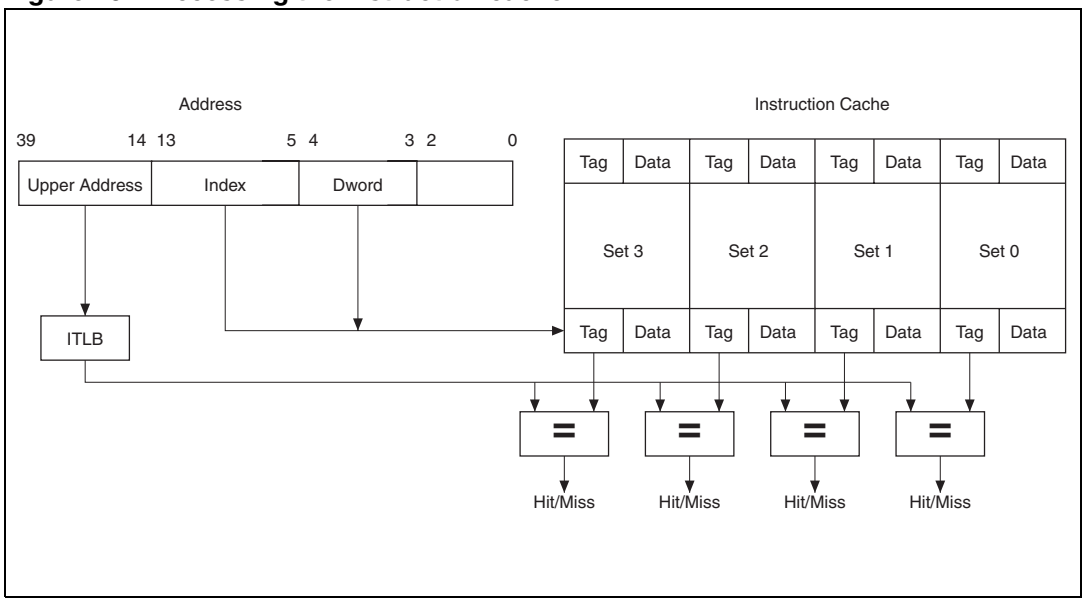


V	Tag valid bit
PTag	28-bit physical address tag (bits 39:12 of the physical address)
Data	Cache data

4.2.2 Accessing instruction cache

The STLS2F01 implements a 4-way set associative cache that is virtually indexed and physically tagged. *Figure 10* shows how the virtual address is divided on an instruction cache access.

Figure 10. Accessing the instruction cache



The lower 14 bits of address are used for indexing the instruction cache as shown in *Figure 10*. 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.

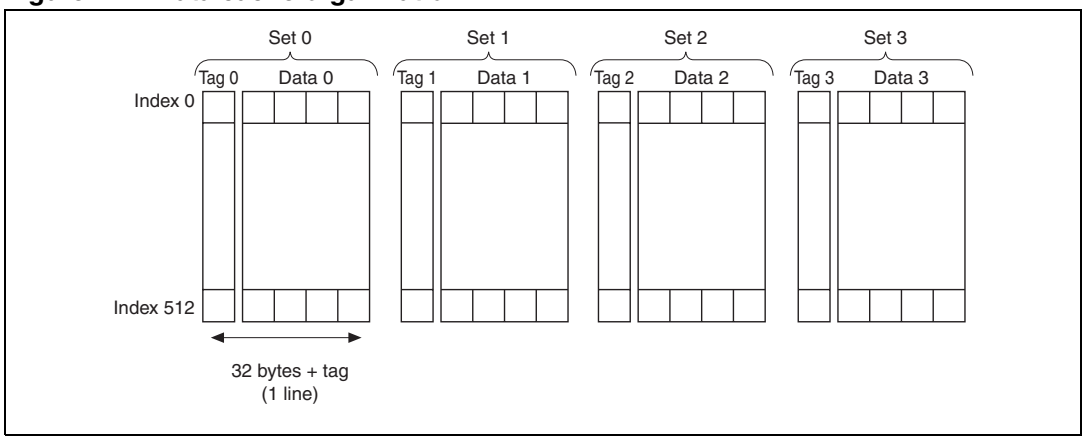
The data cache is virtually indexed and physically tagged. Operating system helps eliminating the potential for virtual aliasing. The data cache is non-blocking, meaning that a miss in the data cache does not stall the pipeline.

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

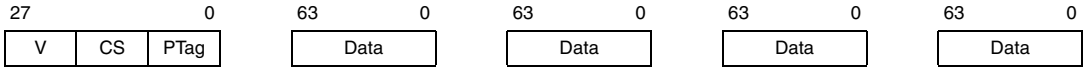
The data cache is organized as shown in *Figure 11*. The cache is 4-way set associative and contains 512 indexed locations. Each time the cache is indexed, the tag and data portion of each set are accessed. Each of the four tag addresses are compared against the translated portion of the virtual address to determine which set contains the correct data.

Figure 11. Data cache organization



When the data cache is indexed, each of the four sets shown in *Figure 11* 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. See *Data cache line format* paragraph.

Data cache line format

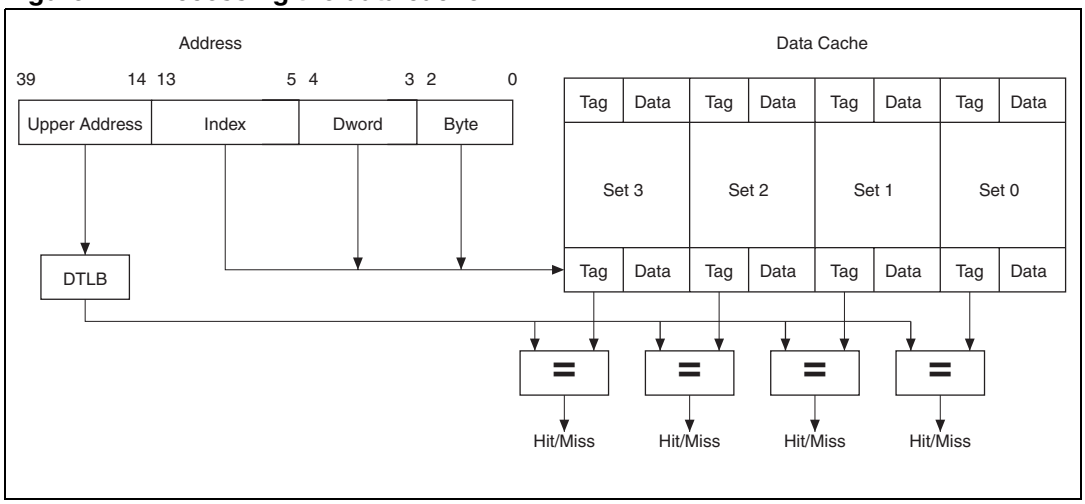


W	Writeback bit (set if cache line has been written)
CS	Primary cache state 00 = INVALID 01 = SHARED 10 = EXCLUSIVE 11 = DIRTY
PTag	28-bit physical address tag (bits 39:12 of the physical address)
Data	Cache data

4.3.2 Accessing the data cache

The STLS2F01 implements a 4-way set associative data cache that is virtually indexed and physically tagged. *Figure 12* shows how the virtual address is divided on a data cache access.

Figure 12. Accessing the data cache



The lower 14 bits of address are used for indexing the data cache as shown in *Figure 12*. 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.

STLS2F01 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 STLS2F01 significantly.

4.4 Secondary cache

The STLS2F01 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 STLS2F01 supports secondary cache sizes of 512Kbytes.

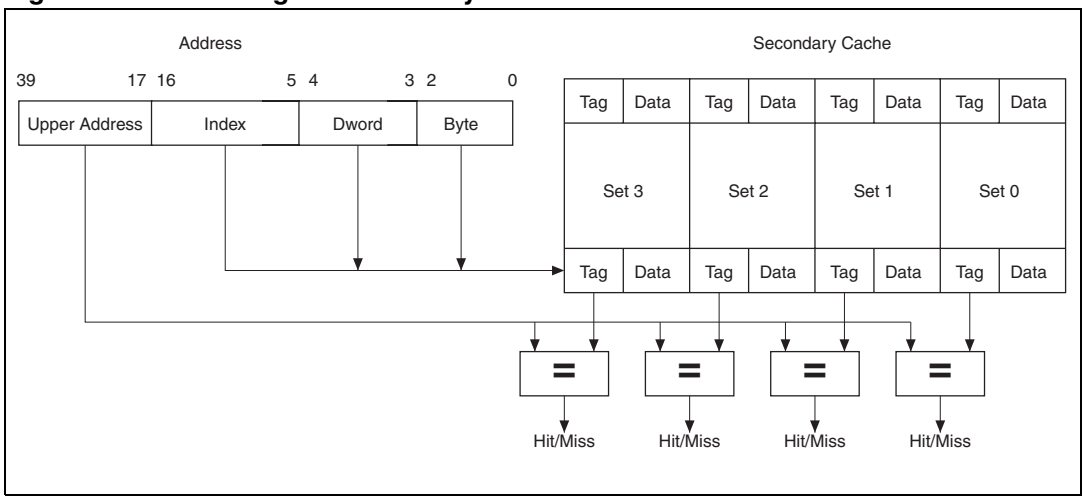
Each indexed location in the cache contains four 64-bit doublewords. Each time the cache is indexed, the tag and data portion of each set are accessed. The tag address is compared against the translated portion of the virtual address to determine if the data resides in the cache.

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 13*.

Figure 13. Accessing the secondary cache



The lower bits of address are used for indexing the data cache as shown in *Figure 13*. 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 STLS2F01 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 15*.

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 15. STLS2F01 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 15*.

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 STLS2F01 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 STLS2F01 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 store is intervened by other uncached stores, individual uncached stores are executed for the buffer content.

Uncached accelerated attributes can accelerate sequential uncached accesses, which is useful for accessing 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. STLS2F01 contains two “Hit” type cache operations for primary data cache:

- Hit_Invalidate
- Hit_Writeback_Invalidate.

The STLS2F01 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 STLS2F01 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 16*.

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 16. Coprocessor 0 registers

Register No.	Register name	Description
0	Index	Programmable register to select TLB entry for reading or writing
1	Random	Pseudo-random counter for TLB replacement
2	EntryLo0	Low half of TLB entry for even VPN (Physical page number)
3	EntryLo1	Low half of TLB entry for odd VPN (Physical page number)
4	Context	Pointer to kernel virtual PTE table in 32-bit addressing mode
5	PageMask	Mask that decides the TLB page size
6	Wired	Number of wired TLB entries (lowest TLB entries not used for random replacement)
7		Reserved
8	BadVaddr	Bad virtual address
9	Count	Clock counter
10	EntryHi	High half of TLB entry (Virtual page number and ASID)
11	Compare	Clock 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 kernel virtual PTE table in 64-bit addressing mode
21		Reserved
22	Diagnose	Enable/disable BTB, RAS and flush ITLB
23		Reserved

Table 16. Coprocessor 0 registers (continued)

Register No.	Register name	Description
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.

See *Index register* paragraph; *Table 17* describes the Index register fields.

Index register

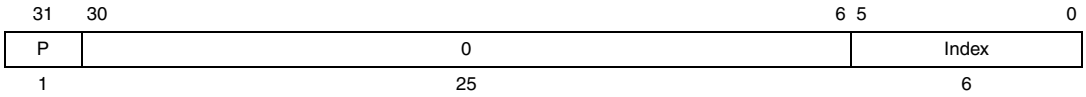


Table 17. Fields in the index register

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

See *Random register* paragraph; *Table 18* describes the Random register fields.

Random register

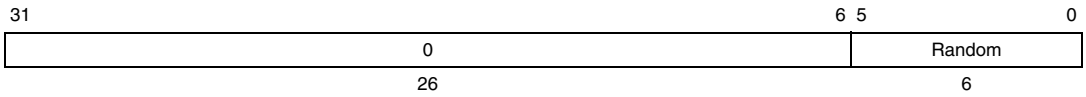


Table 18. Fields in the random register

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

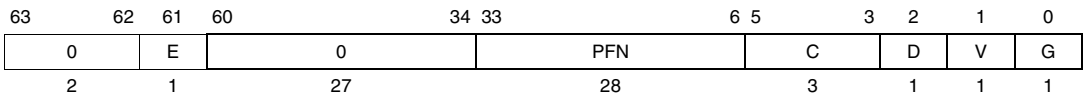
5.3 EntryLo0 (2), and EntryLo1 (3) registers

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. See *Fields of the EntryLo0 and EntryLo1 registers* paragraph.

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 19. Description of EntryLo registers' fields

Field	Description
E	Non-execute1 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 zeros, and returns zeros 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.

See *Context register format* paragraph; *Table 20* describes the Context register fields.

Context register format

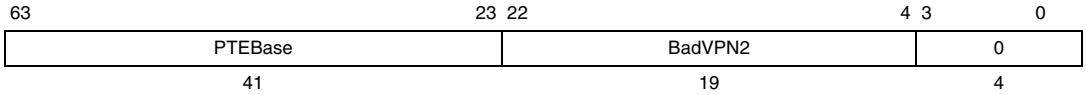


Table 20. Context register fields

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 zeros, and returns zeros 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 21*. Format of the register is shown in *PageMask register format* paragraph.

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 21* the operation of the TLB is undefined. The 0 field is reserved; it must be written as zeroes, and returns zeroes when read.

PageMask register

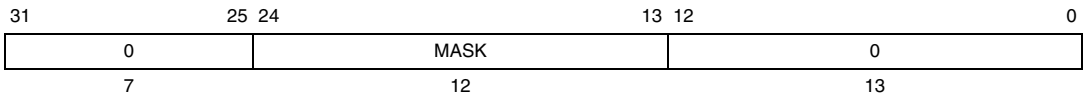


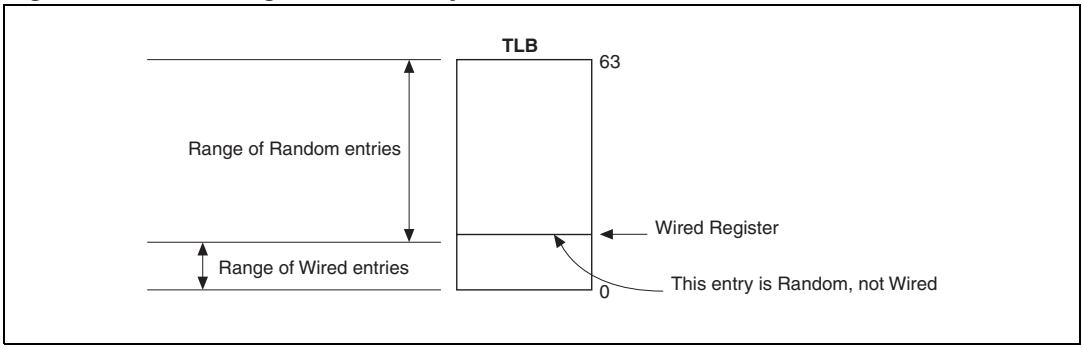
Table 21. Mask field values for page sizes

Page size (Mask)	Bit											
	24	23	22	21	20	19	18	17	16	15	14	13
4 Kbytes	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
16 Mbytes	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 in *Figure 14*. Wired entries are fixed, no replaceable entries, which cannot be overwritten by a TLB write operation. Random entries can be overwritten

Figure 14. 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).

See *Wired register* paragraph; *Table 22* describes the register fields.

Wired register

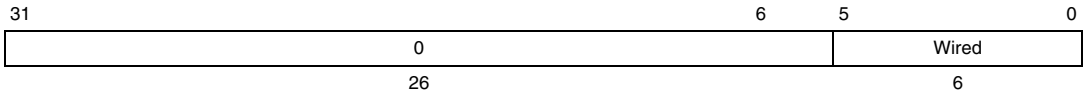


Table 22. Wired register field descriptions

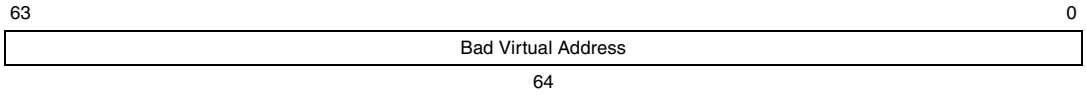
Field	Description
Wired	TLB Wired boundary
0	Reserved. Must be written as zeros, and returns zeros when read.

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.

See *BadVAddr register format* paragraph.

BadVAddr register format



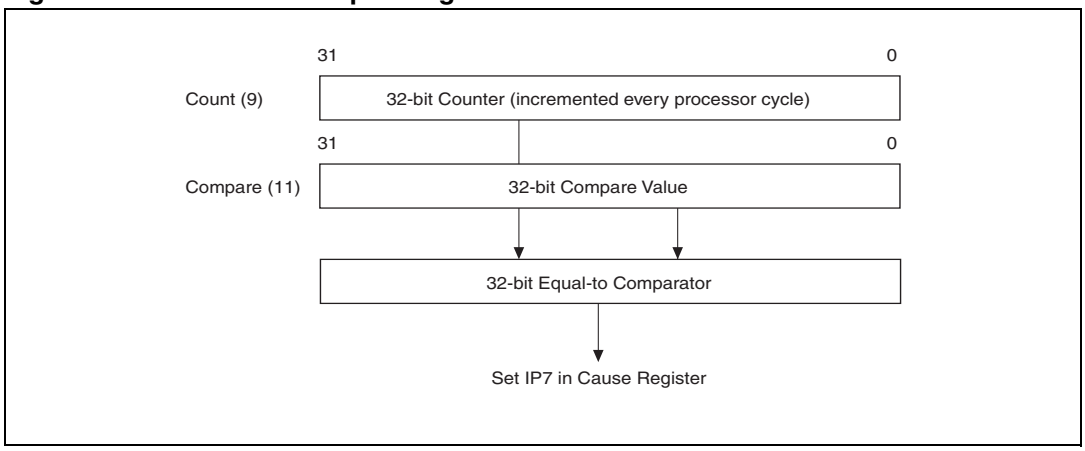
5.8 Count and compare registers (9 and 11)

The Count and Compare registers are 32-bit read/write registers whose formats are shown in *Figure 15*.

The Count register acts as a real-time timer. Like the R4400 implementation, the STLS2F01 Count register is incremented every other PClk cycle. However, unlike the R4400, the STLS2F01 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 15. 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.

EntryHi register paragraph shows the format of this register and *Table 23* describes the register's fields.

EntryHi register

63	62 61	13 12	8 7	0
R	VPN2	0	ASID	
2	49	5	8	

Table 23. 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 zeros, and returns zeros 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; *Status register* 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

31 28	27	26	25	24	23	22	21	20	19 16	15 8	7 5	4 3	2	1	0
CU (cu3:cu0)	0	FR	0	nofdiv	nofsqr	BEV	0	SR	0	IM7-IM0	0	KSU	ERL	EXL	IE
4	1	1	1	1	1	1	1	1	4	8	3	2	1	1	1

5.10.1 Status register format

See *Status register* paragraph. *Table 24* describes the Status register fields.

Table 24. 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 1→bootstrap

Table 24. Fields in the status register (continued)

Field	Description
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

5.10.2 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 = 102, EXL = 0, and ERL = 0.
- The processor is in Supervisor mode when KSU = 012, EXL = 0, and ERL = 0.
- The processor is in Kernel mode when KSU = 002, or EXL = 1, or ERL = 1.

32- and 64-bit Modes: STLS2F01 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.

See *Cause register format* paragraph that shows the fields of this register; *Table 25* describes the Cause register fields. A 5-bit exception code (ExcCode) indicates one of the causes, as listed in *Table 26*.

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.

Cause register format

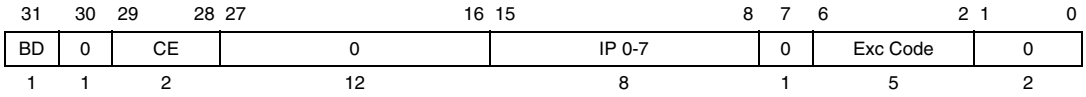


Table 25. Cause register fields

Field	Description
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
0	Reserved. Must be written as zeros, and returns zeros when read.

Table 26. 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)

Table 26. Cause register exccode field (continued)

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.

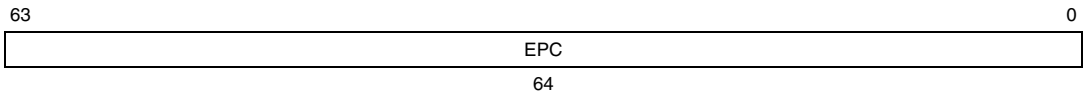
For synchronous exceptions, the EPC register contains either:

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

See *EPC register format* paragraph.

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. See *Processor revision identifier register format* paragraph that shows the format of the PRId register; *Table 27* describes the PRId register fields.

Processor revision identifier register format

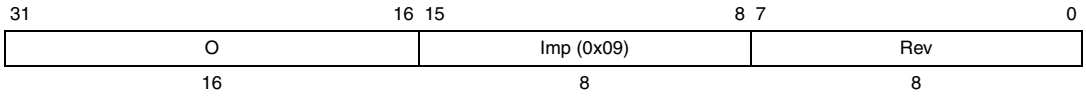


Table 27. PRId register fields

Field	Description
Imp	Implementation number
Rev	Revision number
0	Reserved. Must be written as zeros, and returns zeros 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. “0x63” is the implementation number of the STLS2F01 processor. The revision number is “0x02”. The contents of the high-order half-word (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 STLS2F01 processors; *Table 28* 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.

See *Config register format* paragraph; *Table 28* describes the Config register fields.

Config register format

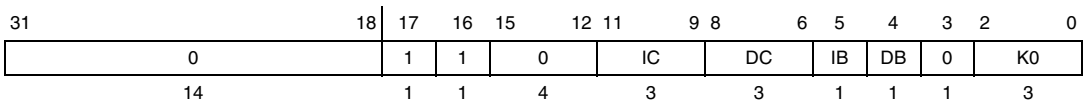


Table 28. Fields in the config register

Field	Description
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 STLS2F01, this bit is set to 1.
DB	Primary D-cache line size 0 →16 bytes 1 →32 bytes In STLS2F01, 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 STLS2F01.

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.

Watch register formats describes the format of the Watch register. *Table 29* describes the fields of the Watch register.

Watch register formats

63	3	2	1	0
VADDR		0	R	W
61		1	1	1

Table 29. 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.

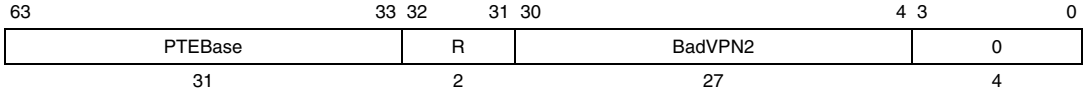
5.17 Xcontext register (20)

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.

See *XContext register format* paragraph; *Table 30* describes the XContext register fields.

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 pair table of 8-byte PTEs. For other page and PTE sizes, shifting and masking this value produces the appropriate address.

Table 30. XContext register fields

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 zeros, and returns zeros 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 STLS2F01 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

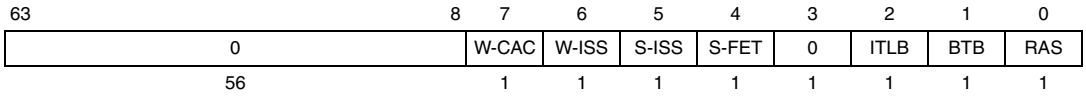


Table 31. Diagnostic register fields

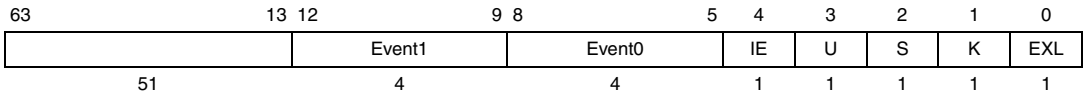
Field	Description
0	Reserved. Must be written as zeroes, and returns zeroes when read.
W-CAC	Cancel the constraint on wait-cache operation.
W-ISS	Cancel the constraint on wait-issue operation.
S-ISS	Cancel the constraint on store-issue operation.
S-FET	Cancel the constraint on store-fetch operation.
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)

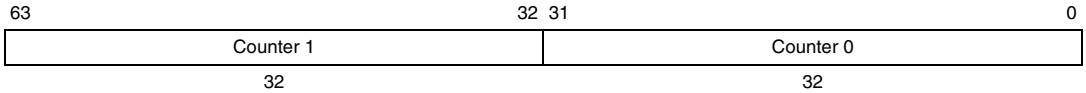
The STLS2F01 processor defines two performance counters, which are mapped into CP0 register 24 and 25. The associated control fields reside in CP0 register 24.

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)



Performance counter register (25)



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 signaled.

See *Performance counter register (24)* and *Performance counter register (25)* paragraphs. *Table 32* describes control fields format and *Table 33* describes count enable bit definition. *Table 34* and *Table 35* describe events of counter 0 and counter 1 respectively.

Table 32. 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 33. 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 34. 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 & Excode=0	Interrupt
1111	Exbus.ex & Excode=63	Internal Exception

Table 35. Counter 1 events

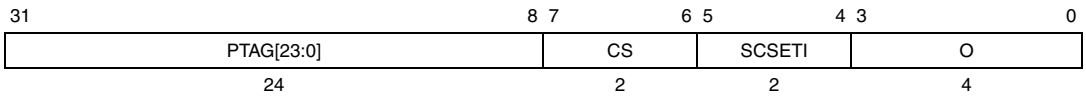
Event	Signal	Description
0000	Cmtbus?.valid	Commit ops
0001	Brbus.brerr	Branch Misprediction
0010	Jrmiss	JR Misprediction
0011	Jr31miss	JR with rs=31 Misprediction
0100	Dmemread.valid& Dmemread_allow	Primary Data cache misses
0101	Rissuebus1.valid	Alu2 op issued
0110	Rissuebus4.valid	Falu2 op issued
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.

TagLo register and *TagLo register* shows the format of these registers for primary cache operations. *Table 36* lists the field definitions of the TagLo and TagHi registers.

TagLo register



TagHi register

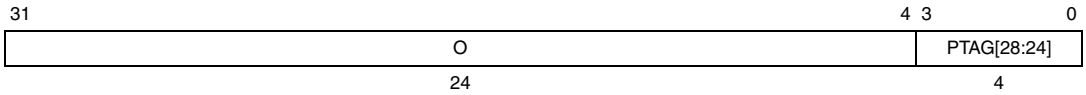


Table 36. Cache tag register fields

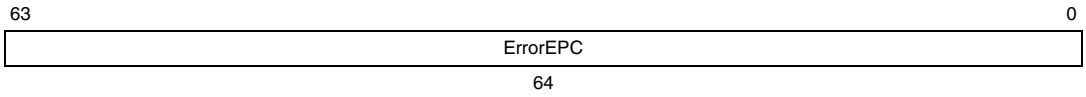
Field	Description
PTAG	Specifies the physical address bits 39:12
CS	Specifies the primary cache state
SCSETI	Specifies the set of secondary cache
0	Reserved. Must be written as zeroes, and returns zeroes when read.

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. See *ErrorEPC register format* paragraph.

ErrorEPC register format



5.22 CP0 instructions

Table 37 lists the CP0 instructions defined for the STLS2F01 processor. Since they are implementation dependent, they are included here and not in the MIPS ISA manual.

Table 37. 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	Probe TLB for matching entry
TLBR	Read indexed TLB entry
TLBWI	Write indexed TLB entry
TLBWR	Write random TLB entry

5.22.1 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 STLS2F01 processor shown in *Table 38*

Table 38. 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 38 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 39* (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 39. 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

6.5.1 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.

6.5.2 Cold reset exception 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.

6.5.3 Cold reset exception 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

6.6.1 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 STLS2F01 processor, there is no way for software to differentiate between a Soft Reset exception and an NMI exception.

6.6.2 Soft reset 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

6.6.3 Soft reset exception 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

6.7.1 NMI exception cause

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

An NMI exception is not maskable.

In STLS2F01 processor, there is no way for software to differentiate between a Soft Reset exception and an NMI exception.

6.7.2 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 38*).

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

6.7.3 NMI exception 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

6.8.1 Address error exceptioncause

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.

6.8.2 Address error exception 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.

6.8.3 Address error exception 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).

Chapter 6.10, *6.11*, *6.12* 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

6.10.1 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.

6.10.2 TLB refill exceptions 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 *Section 3.2.6*). 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.

6.10.3 TLB refill exceptions 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 EntryLo0/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

6.11.1 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.

6.11.2 TLB invalid exception 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.

6.11.3 TLB invalid exception 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

6.12.1 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.

6.12.2 TLB modified exception 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.

6.12.3 TLB modified exception 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

6.13.1 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.

6.13.2 Bus error exception 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.

6.13.3 Bus error exception 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

6.14.1 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.

6.14.2 Integer overflow exception 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.

6.14.3 Integer overflow exception 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

6.15.1 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.

6.15.2 Trap exception 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.

6.15.3 Trap exception 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

6.16.1 System call exception cause

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

6.16.2 System call exception 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.

6.16.3 System call exception 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

6.17.1 Breakpoint exception cause

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

6.17.2 Breakpoint exception 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.

6.17.3 Breakpoint exception 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

6.18.1 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.

6.18.2 Reserved instruction exception 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.

6.18.3 Reserved instruction exception 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

6.19.1 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.

6.19.2 Coprocessor unusable exception 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.

6.19.3 Coprocessor unusable exception 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

6.20.1 Floating-point exception cause

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

6.20.2 Floating-point exception 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.

6.20.3 Floating-point exception servicing

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

6.21 Watch exception

6.21.1 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.

6.21.2 Watch exception processing

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

6.21.3 Watch exception 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 re-enabled. The faulting instruction can be executed either by interpretation or by setting breakpoints.

6.22 Interrupt exception

6.22.1 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.

6.22.2 Interrupt exception 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 disserted 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 interrupts.

6.22.3 Interrupt exception 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 STLS2F01 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 STLS2F01's FPU can execute SIMD fixed-point multimedia instructions.

7.1 Overview

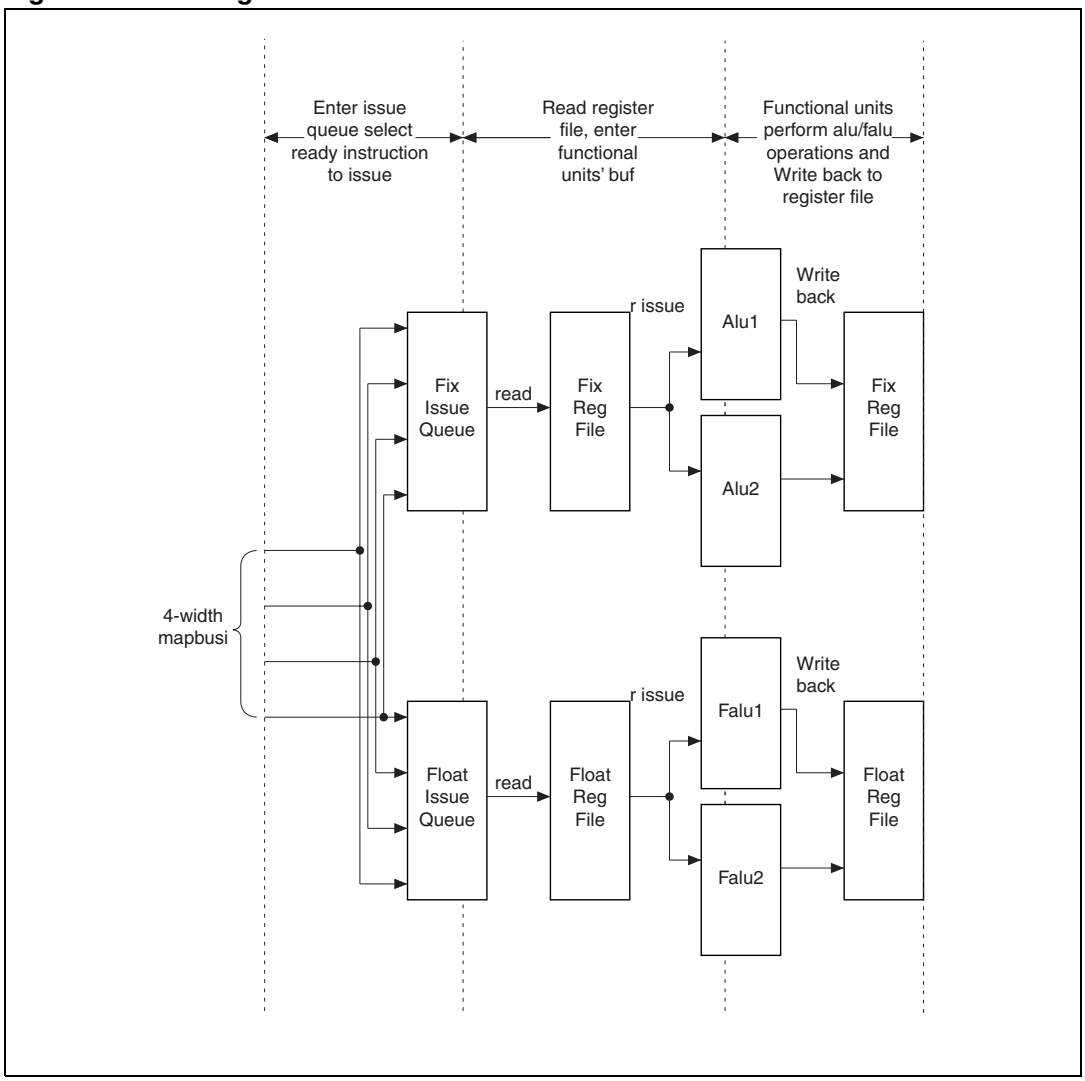
The FPU operates as a coprocessor for the CPU (it is assigned coprocessor label CP1), and extends the CPU instruction set to perform arithmetic operations on floating-point values.

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 STLS2F01 FPU can perform PS (paired-single) and fixed-point multimedia instructions. *Figure 16* illustrates the organization of the functional units in STLS2F01's architecture.

Figure 16. The organization of the functional units in STLS2F01'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 STLS2F01'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 40 describes the Implementation and Revision register (FCR0) fields.

Table 40. 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.

FP control/status register bit assignments shows the format of the Control/Status register, and *Table 41* describes the Control/Status register fields.

FP control/status register bit assignments

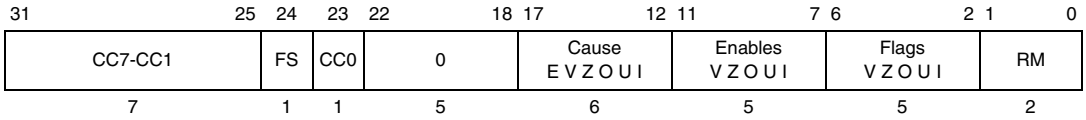


Table 41. Control/status register fields

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 *FP control/status register bit assignments* paragraph, 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 un-enabled 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 42*, these bits specify the rounding mode that the FPU uses for all floating-point operations.

Table 42. 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 $+\infty$: round to value closest to and not less than the infinitely precise result.
3	RM	Round toward $-\infty$: 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 STLS2F01 FPU not only performs floating-point instructions defined by MIPS standard, but also adds some special instructions like multimedia and PS operations to enhance STLS2F01 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 STLS2F01 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 43* 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 44* lists the details.
- Multimedia instructions (fmt =12~31). The media extensions for the STLS2F01 Architecture were designed to enhance performance of advanced media and communication applications. The STLS2F01 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 43. Floating point instructions in STLS2F01 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 44. Paired-single (PS) instructions in STLS2F01 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

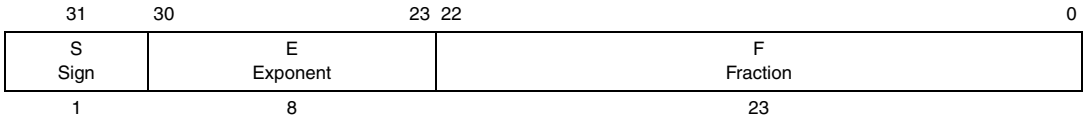
7.4 FPU formats

7.4.1 Floating-point format

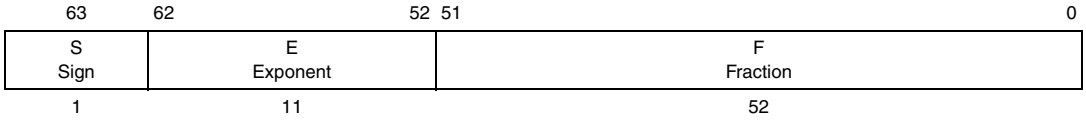
The FPU performs both 32-bit (single-precision) and 64-bit (double-precision) IEEE standard floating-point operations. The 32-bit single-precision format has a 24-bit signed-magnitude fraction field (f+s) and an 8-bit exponent (e); The 64-bit double-precision format has a 53-bit signed-magnitude fraction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

single format constraints two single-precision floating-point format. as shown in *Single precision format*, *Double precision format* and *Paired single format* paragraphs.

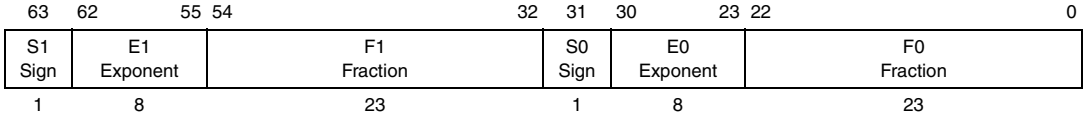
Single precision format



Double precision format



Paired single 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 Emin and Emax inclusive, together with two other reserved values:

- E_{min} - 1 (to encode±0 and denormalized numbers)
- E_{max} +1 (to encode±∞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 45*.

Table 45. Equations to calculate single & double precision FP format values

NO.	Equation
(1)	if $E = E_{max}+1$ and $f \neq 0$, 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 $f \neq 0$, 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 46 defines the values for the format parameters; minimum and maximum floating-point values are given in *Table 47*.

Table 46. 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 47. 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

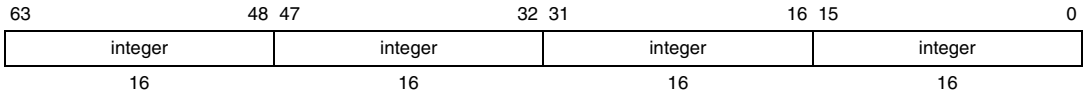
The Multimedia technology introduces new packed data types, each 64 bits long. The data elements can be:

- Eight packed, consecutive 8-bit bytes
- Four packed, consecutive 16-bit half-words
- Ttwo packed, consecutive 32-bit words
- One 64-bit double-word

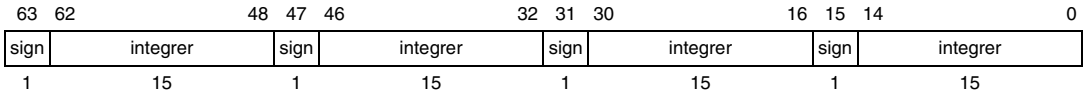
The 64 bits are numbered 0 through 63. Bit 0 is the least significant bit (LSB), and bit 63 is the most significant bit (MSB). The low-order bits are the lower part of the data element and the high-order bits are the upper part of the data element. For example, a word contains 16 bits numbered 0 through 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. See *Packed unsigned half-word format* and *Packed signed half-word format* paragraphs.

Packed unsigned half-word format



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 48* lists the default action taken by the FPU for each of the IEEE exceptions.

Table 48. 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.

7.6.1 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 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.

7.6.2 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: (+∞) + (–∞) or (–∞) – (–∞)
- 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.

7.6.3 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.

7.6.4 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.

7.6.5 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$\pm 2^{Emin}$).

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 .

7.6.6 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 49 lists those privileged instructions of STLS2F01.

Table 49. STLS2F01 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 STLS2F01 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 50*.

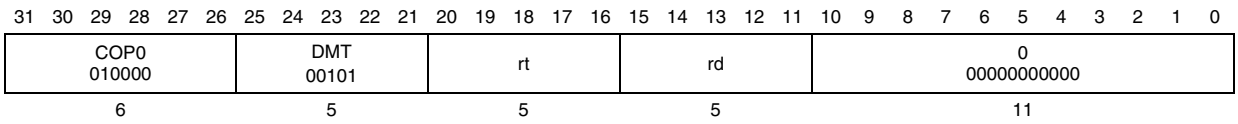
Table 50. CP0 move instructions

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

Doubleword move from system control coprocessor



Format: DMFC0 rt, rd

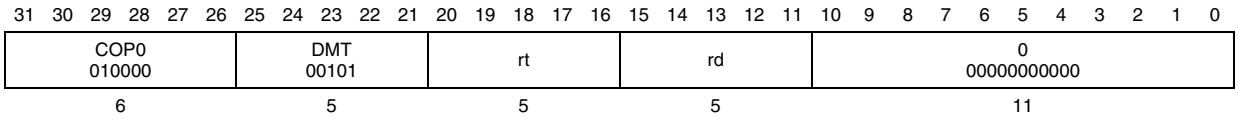
Description: The contents of coprocessor register rd of the CP0 are loaded into general register rt. This operation is defined for the STLS2F01 operating in kernel mode. Execution of these instruction in user or supervisor mode causes a coprocessor unusable exception. All 64-bits of the general register destination are written from the coprocessor register source.

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

Exception: Coprocessor unusable exception

8.1.2 DMTC0 instruction

Doubleword move to system control coprocessor



Format: DMTC0 rt, rd

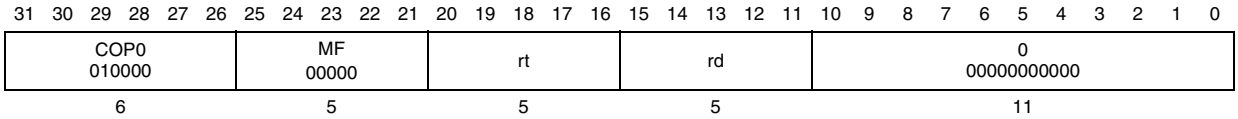
Description: The contents of general register rt are loaded into coprocessor register rd of the CP0. This operation is defined for the STLS2F01 operating in kernel mode. Execution of this instruction in user or supervisor mode causes a coprocessor unusable exception. All 64-bits of the coprocessor register destination are written from the general register source.

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

Exception: Coprocessor unusable exception

8.1.3 MFC0 instruction

Move from system control coprocessor



Format: MFC0 rt, rd

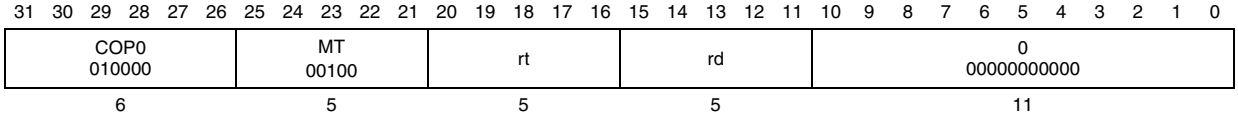
Description: The contents of coprocessor register rd of the CP0 are loaded into general register rt.

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

Exception: Coprocessor unusable exception

8.1.4 MTC0 instruction

Move to system control coprocessor



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]

Exception: Coprocessor unusable exception

8.1.5 Usable CP0 move instruction in user mode

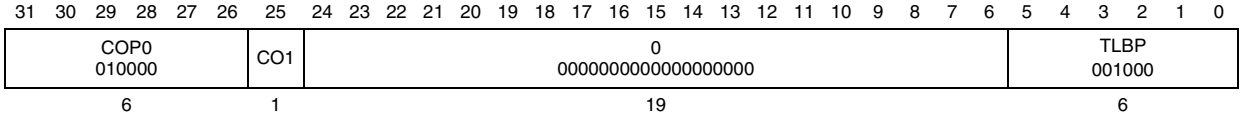
When users use DMFC0 or MFC0 to read the coprocessor 0 register No.24 or No.25 in order to get the performance information of the STLS2F01 processor, this execution doesn't cause a coprocessor unusable exception.

8.2 TLB access instructions

The STLS2F01 processor implements TLB instructions, TLBP, TLBI, TLBWI and TLBWR.

8.2.1 TLBP instruction

Probe TLB for matching entry



Format: TLBP

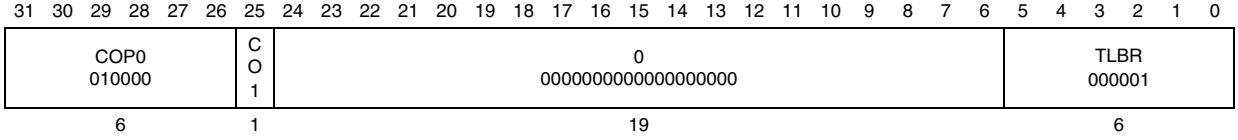
Description: The *Index* register is loaded with the address of the TLB entry whose contents match the contents of the *EntryHi* register. If no TLB entry matches, the high-order bit of the Index register is set to 0x80000000.

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

Exception: Coprocessor unusable exception

8.2.2 TLBR instruction

Read indexed TLB entry



Format: TLBR

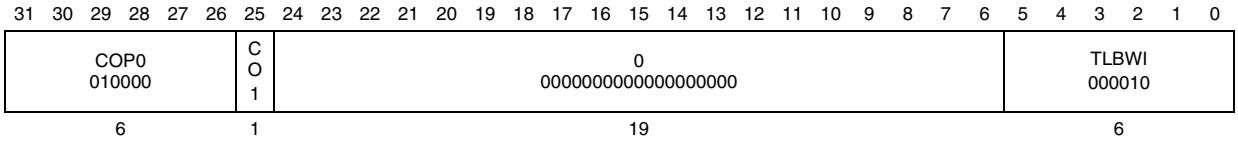
Description: The G bit (which controls ASID matching) read from the TLB is written into both of the *EntryLo0* and *EntryLo1* registers. The *EntryHi* and *EntryLo* registers are loaded with the contents of the TLB entry pointed at by the contents of the TLB Index register. TLBR can be executed in mapped 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[$Index_{5..0}$]$_{140}$

Exception: Coprocessor unusable exception

8.2.3 TLBWI instruction

Write indexed TLB entry



Format: TLBWI

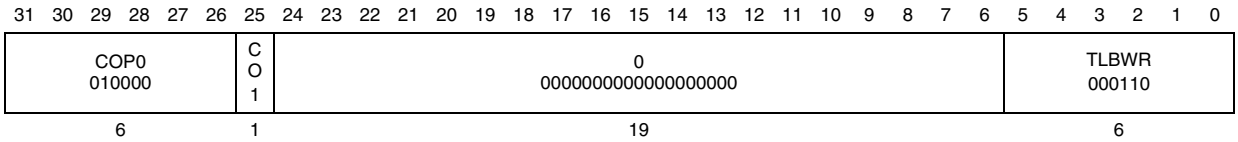
Description: The *G* bit of the TLB is written with the logical AND of the *G* bits in the *EntryLo0* and *EntryLo1* registers. The TLB entry pointed at by the contents of the TLB *Index* register is loaded with the contents of the *EntryHi* and *EntryLo* registers. The operation is invalid (and the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

Operation: TLB[$Index_{5..0}$]<-PageMask||(EntryHi and not PageMask)||EntryLo1||EntryLo0

Exception: Coprocessor unusable exception

8.2.4 TLBWR instruction

Write random TLB entry



Format: TLBWR

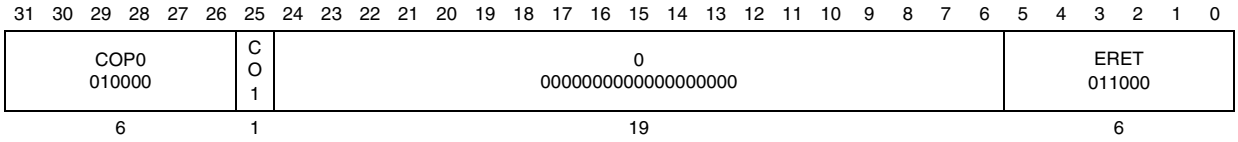
Description: The G bit of the TLB is written with the logical AND of the *G* bits in the *EntryLo0* and *EntryLo1* registers. The TLB entry pointed at by the contents of the TLB *Random* register is loaded with the contents of the *EntryHi* and *EntryLo* registers.

Operation: TLB[$Random_{5..0}$]<-PageMask||(EntryHi and not PageMask)||EntryLo1||EntryLo0

Exception: Coprocessor unusable exception

8.3 ERET instruction

Exception return



Format: ERET

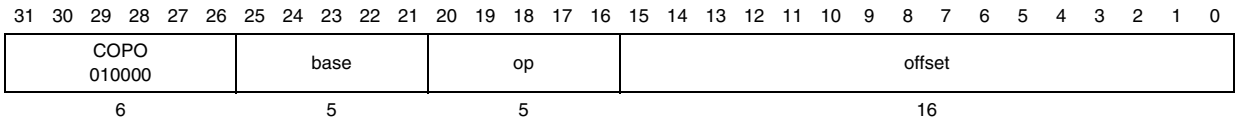
Description: ERET is the instruction for returning from an interrupt, exception, or error trap. Unlike a branch or jump instruction, ERET does not execute the next instruction. ERET must not itself be placed in a branch delay slot. If the processor is servicing an error trap (*SR2* = 1), then load the 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

Exception: Coprocessor unusable exception

8.4 CACHE instruction

Cache intruction



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 51*) 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 51. 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)

Exception: 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 Address window configuration

The main purpose of the configure registers module is to configure the address window. Each window contains three 64-bit registers, BASE, MASK and MMAP. The BASE register is aligned to M bytes. The MASK register has the similar format with the network mask which high bits are ones. The low 2-bit of MMAP register contains the corresponding ASL number. The write to these configuration register can be done by the doubleword store instruction.

Window match formula:

```
(IN_ADDR & MASK) == BASE
```

Address translation formula:

```
OUT_ADDR = (IN_ADDR & ~MASK) | {MMAP[63:20],20'h0}
```

Table 52. Address of the window configuration register

Address	Name	Description
3ff0 0000	M0_WIN0_BASE	The BASE address of Master 0's window 0
3ff0 0008	M0_WIN1_BASE	The BASE address of Master 0's window 1
3ff0 0010	M0_WIN2_BASE	The BASE address of Master 0's window 2
3ff0 0018	M0_WIN3_BASE	The BASE address of Master 0's window 3
3ff0 0020	M0_WIN0_SIZE	The MASK of Master 0's window 0
3ff0 0028	M0_WIN1_SIZE	The MASK of Master 0's window 1
3ff0 0030	M0_WIN2_SIZE	The MASK of Master 0's window 2
3ff0 0038	M0_WIN3_SIZE	The MASK of Master 0's window 3
3ff0 0040	M0_WIN0_MMAP	The MMAP of Master 0's window 0
3ff0 0048	M0_WIN1_MMAP	The MMAP of Master 0's window 1
3ff0 0050	M0_WIN2_MMAP	The MMAP of Master 0's window 2
3ff0 0058	M0_WIN3_MMAP	The MMAP of Master 0's window 3
3ff0 0060	M1_WIN0_BASE	The BASE address of Master 1's window 0
3ff0 0068	M1_WIN1_BASE	The BASE address of Master 1's window 1
3ff0 0070	M1_WIN2_BASE	The BASE address of Master 1's window 2
3ff0 0078	M1_WIN3_BASE	The BASE address of Master 1's window 3
3ff0 0080	M1_WIN0_SIZE	The MASK of Master 1's window 0
3ff0 0088	M1_WIN1_SIZE	The MASK of Master 1's window 1
3ff0 0090	M1_WIN2_SIZE	The MASK of Master 1's window 2
3ff0 0098	M1_WIN3_SIZE	The MASK of Master 1's window 3
3ff0 00a0	M1_WIN0_MMAP	The MMAP of Master 1's window 0
3ff0 00a8	M1_WIN1_MMAP	The MMAP of Master 1's window 1
3ff0 00b0	M1_WIN2_MMAP	The MMAP of Master 1's window 2

Table 52. Address of the window configuration register (continued)

Address	Name	Description
3ff0 00b8	M1_WIN3_MMAP	The MMAP of Master 1's window 3
3ff0 00c0	M2_WIN0_BASE	The BASE address of Master 2's window 0
3ff0 00c8	M2_WIN1_BASE	The BASE address of Master 2's window 1
3ff0 00d0	M2_WIN2_BASE	The BASE address of Master 2's window 2
3ff0 00d8	M2_WIN3_BASE	The BASE address of Master 2's window 3
3ff0 00e0	M2_WIN0_MASK	The MASK of Master 2's window 0
3ff0 00e8	M2_WIN1_MASK	The MASK of Master 2's window 1
3ff0 00f0	M2_WIN2_MASK	The MASK of Master 2's window 2
3ff0 00f8	M2_WIN3_MASK	The MASK of Master 2's window 3
3ff0 0100	M2_WIN0_MMAP	The MMAP of Master 2's window 0
3ff0 0108	M2_WIN1_MMAP	The MMAP of Master 2's window 1
3ff0 0110	M2_WIN2_MMAP	The MMAP of Master 2's window 2
3ff0 0118	M2_WIN3_MMAP	The MMAP of Master 2's window 3
3ff0 0120	M3_WIN0_BASE	The BASE address of Master 3's window 0
3ff0 0128	M3_WIN1_BASE	The BASE address of Master 3's window 1
3ff0 0130	M3_WIN2_BASE	The BASE address of Master 3's window 2
3ff0 0138	M3_WIN3_BASE	The BASE address of Master 3's window 3
3ff0 0140	M3_WIN0_MASK	The MASK of Master 3's window 0
3ff0 0148	M3_WIN1_MASK	The MASK of Master 3's window 1
3ff0 0150	M3_WIN2_MASK	The MASK of Master 3's window 2
3ff0 0158	M3_WIN3_MASK	The MASK of Master 3's window 3
3ff0 0160	M3_WIN0_MMAP	The MMAP of Master 3's window 0
3ff0 0168	M3_WIN1_MMAP	The MMAP of Master 3's window 1
3ff0 0170	M3_WIN2_MMAP	The MMAP of Master 3's window 2
3ff0 0178	M3_WIN3_MMAP	The MMAP of Master 3's window 3

10 DDR2 SDRAM control interface

Design of the integrated DDR2 SDRAM controller of STLS2F01 obeys the JEDEC standard (JESD79-2B). In the STLS2F01, all the memory reads or writes obey the STANDARD JESD79-2B.

10.1 Function of DDR2 SDRAM controller

STLS2F01 CPU support up to 4 physical memory banks (implemented by the 4 chip select signal), with an address bus of 18 bits (ddr2_a[14:0] and ddr2_bank[2:0]). The max addressing space is 128 GB (237).

STLS2F01 CPU supports all types according to the JESD79-2B standard. Users can adjust the configuration of the DDR2 controller to support different memory chip type. The max number of CS_n is 4, RAS_n is 15, CAS_n is 14, BANK_n is 3.

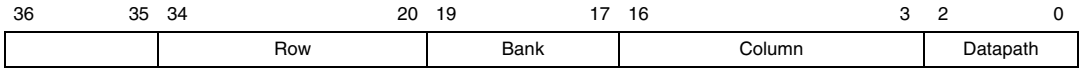
The physical address sent by CPU will be transform to the CAS/RAS in the format below:

For example: in the 4GB memory space as follows:

number of CS_n = 4, number of bank = 8

number of RAS_n = 12 number of CAS_n = 12

Transform form CPU physical address to DDR2 SDRAM address



The integrated DDR2 controller of STLS2F01 CPU only accept memory read or write request from CPU core or the peripheral device. In all memory read or write operation, the DDR2 controller is in the Slave State.

The integrated DDR2 controller of STLS2F01 CPU supports the dynamic page management. The DDR2 controller decides the Open Page or Close Page all by the hardware for the memory access, but not by the software designer. The characters of the integrated DDR2 controller as follows:

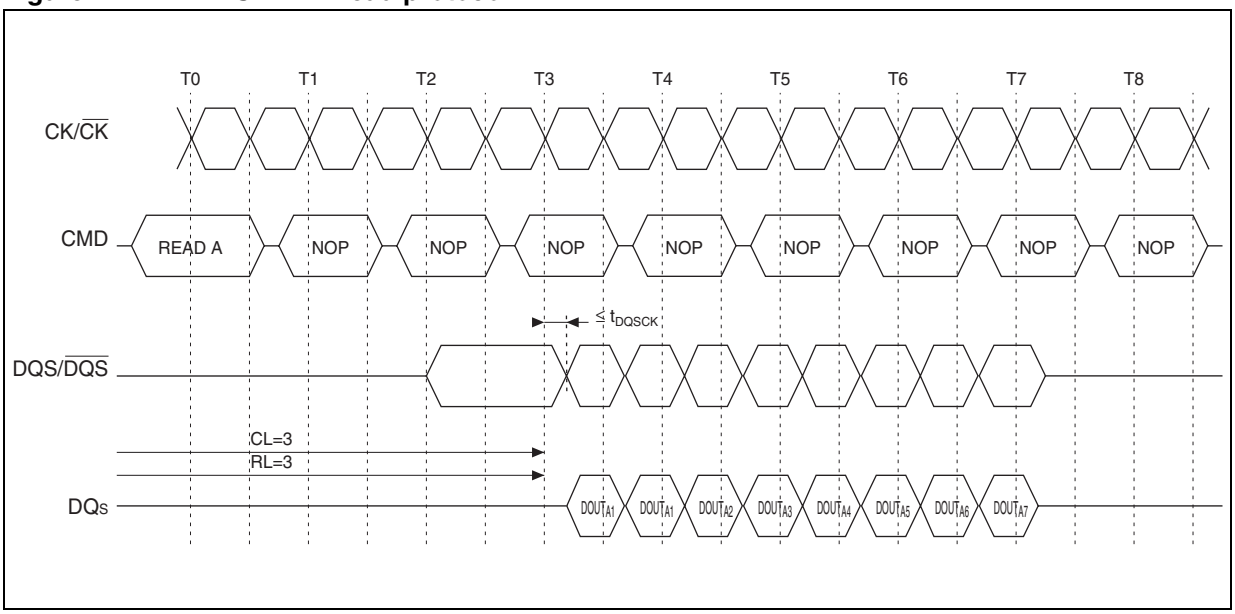
- Full pipelined address/data interface
- Unite of different memory access to improve the bandwidth
- Using register configuration to modify the memory access mode
- Integrated DCC for data send and receive
- ECC for auto correcting 1 bit error and detecting 2 bit error in the data path
- support for 133MHZ – 333MHZ DDR2

10.2 Protocol of DDR2 SDRAM read

Protocol of the DDR2 SDRAM read is as *Figure 17*. CMD in the figure is composed of RAS_n, CAS_n and WE_n. For memory read, RAS_n = 1, CAS_n = 0, WE_n = 1.

In the figure, Cas Latency = 3, Read Latency = 3, Burst Length = 8°

Figure 17. DDR2 SDRAM read protocol

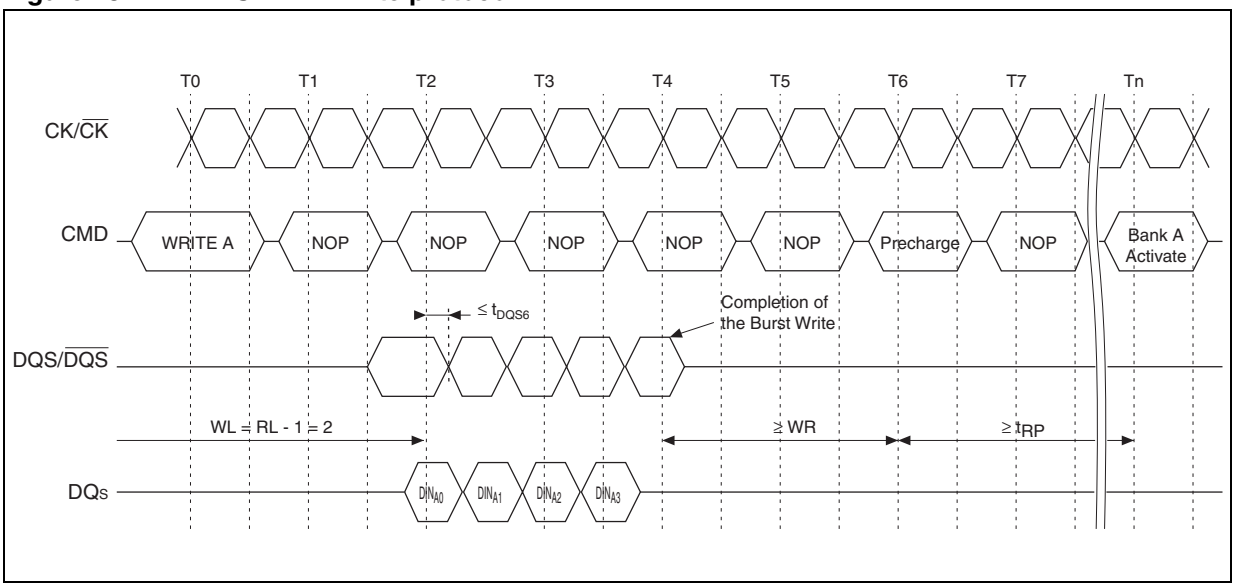


10.3 Protocol of DDR2 SDRAM write

Protocol of the DDR2 SDRAM write is as *Figure 18*. CMD in the figure is composed of RAS_n, CAS_n and WE_n. For memory read, RAS_n = 1, CAS_n = 0, WE_n = 1. Be different from memory read, the write operation needs DQM signals to indicate the write mask. the DQM must be synchronous with the DQ signals°

In the figure, write latency = read latency -1 = 2, burst length = 4°

Figure 18. DDR2 SDRAM write protocol



10.4 Registers of DDR2 SDRAM controller

User must reconfigure the DDR2 SDRAM controller after the system reset to enable different types of DDR2 SDRAM. The initialization of DDR2 SDRAM is presented in the JESD79-2B, before the configuration, the DDR2 SDRAM is not usable. The sequence to initialize the DDR2 SDRAM is as follows:

1. System reset, asresetn = 0, all registers of the DDR2 controller is reset to zero.
2. System reset done., aresetn = 1.
3. The doubleword write command should be issued to the DDR2 controller registers to configure 29 registers. The parameter START of CTRL_03 should be written to 0 in this phase.
4. The doubleword write command should be issued to the DDR2 controller register CTRL_03. The parameter START of CTRL_03 should be written to 1 this time. Then the DDR2 controller will start the initialization of the DDR2 SDRAM automatically.

For the STLS2F01, the configuration of DDR2 SDRAM controller should between the initialization of the system board and the using of the memory. The base address of the configuration registers is 0x000000000FFFFE00. One register may contain one or more parameters of the controller. The parameters of the registers are listed as follows (the bits not listed are all reserved). A configuration method of DDR2 667 is also given by the *Table 52*. Users can reconfigure the registers to optimize the memory access.

Table 53. Formation of DDR SDRAM controller registers

Name	Bit	Default	Range	Description
CONF_CTL_00[31:0] Offset: 0x00			**DDR2 667:0x00000101**	
AREFRESH	24:24	0x0	0x0-0x1	issue a auto refresh command according to the parameter auto_refresh_mode (WRITE ONLY)
AP	16:16	0x0	0x0-0x1	enable the auto-precharge mode of the contoller
ADDR_CMP_EN	8:8	0x0	0x0-0x1	enable address collision detection of command queue placement logic
ACTIVE_AGING	0:0	0x0	0x0-0x1	enable command aging in the command queue
CONF_CTL_00[63:32] Offset: 0x00			**DDR2 667:0x01000100**	
DDR2_SDRAM_MODE	56:56	0x0	0x0-0x1	ddr or ddr2 mode
CONCURRENTAP	48:48	0x0	0x0-0x1	allows to issue command to another bank while a bank is in auto precharge (many of dimms not support this mode)
BANK_SPLIT_EN	40:40	0x0	0x0-0x1	enable bank splitting for command queue placement
AUTO_REFRESH_MODE	32:32	0x0	0x0-0x1	set auto precharge will be next burst or next command boundary
CONF_CTL_01[31:0] Offset: 0x10			**DDR2 667:0x00010000**	
ECC_DISBALE_W_UC_ERR	24:24	0x0	0x0-0x1	disable the corruption or ECC when an uncorrectable error occur

Table 53. Formation of DDR SDRAM controller registers (continued)

Name	Bit	Default	Range	Description
DQS_N_EN	16:16	0x0	0x0-0x1	enable the dqs_n
DLL_BYPASS_MODE	8:8	0x0	0x0-0x1	enable DLL BYPASS mode
DLLLOCKREG	0:0	0x0	0x0-0x1	indicate if DLL locked (READ ONLY)
CONF_CTL_01[63:32] Offset: 0x10			**DDR2 667:0x00100000**	
FWC	56:56	0x0	0x0-0x1	force a write check, Xor XOR_CHECK_BITS with ecc codes and write to memory (WRITE ONLY)
FAST_WRITE	48:48	0x0	0x0-0x1	enable the fast write mode, to issue write to memory as soon as the write command is received
ENABLE_QUICK_SREFRESH	40:40	0x0	0x0-0x1	allow the user to interrupt memory initialization to enter the self refresh mode
EIGHT_BANK_MODE	32:32	0x0	0x0-0x1	if the number of banks is 8
CONF_CTL_02[31:0] Offset: 0x20			**DDR2 667:0x00000000**	
NO_CMD_INIT	24:24	0x0	0x0-0x1	disable DRAM command until the TDLL is expired during the initialization
INTRPTWRITENA	16:16	0x0	0x0-0x1	allow the controller to interrupt the combined write with auto precharge with another write
INTRPTREADA	8:8	0x0	0x0-0x1	allow the controller to interrupt the combined read with auto precharge with another read
INTRPTAPBURST	0:0	0x0	0x0-0x1	allow the controller to interrupt the auto precharge with command of another bank
CONF_CTL_02[63:32] Offset: 0x20			**DDR2 667:0x01000101**	
PRIORITY_EN	56:56	0x0	0x0-0x1	enable priority of the command placement logic
POWER_DOWN	48:48	0x0	0x0-0x1	disable the CKE
PLACEMENT_EN	40:40	0x0	0x0-0x1	enable the command placement logic
ODT_ADD_TURN_CLK_EN	32:32	0x0	0x0-0x1	enable extra turn around clock between back to back command to different cs_n
CONF_CTL_03[31:0] Offset: 0x30			**DDR2 667:0x01000000**	
RW_SAME_EN	24:24	0x0	0x0-0x1	enable the command grouping for placement logic
REG_DIMM_EN	16:16	0x0	0x0-0x1	enable the registered DIMM operation of controller
REDUC	8:8	0x0	0x0-0x1	enable the half data path feature

Table 53. Formation of DDR SDRAM controller registers (continued)

Name	Bit	Default	Range	Description
PWRUP_SREFRESH_EXIT	0:0	0x0	0x0-0x1	Power up by self refresh instead of initialization
CONF_CTL_03[63:32] Offset: 0x30			**DDR2 667:0x01010000**	
SWAP_PORT_RW_SAME_EN	56:56	0x0	0x0-0x1	enable the swapping between command of same type from same port
SWAP_EN	48:48	0x0	0x0-0x1	enable command swapping
START	40:40	0x0	0x0-0x1	start the DRAM initialization
SREFRESH	32:32	0x0	0x0-0x1	set the self refresh mode
CONF_CTL_04[31:0] Offset: 0x40			**DDR2 667:0x00010101**	
WRITE_MODEREG	24:24	0x0	0x0-0x1	write EMRS data (WRITE ONLY)
WRITEINTERP	16:16	0x0	0x0-0x1	allow to interrupt write with a read command
TREF_ENABLE	8:8	0x0	0x0-0x1	issue self refresh the DRAM every TREF cycles
TRAS_LOCKOUT	0:0	0x0	0x0-0x1	allow to issue auto precharge before TRAS_MIN
CONF_CTL_04[63:32] Offset: 0x40			**DDR2 667:0x01000202**	
RTT_0	57:56	0x0	0x0-0x3	ODT resistance setting
CTRL_RAW	49:48	0x0	0x0-0x3	ECC mode: 2'b00 – ECC not used 2'b01 – ECC error reported, but not corrected 2'b10 – no ECC ram available 2'b11 – ECC error reported and corrected
AXI0_W_PRIORITY	41:40	0x0	0x0-0x3	priority of write command
AXI0_R_PRIORITY	33:32	0x0	0x0-0x3	priority of read command
CONF_CTL_05[31:0] Offset: 0x50			**DDR2 667:0x04050202**	
COLUMN_SIZE	26:24	0x0	0x0-0x7	the different between the actual number of the column size and 14
CASLAT	18:16	0x0	0x0-0x7	set the CAS latency
ADDR_PINS	10:8	0x0	0x0-0x7	the different between the actual number of the address pins and 14
RTT_PAD_TERMINATION	1:0	0x0	0x0-0x3	set termination resistance of controller pad
CONF_CTL_05[63:32] Offset: 0x50			**DDR2 667:0x00000000**	
Q_FULLNESS	58:56	0x0	0x0-0x7	quantity that command queue is full

Table 53. Formation of DDR SDRAM controller registers (continued)

Name	Bit	Default	Range	Description
PORT_DATA_ERROR_TYPE	50:48	0x0	0x0-0x7	error type that caused the data error(READ ONLY) bit 0 – data overflow bit 1 – write data interleaved bit 2 – uncorrected data error
OUT_OF_RANGE_TYPE	42:40	0x0	0x0-0x7	error type that caused the out of range error(READ ONLY)
MAX_CS_REG	34:32	0x4	0x0-0x4	the max number of cs_n(READ ONLY)
CONF_CTL_06[31:0] Offset: 0x60			**DDR2 667:0x03040203**	
TRTP	26:24	0x0	0x0-0x7	cycles from read to precharge
TRRD	18:16	0x0	0x0-0x7	cycles between different bank active
TEMRS	10:8	0x0	0x0-0x7	write emrs cycles
TCKE	2:0	0x0	0x0-0x7	the minimum cycles of CKE
CONF_CTL_06[63:32] Offset: 0x60			**DDR2 667:0x0a040305**	
APREBIT	59:56	0x0	0x0-0xf	which bit of address pin to indicate auto recharge
WRLAT	50:48	0x0	0x0-0x7	cycles between write command and the first data
TWTR	42:40	0x0	0x0-0x7	cycles from write to read
TWR_INT	34:32	0x0	0x0-0x7	cycles form write to another active
CONF_CTL_07[31:0] Offset: 0x70			**DDR2 667:0x000f090a**	
ECC_C_ID	27:24	0x0	0x0-0xf	ID of the correctable ECC error(READ ONLY)
CS_MAP	19:16	0x0	0x0-0xf	CS_n available
CASLAT_LIN_GATE	11:8	0x0	0x0-0xf	half cycles from read command to gate open
CASLAT_LIN	3:0	0x0	0x0-0xf	half cycles from read command to first data
CONF_CTL_07[63:32] Offset: 0x70			**DDR2 667:0x00000400**	
MAX_ROW_REG	59:56	0xf	0x0-0xf	maximum number of rows(READ ONLY)
MAX_COL_REG	51:48	0xe	0x0-0xe	maximum number of columns(READ ONLY)
INITAREF	43:40	0x0	0x0-0xf	number of auto refresh when initialization
ECC_U_ID	35:32	0x0	0x0-0xf	ID of the uncorrectable ECC error(READ ONLY)
CONF_CTL_08[31:0] Offset: 0x80			**DDR2 667:0x01020408**	

Table 53. Formation of DDR SDRAM controller registers (continued)

Name	Bit	Default	Range	Description
ODT_RD_MAP_CS3	27:24	0x0	0x0-0xf	enable the ODT of CS_n[3] when CS_n[3] read
ODT_RD_MAP_CS2	19:16	0x0	0x0-0xf	enable the ODT of CS_n[2] when CS_n[2] read
ODT_RD_MAP_CS1	11:8	0x0	0x0-0xf	enable the ODT of CS_n[1] when CS_n[1] read
ODT_RD_MAP_CS0	3:0	0x0	0x0-0xf	enable the ODT of CS_n[0] when CS_n[0] read
CONF_CTL_08[63:32] Offset: 0x80			**DDR2 667:0x01020408**	
ODT_WR_MAP_CS3	59:56	0x0	0x0-0xf	enable the ODT of CS_n[3] when CS_n[3] write
ODT_WR_MAP_CS2	51:48	0x0	0x0-0xf	enable the ODT of CS_n[2] when CS_n[2] write
ODT_WR_MAP_CS1	43:40	0x0	0x0-0xf	enable the ODT of CS_n[1] when CS_n[1] write
ODT_WR_MAP_CS0	35:32	0x0	0x0-0xf	enable the ODT of CS_n[0] when CS_n[0] write
CONF_CTL_09[31:0] Offset: 0x90			**DDR2 667:0x00000000**	
PORT_DATA_ERROR_ID	27:24	0x0	0x0-0xf	ID of the data error(READ ONLY)
PORT_CMD_ERROR_TYPE	19:16	0x0	0x0-0xf	error type of the error command(READ ONLY) bit 0 – size too big bit 1 – starting or ending of wrap address not aligned bit 2 – byte count of wrap command is not log2 value bit 3 – narrow transform error
PORT_CMD_ERROR_ID	11:8	0x0	0x0-0xf	ID of the command error(READ ONLY)
OUT_OF_RANGE_SOURCE_ID	3:0	0x0	0x0-0xf	ID of the out of range error(READ ONLY)
CONF_CTL_09[63:32] Offset: 0x90			**DDR2 667:0x0000050b**	
OCD_ADJUST_PUP_CS0	60:56	0x0	0x0-0x1f	value of OCD pull up when CS_n[0]
OCD_ADJUST_PDN_CS0	52:48	0x0	0x0-0x1f	value of OCD pull down when CS_n[0]
TRP	43:40	0x0	0x0-0xf	cycles of pre-charge operation
TDAL	35:32	0x0	0x0-0xf	cycles of write recovery with auto-precharge
CONF_CTL_10[31:0] Offset: 0xa0			**DDR2 667:0x3f130200**	
AGE_COUNT	29:24	0x0	0x0-0x3f	initial value of command aging of placement logic

Table 53. Formation of DDR SDRAM controller registers (continued)

Name	Bit	Default	Range	Description
TRC	20:16	0x0	0x0-0x1f	cycles between active to the same bank
TMRD	12:8	0x0	0x0-0x1f	cycles to configure the MRD register
TFAW	4:0	0x0	0x0-0x1f	tFAW from 8 banks
CONF_CTL_10[63:32] Offset: 0xa0			**DDR2 667:0x1d1d1d3f**	
DLL_DQS_DELAY_2	62:56	0x0	0x0-0x7f	the percentage of the DQS2 delay, n for n/128 cycle
DLL_DQS_DELAY_1	54:48	0x0	0x0-0x7f	the percentage of the DQS1 delay, n for n/128 cycle
DLL_DQS_DELAY_0	46:40	0x0	0x0-0x7f	the percentage of the DQS0 delay, n for n/128 cycle
COMMAND_AGE_COUNT	37:32	0x0	0x0-0x3f	initial value of individual command aging count
CONF_CTL_11[31:0] Offset: 0xb0			**DDR2 667:0x1d1d1d1d**	
DLL_DQS_DELAY_6	30:24	0x0	0x0-0x7f	the percentage of the DQS6 delay, n for n/128 cycle
DLL_DQS_DELAY_5	22:16	0x0	0x0-0x7f	the percentage of the DQS5 delay, n for n/128 cycle
DLL_DQS_DELAY_4	14:8	0x0	0x0-0x7f	the percentage of the DQS4 delay, n for n/128 cycle
DLL_DQS_DELAY_3	6:0	0x0	0x0-0x7f	the percentage of the DQS3 delay, n for n/128 cycle
CONF_CTL_11[63:32] Offset: 0xb0			**DDR2 667:0x507f1d1d**	
WR_DQS_SHIFT	62:56	0x0	0x0-0x7f	the percentage of the clk_wr delay, n for n/128 cycle
DQS_OUT_SHIFT	54:48	0x0	0x0-0x7f	the percentage of dqs_out delay, n for n/128 cycle
DLL_DQS_DELAY_8	46:40	0x0	0x0-0x7f	the percentage of the DQS8 delay, n for n/128 cycle
DLL_DQS_DELAY_7	38:32	0x0	0x0-0x7f	the percentage of the DQS7 delay, n for n/128 cycle
CONF_CTL_12[31:0] Offset: 0xc0			**DDR2 667:0x0e000000**	
TRAS_MIN	31:24	0x0	0x0-0xff	cycles of valid time of active command
OUT_OF_RANGE_LENGTH	23:16	0x0	0x0-0xff	length of out of rang error command(READ ONLY)
ECC_U_SYND	15:8	0x0	0x0-0xff	syndrome of uncorrectable error(READ ONLY)
ECC_C_SYND	7:0	0x0	0x0-0xff	syndrome of correctable error(READ ONLY)
CONF_CTL_12[63:32] Offset: 0xc0			**DDR2 667:0x002a3305**	

Table 53. Formation of DDR SDRAM controller registers (continued)

Name	Bit	Default	Range	Description
DLL_DQS_DELAY_BYPASS_0	56:48	0x0	0x0-0x1ff	value of DLL_DQS_DELAY_0 of DLL bypass mode
TRFC	47:40	0x0	0x0-0xff	cycles of the refresh command
TRCD_INT	39:32	0x0	0x0-0xff	cycles from active to read/write command
CONF_CTL_13[31:0] Offset: 0xd0			**DDR2 667:0x002a002a**	
DLL_DQS_DELAY_BYPASS_2	24:16	0x0	0x0-0x1	value of DLL_DQS_DELAY_2 of DLL bypass mode
DLL_DQS_DELAY_BYPASS_1	8:0	0x0	0x0-0x1	value of DLL_DQS_DELAY_1 of DLL bypass mode
CONF_CTL_13[63:32] Offset: 0xd0			**DDR2 667:0x002a002a**	
DLL_DQS_DELAY_BYPASS_4	56:48	0x0	0x0-0x1ff	value of DLL_DQS_DELAY_4 of DLL bypass mode
DLL_DQS_DELAY_BYPASS_3	40:32	0x0	0x0-0x1ff	value of DLL_DQS_DELAY_3 of DLL bypass mode
CONF_CTL_14[31:0] Offset: 0xe0			**DDR2 667:0x002a002a**	
DLL_DQS_DELAY_BYPASS_6	24:16	0x0	0x0-0x1ff	value of DLL_DQS_DELAY_6 of DLL bypass mode
DLL_DQS_DELAY_BYPASS_5	8:0	0x0	0x0-0x1ff	value of DLL_DQS_DELAY_5 of DLL bypass mode
CONF_CTL_14[63:32] Offset: 0xe0			**DDR2 667:0x002a002a**	
DLL_DQS_DELAY_BYPASS_8	56:48	0x0	0x0-0x1ff	value of DLL_DQS_DELAY_8 of DLL bypass mode
DLL_DQS_DELAY_BYPASS_7	40:32	0x0	0x0-0x1ff	value of DLL_DQS_DELAY_7 of DLL bypass mode
CONF_CTL_15[31:0] Offset: 0xf0			**DDR2 667:0x00000004**	
DLL_LOCK	24:16	0x0	0x0-0x1ff	value of DLL lock when locked
DLL_INCREMENT	8:0	0x0	0x0-0x1ff	increment value in DLL detecting phase
CONF_CTL_15[63:32] Offset: 0xf0			**DDR2 667:0x00b4000a**	
DQS_OUT_SHIFT_BYPASS	56:48	0x0	0x0-0x1ff	value of DQS_OUT_SHIFT of DLL bypass mode
DLL_START_POINT	40:32	0x0	0x0-0x1ff	initial value in DLL detecting phase
CONF_CTL_16[31:0] Offset: 0x100			**DDR2 667:0x00000087**	
INT_ACK	25:16	0x0	0x0-0x3ff	interrupt acknowledge
WR_DQS_SHIFT_BYPASS	8:0	0x0	0x0-0x1ff	value of WR_DQS_SHIFT of DLL bypass mode

Table 53. Formation of DDR SDRAM controller registers (continued)

Name	Bit	Default	Range	Description
CONF_CTL_16[63:32] Offset: 0x100 DDR2 667:0x00000000				
INT_STATUS	58:48	0x0	0x0-0x7ff	interrupt cause(READ ONLY) bit 0 – one command out of physical address bit 1 – more command out of physical address bit 2 – one ECC correctable error bit 3 – more ECC correctable error bit 4 – one ECC uncorrectable error bit 5 – more ECC uncorrectable error bit 6 – error of controller address channel bit 7 – error of controller data channel bit 8 – initialization complete bit 9 – DLL not locked bit 10 – OR of lower bits
INT_MASK	42:32	0x0	0x0-0x7ff	interrupt mask
CONF_CTL_17[31:0] Offset: 0x110 DDR2 667:0x0000181b				
EMRS1_DATA	30:16	0x0	0x0-0x7ff	value written to EMRS1 when initialization
TREF	13:0	0x0	0x0-0x3ff	cycles between two refresh command
CONF_CTL_17[63:32] Offset: 0x110 DDR2 667:0x00000000				
EMRS2_DATA_1	62:48	0x0000	0x0-0x7fff	value written to EMRS2 of CS[1] when initialization
EMRS2_DATA_0	46:32	0x0000	0x0-0x7fff	value written to EMRS2 of CS[0] when initialization
CONF_CTL_18[31:0] Offset: 0x120 DDR2 667:0x00000000				
EMRS2_DATA_3	30:16	0x0000	0x0-0x7fff	value written to EMRS2 of CS[3] when initialization
EMRS2_DATA_2	14:0	0x0000	0x0-0x7fff	value written to EMRS2 of CS[2] when initialization
CONF_CTL_18[63:32] Offset: 0x120 DDR2 667:0x001c0000				
AXI0_EN_LT_WIDTH_INSTR	63:48	0x0000	0x0-0xffff	enable narrow command
EMRS3_DATA	46:32	0x0000	0x0-0x7fff	value written to EMRS3 when initialization
CONF_CTL_19[31:0] Offset: 0x130 DDR2 667:0x00c8006b				
TDLL	31:16	0x0000	0x0-0xffff	cycles for DLL locking phase

Table 53. Formation of DDR SDRAM controller registers (continued)

Name	Bit	Default	Range	Description
TCPD	15:0	0x0000	0x0-0xffff	cycles from CKE to precharge
CONF_CTL_19[63:32] Offset: 0x130 DDR2 667:0x48e10002				
TRAS_MAX	63:48	0x0000	0x0-0xffff	MAX cycles of active command
TPDEX	47:32	0x0000	0x0-0xffff	cycles of power down exit
CONF_CTL_20[31:0] Offset: 0x140 DDR2 667:0x00c8002f				
TXSR	31:16	0x0000	0x0-0xffff	cycles of self refresh exit
TXSNR	15:0	0x0000	0x0-0xffff	parameter tXSNR
CONF_CTL_20[63:32] Offset: 0x140 DDR2 667:0x00000000				
XOR_CHECK_BITS	63:48	0x0000	0x0-0xffff	value when force write check
VERSION	47:32	0x2041	0x2041	version of controller(READ ONLY)
CONF_CTL_21[31:0] Offset: 0x150 DDR2 667:0x00000036				
ECC_C_ADDR[7:0]	31:24	0x0000	0x0-0x1fffffff	address[7:0] of the correctable ECC error(READ ONLY)
TINIT	23:0	0x0000	0x0-0xfffff	cycles of initialization
CONF_CTL_21[63:32] Offset: 0x150 DDR2 667:0x00000000				
ECC_C_ADDR[36:8]	60:32	0x0	0x0-0x1fffffff	address[36:8] of the correctable ECC error(READ ONLY)
CONF_CTL_22[31:0] Offset: 0x160 DDR2 667:0x00000000				
ECC_U_ADDR[31:0]	31:0	0x0	0x0-0x1fffffff	address[31:0] of the uncorrectable ECC error(READ ONLY)
CONF_CTL_22[63:32] Offset: 0x160 DDR2 667:0x00000000				
ECC_U_ADDR[36:32]	36:32	0x0	0x0-0x1fffffff	address[36:32] of the uncorrectable ECC error
(READ ONLY)				
CONF_CTL_23[31:0] Offset: 0x170 DDR2 667:0x00000000				
OUT_OF_RANGE_ADDR[31:0]	31:0	0x0	0x0-0x1fffffff	address[31:0] of the out of range error(READ ONLY)
CONF_CTL_23[63:32] Offset: 0x170 DDR2 667:0x00000000				
OUT_OF_RANGE_ADDR[36:32]	36:32	0x0	0x0-0x1fffffff	address[36:32] of the out of range error(READ ONLY)
CONF_CTL_24[31:0] Offset: 0x180 DDR2 667:0x00000000				
PORT_CMD_ERROR_ADDR[31:0]	31:0	0x0	0x0-0x1fffffff	address[31:0] of the port cmd error(READ ONLY)

Table 53. Formation of DDR SDRAM controller registers (continued)

Name	Bit	Default	Range	Description
CONF_CTL_24[63:32] Offset: 0x180 DDR2 667:0x00000000				
PORT_CMD_ERROR_ADDR[36:32]	36:32	0x0	0x0-0x1fffffff	address[36:32] of the port cmd error(READ ONLY)
CONF_CTL_25[31:0] Offset: 0x190 DDR2 667:0x00000000				
ECC_C_DATA[31:0]	31:0	0x0	0x0-0x1fffffff	data[31:0] of the correctable ECC error(READ ONLY)
CONF_CTL_25[63:32] Offset: 0x190 DDR2 667:0x00000000				
ECC_C_DATA[63:32]	63:32	0x0	0x0-0x1fffffff	data[63:32] of the correctable ECC error(READ ONLY)
CONF_CTL_26[31:0] Offset: 0x1a0 DDR2 667:0x00000000				
ECC_U_DATA[31:0]	31:0	0x0	0x0-0x1fffffff	data[31:0] of the uncorrectable ECC error(READ ONLY)
CONF_CTL_26[63:32] Offset: 0x1a0 DDR2 667:0x00000000				
ECC_U_DATA[63:32]	63:32	0x0	0x0-0x1fffffff	data[63:32] of the uncorrectable ECC error(READ ONLY)
CONF_CTL_27[63:32] Offset: 0x1b0 DDR2 667:0x00000000				
CKE_DELAY	2:0	0x0	0x0-0x7	CKE delay
CONF_CTL_28[63:32] Offset: 0x1c0 DDR2 667:0x00000001				
UB_DIMM	0:0	0x0	0x0-0x1	enable unbuffered dimm mode

Some presentation of the registers:

1. (CONF_CTL_00 AP
   The parameter is used to enable the AUTOPRECHARGE, once AUTOPRECHARGE enabled, DDR2 SDRAM will close page after every memory access. This will cause performance descend when a lot of successive memory access.
2. CONF_CTL_00 CONCURRENTAP
   The parameter is used to issue concurrent autoprecharge, but not all memory chip support this mode.
3. CONF_CTL_03 SREFRESH
   The parameter is used to force the DDR2 SDRAM into self refresh mode. User must write 0 to this register when need to be back from self refresh.
4. CONF_CTL_07 CASLAT_LIN_GATE
   The parameter is used to control the sample of memory read, usually half cycle less than the parameter CASLAT_LIN. The parameter CASLAT_LIN is the number of half cycles of CAS latency.
5. CONF_CTL_15 DLL_INCREMENT
   The parameter should not be set to zero.
6. CONF_CTL_15 DLL_START_POINT
   The parameter should not be set to zero or 1. And it must be less than one and a half of DLL_LOCK_VALUE when locked.
7. CONF_CTL_28 UB_DIMM
   The parameter should be set to 1 when unbuffered DIMM used, and should be set to 0 when memory chips are used.

11 Integrated IO controller

11.1 Introduction of IO controller

The STLS2F01 has built in PCI/PCIX controller, Local IO controller, GPIO, interrupt controller and some video accelerate unit. These controllers share a slave port of cross bar, see *Figure 19*. Requests coming from the CPU core walk though the cross bar, and send to appropriate controller according to their address (refer to *Table 54* for the address space).

Figure 19. IO controller architecture

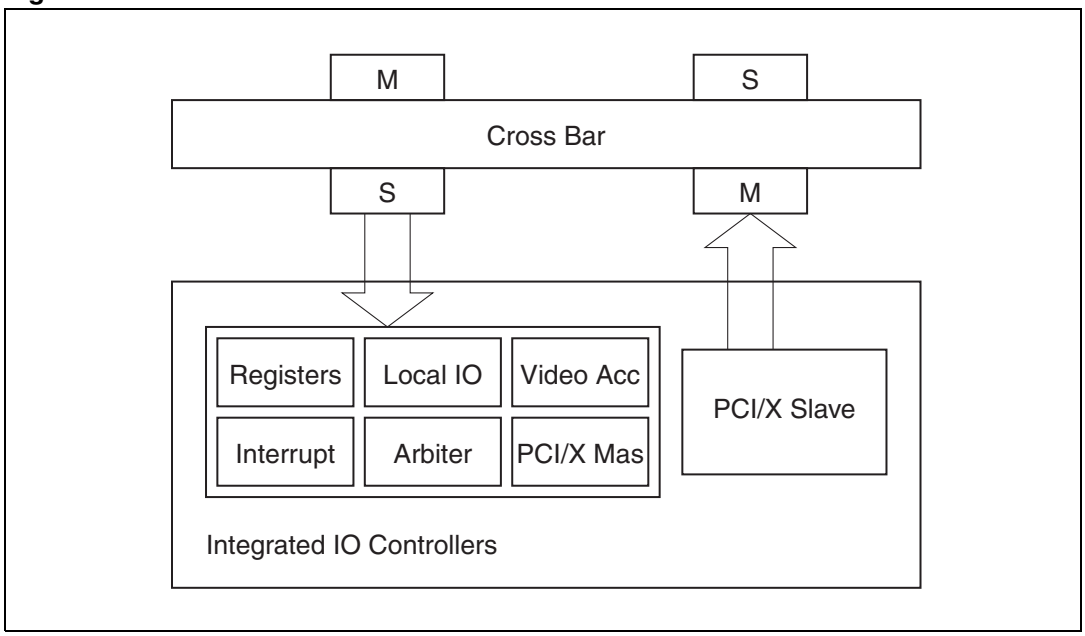


Table 54. IO controller address space

Start address	Size	The space	Access type	Note
0x00000000	256M	-	-	
0x10000000	64M	PCI MEM Lo0	CDWHB	(1)
0x13f00000	1M	Video Acc	CDWHB	(2)
0x14000000	64M	PCI MEM Lo1	CDWHB	(1)
0x18000000	64M	PCI MEM Lo2	CDWHB	(1)
0x1c000000	32M	LIO ROM	CDWHB	
0x1e000000	28M	LIO IO	CDWHB	
0x1fc00000	1M	BOOT ROM	CDWHB	
0x1fd00000	1M	PCI IO	WHB	
0x1fe00000	256B	Registers	WHB	
0x1fe00100	256B	PCI Header	WHB	

Table 54. IO controller address space (continued)

Start address	Size	The space	Access type	Note
0x1fe80000	2K	PCI CONF	WHB	
0x1ff00000	1M	LIO IO	CDWHB	(3)
0x20000000	1023.5G	PCI MEM Hi	CDWHB	(1)

1. Block read is guarded by mem_win_base, mem_win_mask register pair.
2. Write only space, bypass when video acceleration not enabled.
3. Block read is guarded by liocfg.

11.1.1 PCIX controller

The PCIX controller of STLS2F01 conforms to PCI-X 1.0b and PCI 2.3 specification. It can be configured as both host and agent mode. The configuration header located at 0x1fe00000, see *Table 55*.

Table 55. PCIX controller configuration header

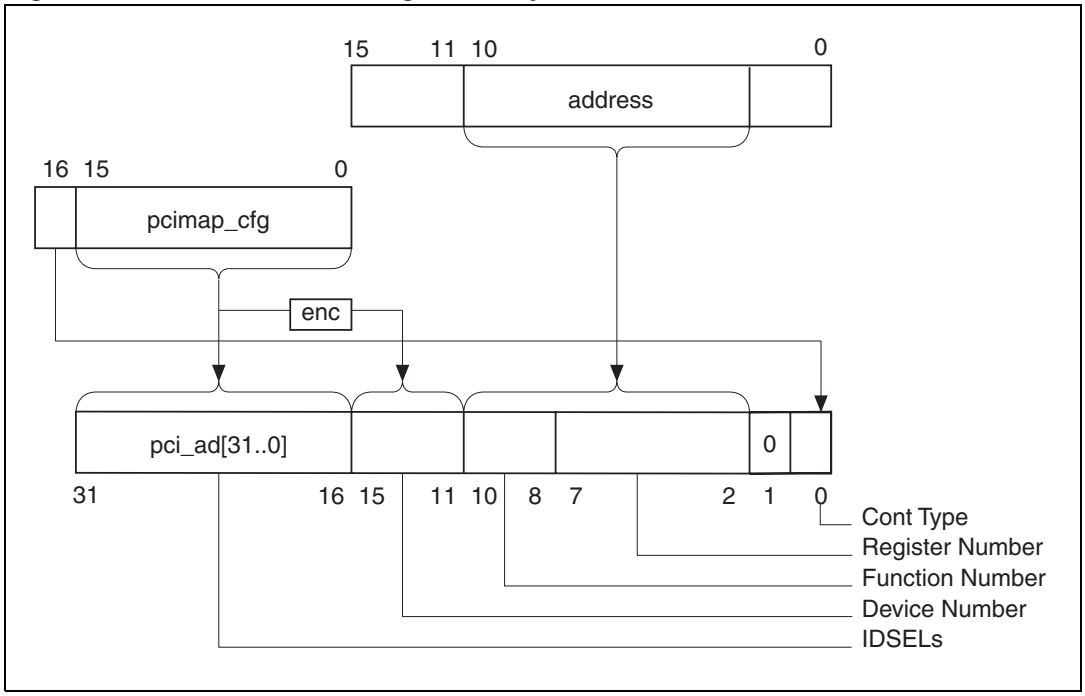
Byte 3	Byte 2	Byte 1	Byte 0	Offset
Device ID		Vendor ID		00
Status		Command		04
Class code			Revision ID	08
BIST	Header type	Latency timer	Cacheline size	0C
Base Address register 0				10
Base Address register 1				14
Base Address register 2				18
Base Address register 3				1C
Base Address register 4				20
Base Address register 5				24
				28
Subsystem ID		Subsystem Vendor ID		2C
				30
			Capabilities pointer	34
				38
Maximum latency	Minimum grant	Interrupt pin	Interrupt line	3C
PCIX Command Register				E0
PCIX Status Register				E4

Initiating configuration cycle

Before the application can initiate configuration cycle, the pcimap_cfg register must be written with appropriate values, telling the controller what the configuration cycle type and higher 16bits of the address cycle are. Following load/stores to 2K region starting at

0x1fe80000 are map to the indicated PCI device(see *Figure 20*). The device number is a priority encoding of pcimap_cfg[15:0].

Figure 20. Generation of configuration cycle address



11.1.2 LocalIO controller

The LocalIO controller provides a simple interface for accessing legacy IO devices. It's built for system booting and has two chip select pins. The data bus width and access delay can be individually configured (refer to CR08 liocfg). The wait parameter is the low period of liord or liowr signal measured in PCI clock cycles. See *Figure 21* and *Figure 22* for timing. When the data bus width is 16bits, the address is derived with the physical address shifting one bit right.

Figure 21. LocalIO read timing

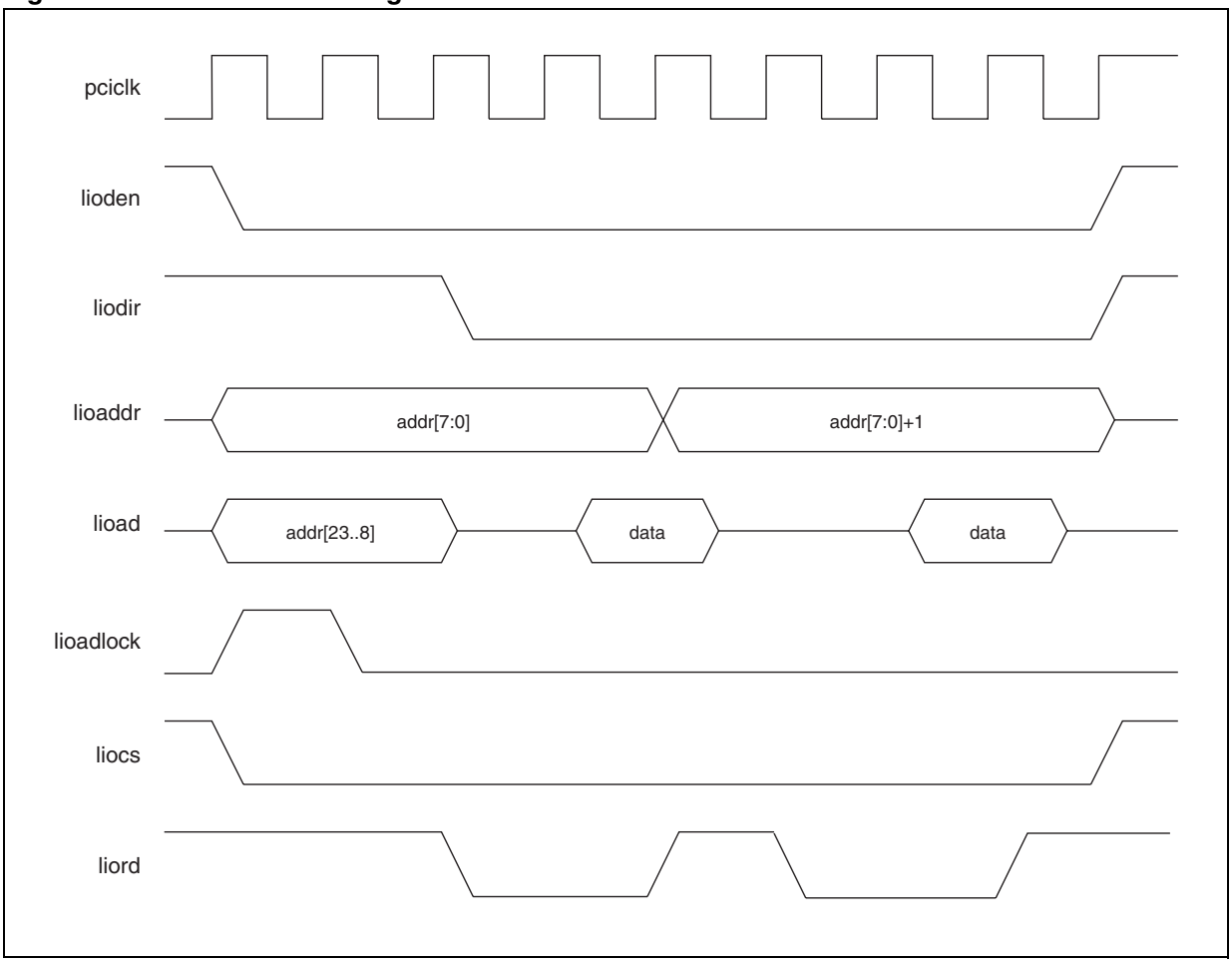
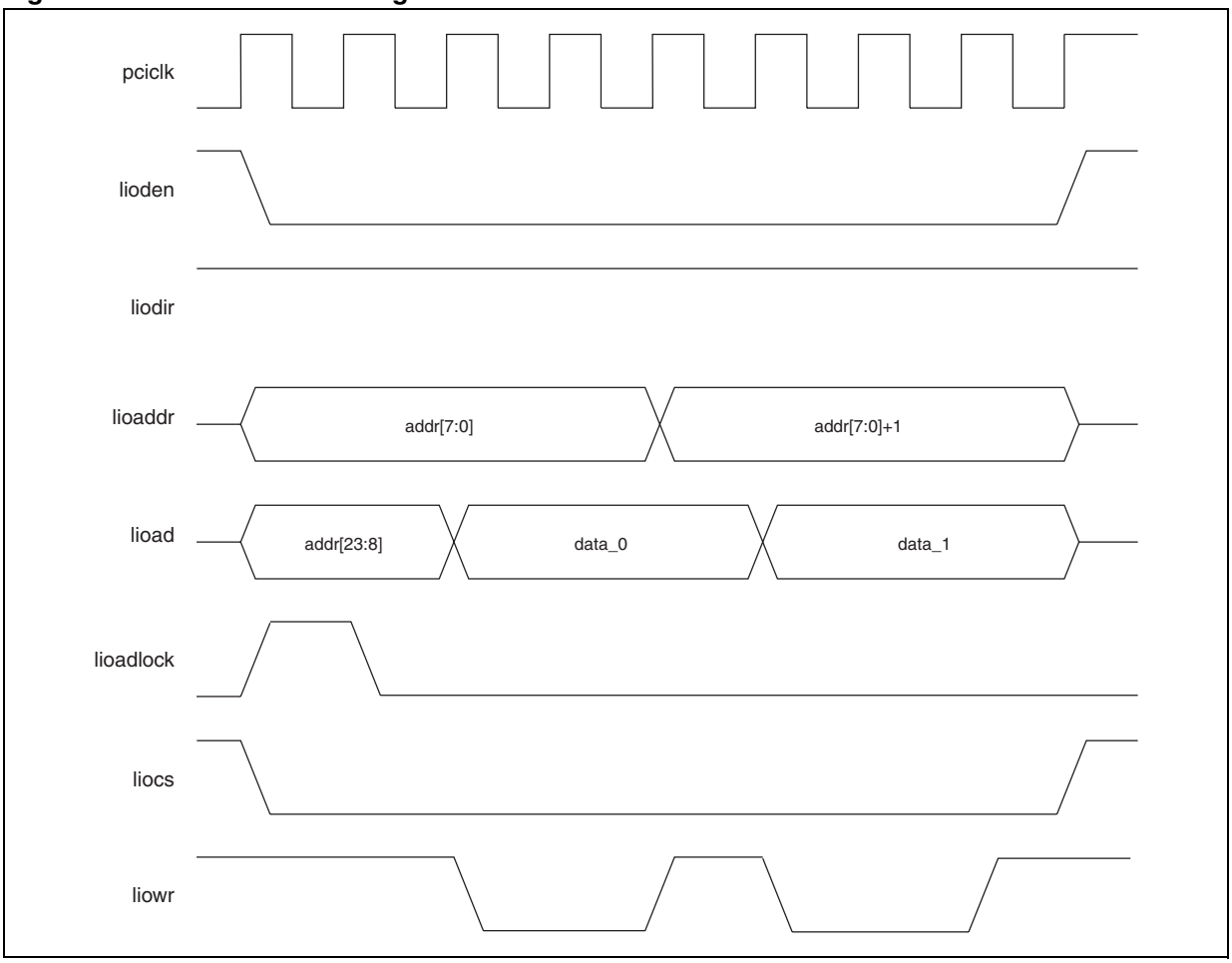


Figure 22. LocalIO write timing



11.1.3 Interrupt controller

There are several sources of interrupt in STLS2F01, both from inside the chip and pins. The interrupt controller is in charge of handling the line enable, polarity, pulse recording for each interrupt source. The polarities of all external interrupts are configurable, and reset to active low. INT0-3 is connected directly to CPU core int0-3, since the core has six masks for each external interrpt, the interrupt controller does not mask them. Other interrupts can be disabled. Pulse form interrupt (e.g. PCI_SERR) can be recorded by setting the integer bit. The recorded interrupt status can be clear by write the according bit at intenclr.

Table 56. Interrupt controller bit mappings

Field	Register			Int. Source
	intpol (acc/def)	intedge (acc/def)	inten (acc/def)	
3 : 0	RW / 0	RW / 0	RW / 0	GPIO
7 : 4	RO / 0	RO / 0	RW / 0	PCI_INTn
8	RO / 1	RO / 0	RW / 0	PCI_PERR

Table 56. Interrupt controller bit mappings (continued)

Field	Register			Int. Source
	intpol (acc/def)	intedge (acc/def)	inten (acc/def)	
9	RO / 1	RO / 1	RW / 0	PCI_SERR
10	RO / 1	RO / 1	RW / 0	denali
14 : 11	RW / 0	RW / 0	RO / f	INTn
31 : 15				Reserved

11.1.4 PCI/PCIX arbiter

The PCI/PCIX arbiter implements a two level round-robin arbitrating algorithm, bus parking and broken master isolation. Its config and status register are pxarb_config and pxarb_status. *Table 57* describes the request lines from the view of arbiter.

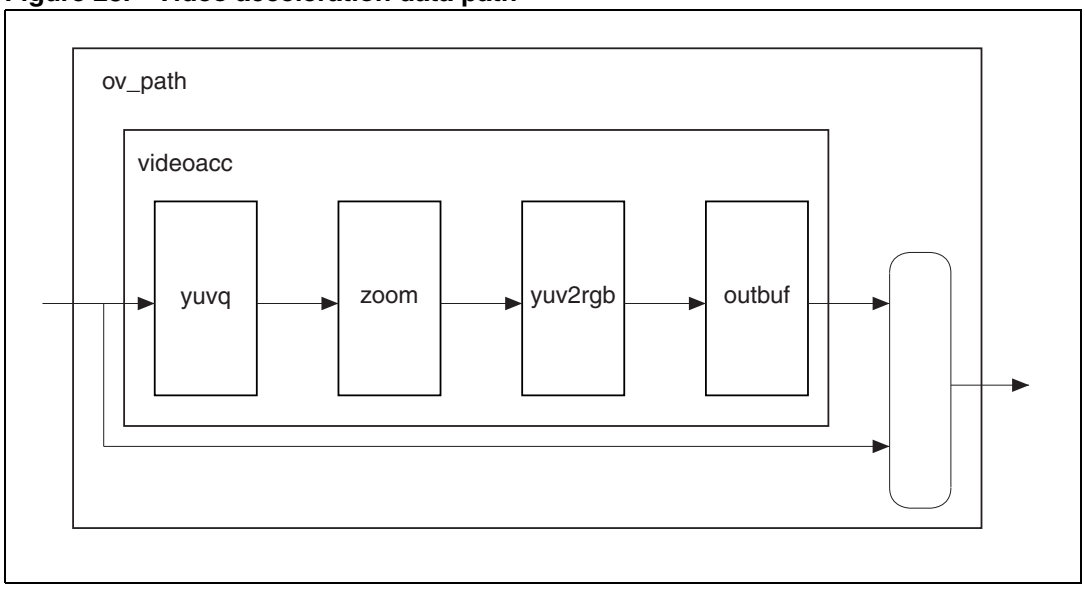
Table 57. PCI bus arbitration line routing

Line	Description
0	Internal request from PCI/PCIX bridge
7:1	External PCIREQ6~0

11.1.5 Video acceleration

Media application maps poorly in superscalar general purpose processor. The color space conversion and resize is such an example. They need only trivial computing, however, when taken by the core, there won't be any idle time. These jobs are offloaded from the core in Loongson 2F with small modification in data path (see *Figure 23*) to PCIX controller. When enabled, writes to 0x13f00000~0x13ffffff would interpret as original YUV data of a frame.

Figure 23. Video acceleration data path



Given the simplicity of the hardware, the software shall take more job and responsibility. Inappropriate setting would easily halt the controller. The ov_en (lease significant bit of gencfg) is global enable bit. When cleared, the acceleration unit is transparent. Note that, clearing this bit when the data is being send to graphic card frame buffer would be dangerous. Before changing the config registers, make sure ov_ctrl.reset is set.

Under YUV422 mode, frame data shall be written in the unit of 64 pixels. It consists of 32*4 bytes, with 32 bytes of Y in the head, 32 bytes of U and V in the middle and 32 bytes of Y at the tail.

Under YUV444 mode, frame data is also written as a unit of 64 pixels. It contains 32*6 bytes, in the order of YUVYUV, 32 bytes each.

For video acceleration to be meaningful, the media player shall use TLB entry with Uncache Accelerate property set.

11.2 Register description

11.2.1 Configuration Registers

All the configurable parts of IO controller except for PCI header is located at 256 bytes starting from 0x1fe00100. See *Table 58*.

Table 58. Controller registers

Address	Register	Description
00	poncfg	Power on config
04	gencfg	General config
08	liocfg	LocalIO config
0C	-	Reserved
10	pcimap	PCI mapping config
14	pcix_bridge_cfg	PCI/X bridge config
18	pcimap_cfg	PCI configuration access config
1C	gpio_data	GPIO data
20	gpio_en	GPIO input/output config
24	intedge	Interrupt pulse mode
28	-	Reserved
2C	intpol	Active interrupt level
30	intenset	Set interrupt enable
34	intenclr	Clear interrupt enable
38	inten	Interrupt enable status
3C	intisr	Interrupt status register
40	mem_win_base_l	Lower word of memory window base
44	mem_win_base_h	Higher word of memory window base
48	mem_win_mask_l	Lower word of memory window mask

Table 58. Controller registers (continued)

Address	Register	Description
4C	mem_win_mask_h	Higher word of memory window mask
50	pci_hit0_sel_l	Lower word of PCI image0 config
54	pci_hit0_sel_h	Higher word of PCI image0 config
58	pci_hit1_sel_l	Lower word of PCI image1 config
5C	pci_hit1_sel_h	Higher word of PCI image1 config
60	pci_hit2_sel_l	Lower word of PCI image2 config
64	pci_hit2_sel_h	Higher word of PCI image2 config
68	pxarb_config	PCIX arbiter config
6C	pxarb_status	PCIX arbiter status
70	-	Reserved
74	-	Reserved
8	-	Reserved
7C	-	Reserved
80	chip_config0	Chip config
84	pad1v8_ctrl	Chip config
88	pad3v3_ctrl	Chip config
8C	-	Reserved
90	comp_code	Chip sample
94	chip_sample1	Chip sample
98	-	Reserved
9C	-	Reserved
A0	ov_ctrl	Video accelerate control
A4	ov_ori_size	Original image size
A8	ov_zoom_size	Scaled image size
AC	ov_fb_base	Start address of first pixel in frame buffer
B0	ov_fb_stride	Distance in bytes between two vertical consecutive pixels in frame buffer
B4	ov_hor_zoom1	Horizontal zooming ctrl 1
B8	ov_hor_zoom2	Horizontal zooming ctrl 2
BC	ov_ver_zoom	Vertical zooming ctrl
C0	ov_x_pos	X coordinate of left most pixel
C4	ov_x_width	Screen width in pixels
C8	ov_fb_base	Frame buffer memory base
CC	ov_fb_mask	Frame buffer memory mask

Table 59. Detailed description of config registers

Field	Name	Access	Default	Description
CR00: poncfg				
15:0	pcix_bus_dev	RO	lio_ad[7:0]	Initial bus/device number for PCIX agent mode booting
15:8	-	RO	lio_ad[15:8]	Reserved
23:16	pon_pci_configi	RO	pci_configi	pci_configi pin value
31:24	-	RO		Reserved
CR04: gencfg				
0	ov_en	RW	0	Video accelerate enable
31:1	-	RO	0	Reserved
CR08: liocfg				
1:0	-	RO	0	Reserved
6:2	rom_wait	RW	5'b11111	Rom access delay cycles
7	rom_width	RW	pci_config[0]	Rom data width 0: 8 bits 1: 16 bits
12:8	io_wait	RW	5'b11111	IO access delay cycles
13	io_width	RW	1'b0	IO data width 0: 8 bits 1: 16 bits
14	iopf_en	RW	1'b0	IO prefetch enable 0: no block read 1: block readable
31:15	-	RO	0	Reserved
CR10: pcimap				
5:0	trans_lo0	RW	0	Higher 6 bits of translated PCI address for pci_mem_lo0 image
11:6	trans_lo1	RW	0	Higher 6 bits of translated PCI address for pci_mem_lo1 image
17:12	trans_lo2	RW	0	Higher 6 bits of translated PCI address for pci_mem_lo2 image
31:18	-	RO	0	Reserved
CR14: pcix_bridge_cfg				
5:0	pcix_rgate	RW	6'h18	Read issue threshold in PCIX mode
6	pcix_ro_en	RW	0	Relax order enable for PCIX bridge
31:18	-	RO	0	Reserved
CR18: pcimap_cfg				
15:0	dev_addr	RW	0	Higher 16 bits in configuration access

Table 59. Detailed description of config registers (continued)

Field	Name	Access	Default	Description
16	conf_type	RW	0	PCI configuration cycle type
31:17	-	RO	0	Reserved
CR1C: gpio_data				
3:0	gpio_out	RW	0	Output data source of GPIO
15:4	-	RO	0	Reserved
19:16	gpio_in	RW	0	GPIO input value
31:20	-	RO	0	Reserved
CR20: gpio_en				
3:0	gpio_en	RW	F	GPIO input enable
31:4	-	RO	0	Reserved
CR50,54/58,5C/60,64: pci_hit*_sel				
0	-	RO	0	Reserved
2:1	pci_img_size	RW	2'b11	00: 32 bits; 10: 64 bits; else: invalid
3	pref_en	RW	0	prefetch enable
11:4	-	RO	0	Reserved
62:12	bar_mask	RW	0	Bar size mask
63	burst_cap	RW	1	Burst capable
CR68: pxarb_config				
0	device_en	RW	1	External PCI master enable
1	disable_broken	RW	0	Disable broken master
2	default_mas_en	RW	1	Park bus to default master 0: park to the last bus master 1: park to default master
5:3	default_master	RW	0	Default master id
7:6	park_delay	RW	0	Delay from no master requesting bus to parking defalt master 00: 0 cycle 01: 8 cycles 10: 32 cycles 11: 128 cycles
15:8	level	RW	8'h01	Masters on the first level
23:16	rude_dev	RW	0	Device with bus holding requirement
31:13	-	RO	0	Reserved
CR6C: pxarb_status				
7:0	broken_master	RO	0	Broken master (cleared when disable broken policy)
10:8	last_master	RO	0	ID of last master that use bus

Table 59. Detailed description of config registers (continued)

Field	Name	Access	Default	Description
31:11	-	RO	0	Reserved
CR80: core_config				
2:0	freq_scale	RW	3'b111	Frequency scale control
3	disable_scache	RW	0	Disable second cache
4	imp_first	RW	1	Import word first
7:5	-	RW	0	Reserved
8	disable_ddr_conf	RW	0	Disable DDR2 config space
9	ddr_buffer_cpu	RW	1	Buffer CPU write to DDR2
10	ddr_buffer_pci	RW	1	Buffer PCI write to DDR2
31:11	-	RO	0	Reserved
CR84: pad1v8_ctrl				
0	compen	RW	0	
1	comptq	RW	0	
2	freeze	RW	0	
3	accurate	RW	0	
10:4	nasrc	RW	7'b1111000	
11	proga	RW	1	
12	progb	RW	0	
13	mod	RW	0	
14	strb	RW	0	
15	en	RW	0	
16	zoutproga	RW	0	
CR88: pad3v3_ctrl				
0	compen	RW	0	
1	comptq	RW	0	
2	freeze	RW	0	
3	accurate	RW	0	
10:4	nasrc	RW	7'b1111000	
CR90: compcode				
6:0	ddr2_asrc	RO		
7	-	RO		Reserved
14:8	pci_asrc	RO		
15	-	RO		Reserved
22:16	sys_asrc	RO		

Table 59. Detailed description of config registers (continued)

Field	Name	Access	Default	Description
32:23	-	RO		Reserved
CR94: chip_sample1				
9:0	sys_clksel	RO	sys_clksel	Value of PLL control pins
31:10	-	RO		Reserved

11.2.2 Video acceleration config registers

Table 60. Video acceleration config registers

Field	Name	Access	Default	Description
ov_ctrl:Video accelerate control				
0	reset	RW	0	Reset video acceleration unit. Writes shall delay long enough (e.g. 1ms) after this bit cleared
1	Y2R_EN	RW	0	YUV to RGB enable
2	ZoomEn	RW	0	Zoom enable
4:3	inFMT	RW	0	Input video format 01: YUV422 10: YUV444
6:5	outFMT	RW	0	Output frame buffer data format 00: RGB16 01: RGB24 10: RGB32
10:7	resolution	RW	0	Display resolution 0000: 320x200 0001: 320x350 0010: 360x400 0011: 640x200 0100: 640x350 0101: 640x480 0110: 720x350 0111: 720x400 1000: 800x600 1001:1024x768 1010:1280x1024 1011:1600x1200
ov_ori_size:Original image size				
10:0	X	RW	0	
21:11	Y	RW	0	
ov_zoom_size:Scaled image size				
10:0	X	RW	0	

Table 60. Video acceleration config registers (continued)

Field	Name	Access	Default	Description
21:11	Y	RW	0	
ov_fb_base:Start address of first pixel in frame buffer				
31:0	addr	RW	0	The address of the first pixel in frame buffer
ov_fb_stride:Frame buffer stride				
31:0	stride	RW	0	Distance in bytes between two vertical consecutive pixels in frame buffer
ov_hor_zoom1:Horizontal zooming ctrl 1				
10:0	ov_seg_size	RW	0	Size of 32 pixel segment after zoomed. (32/ratio+1) rounded down.
27:11	ov_stepx	RW	0	Horizontal zooming ratio. The original size divides by output size. Store as 5.12 fixed point binary.
ov_hor_zoom2:Horizontal zooming ctrl 2				
10:0	ov_last_seg_size	RW	0	The number of pixels in the last segment.
28:11	ov_size_mul_step	RW	0	Equal to ov_segment_size mul ov_stepx.
ov_ver_zoom:Vertical zooming ctrl				
16:0	ov_stepy	RW	0	Same as ov_stepx, only binary less or equal to 17'b00001_0000_0000_0000 accepted
ov_x_pos: X coordinate of left most pixel				
12:0	ov_x_pos	RW	0	Coordinate of output image's left most pixel(signed integer)
ov_x_width:Screen width in pixels				
10:0	ov_x_width	RW	0	Screen width in pixels
ov_fb_base:frame buffer memory base				
31:0	ov_fb_base	RW	0	Frame buffer memory base
ov_fb_mask: frame buffer memory mask				
31:0	ov_fb_mask	RW	0	Frame buffer memory mask Writes outside this region will not send to PCI bus.

12 Performance tuning

This chapter describes some architecture impacts of STLS2F01 on software and ways to make efficient software for STLS2F01. The STLS2F01 architecture, like all other RISC architectures, depends on careful attention of data alignment and instruction scheduling to achieve high performance.

12.1 User instruction latency and repeat rate

Table 61 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 61. Latencies and repeat rates for user instructions

Instructions	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

Table 61. Latencies and repeat rates for user instructions (continued)

Instructions	Unit	Latency	Repeat Rate
Sqrt.d	FALU2	5-32	4-31
Lwc1,Ldc1	MEM	5	1
Swc1,Sdc1	MEM	-	1

Please note the following about *Table 53*:

- 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.

12.2 Instruction extensions

STLS2F01 implements several instruction extensions.

- Fixed-point multiplies and divisions 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.

- Fixed-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.

12.3 Instruction stream

The following sections describe considerations for the instruction stream.

12.3.1 Instruction alignment

Every cycle STLS2F01 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.

12.3.2 Branch handling

In STLS2F01 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 STLS2F01 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:

- STLS2F01 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.
- 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 STLS2F01 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 STLS2F01 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 STLS2F01 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.

12.3.3 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.

12.3.4 Instruction scheduling

STLS2F01 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 STLS2F01 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.

12.4 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 slower 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 STLS2F01 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 STLS2F01 processors (even a cache-hit operation takes 5 cycle in STLS2F01), 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.

12.5 Other tips

- Utilize all floating-point registers. STLS2F01 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. STLS2F01’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.

13 MIPS compliancy

The design goal of STLS2F01 microprocessor is to be compatible to MIPS III architecture. The ISA (Instruction Set Architecture) of STLS2F01 processor is almost the same with MIPSIII ISA.; only a few unimportant differences still exist. On the contrary, MIPS III architecture did not define its PRA (Privilege Resource Architecture). In practice, however, OS kernel developers always use ‘traditional’ MIPS processors R4000 and R10000 as the standard of MIPS privilege architecture. Moreover, in recent years, MIPS Inc. defines MIPS64 ISA as the superset of MIPS III, in which the PRA is defined. STLS2F01 obeys most of MIPS10000 privilege architecture as well as the MIPS64 PRA, but there are still some significant differences. This chapter describes the STLS2F01 architecture’s difference with ’traditional’ MIPS processors, including the following sections: the overview of the STLS2F01 processor’s compatibility, the special CP0 features, and the special CPU and FPU instruction features.

13.1 The compliance overview

The STLS2F01 processor is the enhanced version of Loongson-2E processor. The most architecture design in STLS2F01 is same with the Loognson-2E processor. Some incompatible features in the Loongson-2E processor are modified and this makes STLS2F01 processor “more compatible” with MISP ISA 64 than Loongson-2E processor.

The STLS2F01 processor’s instruction set includes the whole MIPS III instruction set. It is also extended with some new instructions to enhance the performance of the floating-point/ fixed-point computation and the multimedia application. The STLS2F01 processor’s instruction set consists of two parts: MIPS III instructions instruction set and STLS2F01 enhanced instructions, including SIMD instructions for media and some other instructions. A MIPS III binary program can run on STLS2F01 perfectly without any modification, while programmers can further improve the software by adding STLS2F01’s special instructions.

Please be noted that Loongson2E cannot be categorized to ‘MIPS compatible’, as some of its enhanced instructions occupies the reserved instruction encoding space of MIPS III. It is still safe till the time being, but once MIPS Inc. defines some new instructions in these reserved encoding slots and their semantics are different with what Loongson2E defines, problem will occur. In the STLS2F01 processor, these enhanced instructions are transferred to the safe encoding slots that are available to MIPS partner and can not trigger any problem in the future MIPS instructions. The encoding of these instructions can be seen in the Appendix.

In terms of user-level instruction set architecture, they are always similar among the MIPS compatible processors or MIPS like processors such as STLS2F01. However, the privileged instructions or resources for processor control used by the implementation-specific system control coprocessor can be greatly different between difference processors as there is no privilege resource architecture definition before MIPS32/64 standard. In practice, however, MIPS processor designers always tried to make it similar with the implementation of R4000/R10000. The STLS2F01 processor‘s privilege resource architecture is also more or less similar with MIPS R4000/R10000. In the other hand, it still contains some special features which are different with the mainstream MIPS III processors. These special CP0 features are described in *Chapter 13.2 on page 146*.

In user instruction architecture level, the STLS2F01’s MIPS III compliant instructions have the same format and operation with the instruction defined in the MIPS III specification.

These instructions run on the STLS2F01 processor and other MIPS III processors, such as R4000 or R10000, behaving identically. For the implementation reasons, however, there are two instructions that have slightly different semantics between STLS2F01 and MIPS R10000. Although they are quite unimportant and almost ignorable, the programmers should not presume they are 100% identical. These different points are described in *Chapter 13.3 on page 149*.

The multimedia instructions are the Single Instruction Multiple Data (SIMD) instructions which extend the STLS2F01 architecture to enhance the performance of advanced media and communication application. These instructions are the SSE like instructions and differentiate with all MIPS instructions. The description about the STLS2F01 processor's multimedia instructions can be seen in the *Appendix C* The STLS2F01 Multimedia Technology.

The STLS2F01 processor implements several special MIPS IV instructions as the supplement to the MIPS III instructions. These instructions include two MIPS IV instructions (i.e. MOVZ and MOVNZ) and four MIPS IV like instructions which perform a combined multiply-accumulate of FP value with three operands instead of the four operands in MIPS IV specification. The description about the STLS2F01 processor's special MIPS IV instruction can be seen in the *Appendix B* STLS2F01 new floating point instruction.

The STLS2F01 processor defines twelve special fixed-point multiply and divide instructions to extend the MIPS's fixed-point multiply and divide instructions to enhance the performance of fixed-point multiply and divide operation extensive applications. These instructions perform the multiply, divide or module operation of 64-bit fixed-point values, and produce the 64-bit result instead of the 128-bit result. The descriptions about these fixed-point instructions can be seen in the *Appendix A* STLS2F01 new integer instructions.

13.2 The special CP0 features

The MIPS defines the privilege architecture as the part of the specifications, i.e. CP0. To allow flexibility in implementations, MIPS provides the subsetting rules to allow the designs to only implement a set of required features to be compliant with MIPS architecture. The STLS2F01 processor implements the part of MIPS privilege architecture as well as some special features that may be not compliant with the MIPS privilege architecture specifications. The programmers should consider this non-compatible privilege architecture when porting the system software to the STLS2F01 processor. The following sections describe the special features which differentiate with other MIPS III compatible processor's CP0, such as R4000 or R10000.

13.2.1 The ITLB flushing

The STLS2F01 contains the separate instruction TLB (i.e. ITLB) to minimize contention for the joint TLB and to reduce power dissipation. When a miss occurs on an instruction address translation by the ITLB, a randomly selected ITLB entry is filled from the joint TLB. The refill and lookup operation of the ITLB is completely transparent to the user.

However, ITLB does not have auto-flush function. When the address mapping in the JTLB is changed by the TLBWI or TLBWR instructions, ITLB can not be updated automatically by the hardware to maintain the coherence with the JTLB. Therefore, programmers must use instructions to flush the ITLB after writing to JTLB. For the same reason, whenever any field in the JTLB entry which was already filled in the ITLB is modified, such as one page is invalidated or the mask of page size is changed, ITLB must be flushed by software.

The method to flush ITLB is to set the DiagI bit in CP0 diagnostic register.

Each ITLB entry can not be flushed individually, and flushing the whole ITLB is the unique operation to discard the conflicting ITLB entry. The operation of the ITLB flushing can be seen in the following example.

A TLB entry with different PFN values as supplied by the t1 and t0 parameters is created. The old TLB entry indexed by t5 parameters is valid and is already filled in ITLB by the previous execution. In R4000 and R10000's TLB design, the old TLB entry will be filled by the new TLB entry after the TLBWI instruction do a TLB write. Since in the STLS2F01 processor the ITLB can not be updated automatically by the hardware, after the new TLB entry is written into JTLB by the TLBWI instruction, the ITLB must be flushed by software.

```
/*  The description of the parameters are
     t4 - 32 bit Virtual address
  t3 - ASID value
  t1 - 32 bit physical address for EntryLo0
  t0 - 32 bit physical address for EntryLo1
  attr0 - TLB attribute for EntryLo0
  attr1 - TLB attribute for EntryLo1
  t5 - Index value of the TLB entry
  t6   - temp register             */


  srl       t6, t4, 13;           /* Clean up lower order bits      */
  sll       t6, t6, 13;           /* Pad zeros                      */
  or        t6, t6, t3;           /* Include the ASID value         */
  mtc0      t6, C0_EntryHi;       /* Write to entry Hi register     */
  srl       t6, t1, 6;            /* align PFN for entry Lo reg     */
  sll       t6, t6, 6;
  ori       t6, t6, attr0;        /* Include the attribute field    */
  mtc0      t6, C0_EntryLo0;      /* Write to entry lo0 reg         */
  srl       t6, t0, 6;            /* align PFN for entry Lo reg     */
  sll       t6, t6, 6;
  ori       t6, t6, attr1;        /* Include the attribute field    */
  mtc0      t6, C0_EntryLo1;      /* Write to entry lo1 reg         */
  mtc0      t5, C0_Index;         /* Write to Index register        */
  tlbwi;                          /* Do a TLB write                 */
  li        k1, (0x1 << 2);       /* Set ITLB flushing bit          */
  mtc0      k1, C0_Diag           /* Write to Diagnostic reg        */
```

13.2.2 The diagnostic register

The Diagnostic register, CP0 register 22, is a supplemental 64-bit register for STLS2F01 specific diagnostic functions. This register handles ITLB flushing, BTB (branch target buffer) flushing and RAS (return address stack) enabling. The I field (Bit 2) in the Diagnostic register handles ITLB flushing. When write bit 1 into the DiagI bit, the ITLB is flushed. It should be considered carefully that the unused fields in diagnostic register must be written as zero, and return zero when read. The behavior of the STLS2F01 processor is UNDEFINED if the unused fields of diagnostic register are not written as zero.

13.2.3 The performance counter register

Two supplemental registers, CP0 register 24 and 25, are the powerful performance counter registers specially implemented in the STLS2F01 processor. They are able to count many kinds of important events or cycles in order for helping processor's performance analysis.

In MIPS64 ISA compatible processors, CP0 register 24 is the DEPC register for the EJTAG debug function. The CP0 register 25 are defined as Performance Counter register by the MIPS specification. The STLS2F01 processor's Performance Counter registers contain the similar function fields with it but are mapped into the different select value. The detail descriptions about the Performance Counter register can be seen in the *Chapter 5: CP0*. The unused fields in STLS2F01 processor's performance counter registers must be written as zero, and return zero when read. The behavior of the STLS2F01 processor is UNDEFINED if the unused fields are not written as zero.

13.2.4 The CacheErr exception

The STLS2F01 processor can not detect a cache tag or data error, or a parity or ECC error on the system bus since the parity or ECC in the Cache, or on the system bus, is not implemented. The Cache Error exception can not be generated in any conditions. The CacheError register, CP0 register 27, is also reserved in the STLS2F01 processor.

13.2.5 Address translation for the kuseg segment when statusERL = 1

In the MIPS64 compatible processors, when a CacheError exception happens due to an ECC error, the processor is able to bypass cache, fix them and keep running. Since the exception handler has no register safely used, the processor provides one uncached and unmapped window from 0 through 0x7FFFFFFF to help saving register values directly to uncached memory by using base + offset address off zero register. Since the CacheErr exception is not implemented, the STLS2F01 processor needs not to provide the support for the cache error handler. Hence, the kuseg segment always becomes the mapped and cached segment no matter what the ERL bit in the Status register is. The reference address accessing the user space is translated by the TLB in all processor modes. If the address translation is failed, the TLB exception will be generated immediately.

13.2.6 Exception return when statusERL = 1

The ERET instruction returns from the exception, interrupt or error trap. When a Reset, Soft Reset, NMI or Cache Error exception is taken, the ERL bit is set by the processor. In the MIPS64 compatible processor, when StatusERL = 1, the ERET instruction will clear StatusERL bit and return form ErrorEPC register. In the STLS2F01 processor, when ERL of Status register is set, the ERET instruction will clear StatusERL and return from CP0 EPC register, not from the ErrorEPC register.

13.2.7 Page size setting in the TLB entries

The pages are defined as the blocks in which the translated virtual addresses retrieve the data. In the STLS2F01 processor, the size of page is variable and may be selected from 4 Kbytes to 16 Mbytes inclusive, in power of 4 (that is, 4 Kbytes, 16 Kbytes, 64 Kbytes, 256 Kbytes, 1 Mbytes, 4 Mbytes or 16 Mbytes.). The page size can be set in TLB entry's Mask field, as described in the *Chapter 3.3.1: Format of a TLB Entry*. This field can be read or written through the PageMask register, described in *Chapter 5.5: PageMask register (5)*.

It must be considered that in the STLS2F01 processor, all TLB entries share a unique Mask field, and each TLB entry's page size can not be set individually. That is, whenever the comparison mask for the page size is set by using TLBWI or TLBWR instruction through the PageMask register, the size of each page mapped in the TLB entries will be modified. It should be considered carefully that the page size in all the TLB entries ahead will be changed when the new page with the different page size is created in the TLB. It is recommended that the page size should be not changed.

13.2.8 The 64-bit address space

In the MIPSISA64 compatible processors, the address space can be select from the 32-bit or 64-bit memory address. The processor uses either 32-bit or 64-bit address space depending on the operating mode (user, supervisor or kernel) and addressing mode set by the Status register (i.e. UX, SX or KX bit). For example, the user space is 64-bit memory address when the UX bit in the Status register is one, no matter what the operation mode of the processor is. That is, at that time the user space xkuseg can be accessible when the processor is in the kernel mode. When UX bit in the Status register is zero, the reference to the user space (xuseg, xsuseg or xkuseg) is invalid, no matter what the operation mode of processor is. The SX bit in the Status register control whether the supervisor space (xsseg or xksseg) is accessible. The KX bit in the Status register control whether the reference for the kernel space (kxseg or kxphys) is valid.

In the STLS2F01 processor, however, the 64-bit address space is always enabled. No matter what the operation mode of processor is, the reference for the user space (xuseg, xsuseg or xkuseg), the supervisor space (xsseg or xksseg) or the kernel space (kxseg or kxphys) is always valid. The *Chapter 3: Memory management* describes the detail region of the STLS2F01 processor's virtual address spaces. The KX, SX or UX bit in the Status register is always a one, and can not be written to zero by the software. Software can not change the user/supervisor/kernel address space to 32-bit address.

Since the STLS2F01 processor can not invalidate the 64-bit address space, the default TLB refill exception is the XTLB refill exception. When a TLB miss occurs, the choice of the Exception Vector is not required to be determined by the 64-bit address enable. The TLB Refill vector is used when the XTLB refill exception is taken. The XTLB Refill exception vector in the STLS2F01 processor has the same location as the TLB refill exception vector, i.e. 0x000. This STLS2F01 processor's feature differentiates with other MIPSISA64 compatible processors.

13.3 The special CPU and FPU instructions features

ALL MIPS III instructions can be run on the STLS2F01 processor without any exceptions. But for the implementation reasons, the STLS2F01 processor has the following two special features which are different with other MIPS III processors: one is for the load-to-zero instruction and other is for the floating point conversion instructions.

13.3.1 The special feature for the load-to-zero instruction

In normal operations, the load instruction will access the memory location and fetch the data back into one of the general registers. When the reference address is not an effective address or the address translation is failed, an address error or a TLB exception will be generated. The zero register is the read-only register in MIPS architecture. All data written into zero register will be ignored. In the STLS2F01 processor, a load instruction, such as

LB/LH/LW/LD etc, whose target is zero register, will not lead to any exception even if the reference address is not in TLB or is an error. Actually, the STLS2F01 processor uses load-to-zero instruction to implement the prefetch function, while there is no dedicated prefetch instruction. In the case that the reference address is error or not in the TLB, the operation of load-to-zero instruction will be same as the nop instruction. For example, in the following code the load-to-zero instruction has the different behaviors.

It is the assumption that TLB entry with Lo0 and Lo1 has the different PFN; no data in the physical page mapped in TLB are in the Cache. The load-to-zero instruction will prefetch the data to the Cache. When ASID is changed, any access to the mapped page should generate TLB refill exception, except that the load-to-zero instruction will not. Since ASID is different, the address translation of load-to-zero instruction will fail, and load instruction will be same as the nop instruction.

```
/* the parameter is that: t2 - the Virtual address in the first page */

      lb        zero, 0x0(t2);          # prefetch the data to cache
      mfc0      t1, C0_EntryHi;         # read the EntryHi reg
      addiu     t1, t1, 0x1;            # Change the ASID value.
      mtc0      t1, C0_EntryHi;
      lb        zero, 0x20(t2);         # nop operation without TLB exception
      addiu     t2, t2, K_PageSize;     # Use the next page for reference.
      lb        zero, 0x0(t2);          # nop operation
      lb        t3, 0x20(t2);           # generate the TLB exception
```

13.3.2 The special feature for the floating point conversion instructions

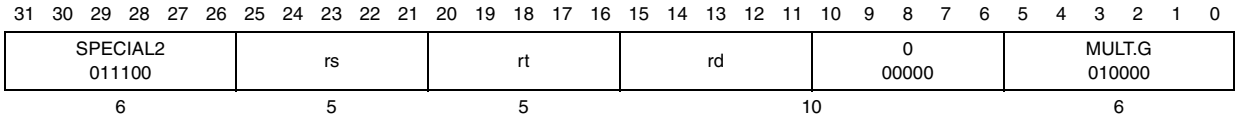
The floating point conversion instructions perform the conversions between the floating-point data types and fixed-point data types in the floating point registers. The floating point data types represent the numeric values, such as the following: normalized value, Non-Numbers (SNan and QNan), Infinities (+8 or -8), zero (+0 or -0) etc. The fixed point data types represent the word and longword fixed point data. For the conversion instructions which convert the floating-point data types to fixed-point data types, the result can not be represented correctly when the source value is Infinity, NaN, or rounds to an integer outside the range that the fixed point data can be represent. In such conditions, an IEEE Invalid Operation condition exists, and the STLS2F01 processor sets the Invalid Operation flag in the FCSR. If the Invalid Operation Enable bit is set in the FCSR, the STLS2F01 processor dose not write any result to the floating point destination register and an Invalid Operation exception is taken immediately. Otherwise, the Loonson-2F processor will take the following operation which is slightly different with the MIPS specifications. (MIPS specifications don't differentiate the positive and negative input values.)

- For the instructions converting floating point data to word fixed point data, such as cvt.w.fmt, round.w.fmt, floor.w.fmt, ceil.w.fmt, trunk.w.fmt, when the source value is Infinity, NaN, or rounds to an integer outside the range - to -1, and the Invalid Operation Enable bit is not set in the FCSR, the STLS2F01 processor will not generate the Invalid Operation exception. In this case, when the input operand is positive, 0x7FFFFFFF is written to the destination register. When the input operand is negative, 0x80000000 is written to the destination register.
- For the instructions converting floating point data to longword fixed point data, such as cvt.l.fmt, round.l.fmt, floor.l.fmt, ceil.l.fmt, trunk.l.fmt, when the source value is Infinity, NaN, or rounds to an integer outside the range - to -1, and the Invalid Operation Enable bit is not set in the FCSR, the STLS2F01 processor will not generate the Invalid Operation exception. In this case, when the input operand is positive,

0x7FFFFFFFFFFFFFFF is written to the destination register. When the input operand is negative, 0x8000000000000000 is written to the destination register.

Appendix A STLS2F01 new integer instructions

A.1 MULT.G - multiply word (STLS2F01)



Format: MULT.G rd, rs, rt

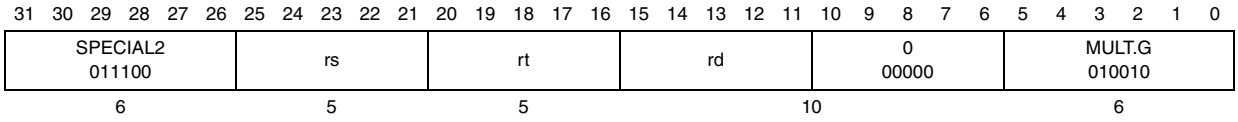
Purpose: To multiply 32-bit signed intergers.

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$_{31..0}$)

Exception: None

A.2 MULTU.G - multiply unsigned word (STLS2F01)



Format: MULTU.G rd, rs, rt

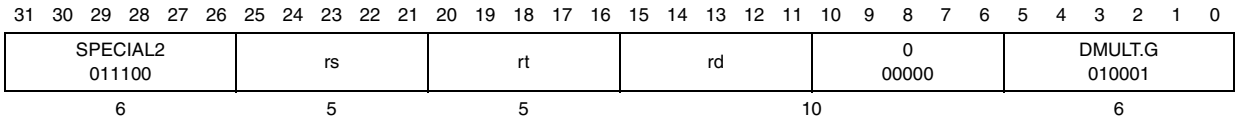
Purpose: To multiply 32-bit unsigned intergers.

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$_{31..0}$)

Exception: None

A.3 DMULT.G - doubleword multiply (STLS2F01)



Format: DMULT.G rd, rs, rt

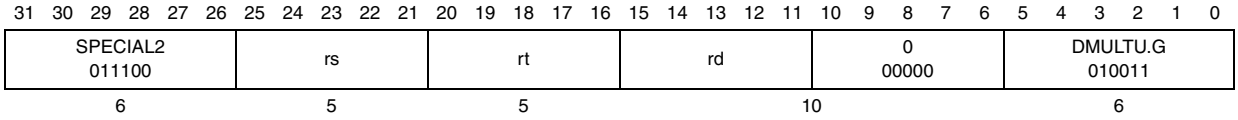
Purpose: To multiply 64-bit signed intergers.

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 ← prod63..0

Exception: None

A.4 DMULTU.G - doubleword multiply unsigned (STLS2F01)



Format: DMULT.G rd, rs, rt

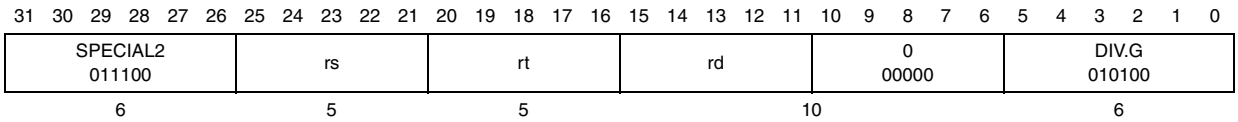
Purpose: To multiply 64-bit unsigned intergers.

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 ← prod63..0

Exception: None

A.5 DIV.G - divide word (STLS2F01)



Format: DIV.G rd, rs, rt

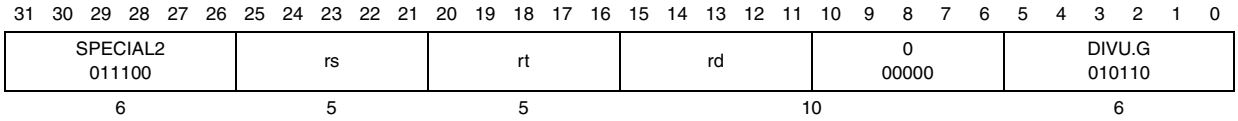
Purpose: To divide 32-bit signed intergers.

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 ← GPR[rs]31..0 div GPR[rt]31..0
LO ← sign_extend(q31..0)

Exception: None

A.6 DIVU.G - divide unsigned word (STLS2F01)



Format: DIVU.G rd, rs, rt

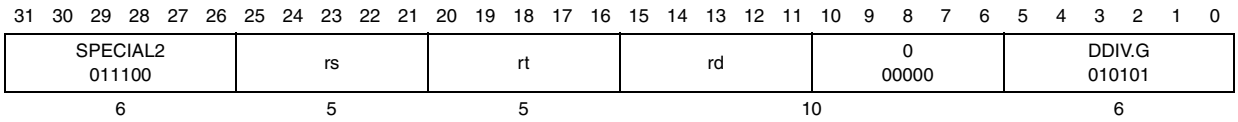
Purpose: To divide 32-bit unsigned intergers.

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

Exception: Reserved Instruction

A.7 DDIV.G - doubleword divide (STLS2F01)



Format: DDIV.G rd,rs, rt

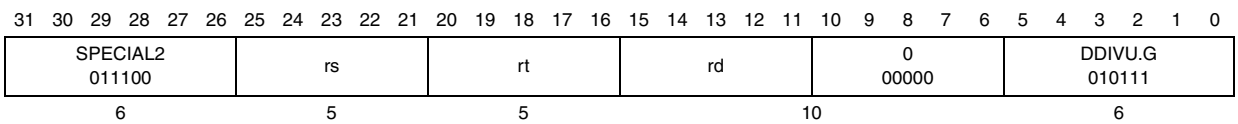
Purpose: To divide 64-bit signed intergers.

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]

Exception: None

A.8 DDIVU.G - doubleword divide unsigned (STLS2F01)



Format: DDIVU.G rd, rs, rt

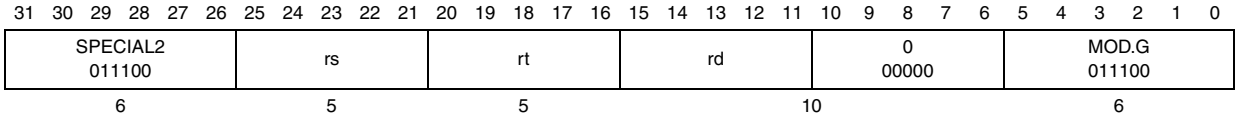
Purpose: To divide 64-bit unsigned intergers.

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])

Exception: None

A.9 MOD.G - mod word (STLS2F01)



Format: MOD.G rd, rs, rt

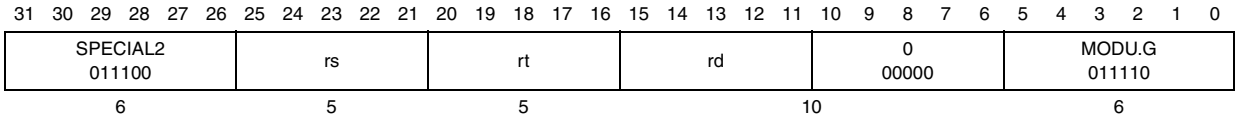
Purpose: To mod 32-bit signed intergers.

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

Exception: None

A.10 MODU.G - mod unsigned word (STLS2F01)



Format: MODU.G rd, rs, rt

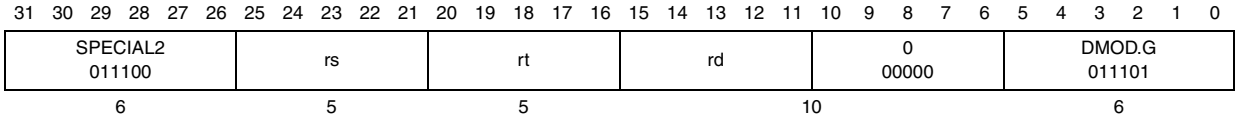
Purpose: To mod 32-bit unsigned intergers.

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$_{31..0}$)

Exception: Reserved Instruction

A.11 DMOD.G - doubleword mod (STLS2F01)



Format: DMOD.G rd, rs, rt

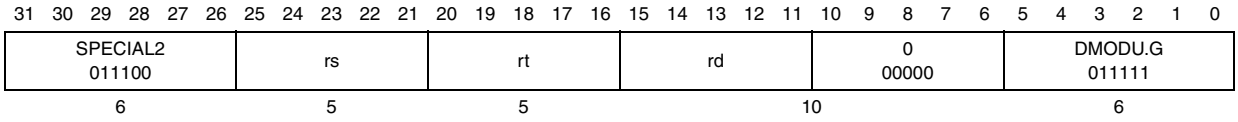
Purpose: To mod 64-bit signed intergers.

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]

Exception: None

A.12 DMODU.G - doubleword mod unsigned (STLS2F01)



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])

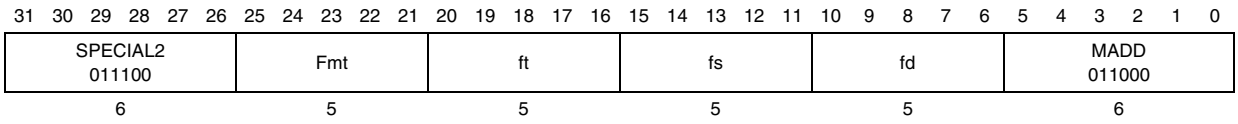
Exception: None

Appendix B STLS2F01 new float-point instructions

Table 62. Paired-single (PS) instructions in STLS2F01 FPU

Fmt Op	Fmt=22
ADD	Add.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

B.1 MADD.fmt - floating-point multiply add



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.

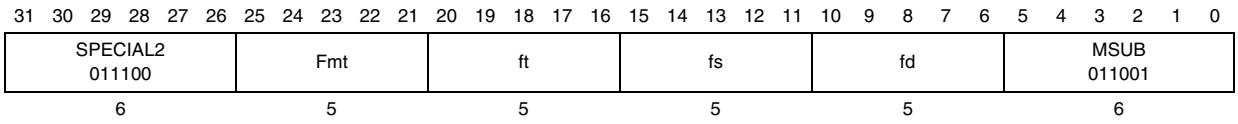
Operation:

vfd ← ValueFPR(fd, fmt)
vfs ← ValueFPR(fs, fmt)
vft ← ValueFPR(ft, fmt)
StoreFPR(fd, fmt, vfd + vfs * vft)

Exception:

Coprocessor Unusable
Reserved Instruction
Floating-Point
Inexact Unimplemented Operation Unimplemented Operation
Invalid Operation Overflow Overflow
Underflow

B.2 MSUB.fmt - floating-point multiply subtract



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.

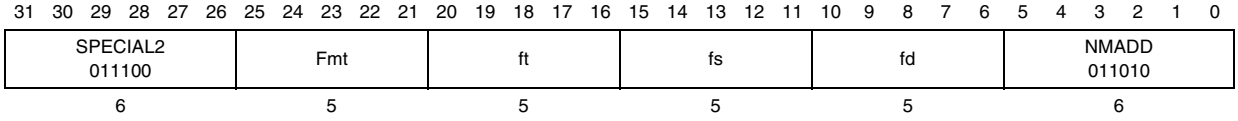
Operation:

vfd ← ValueFPR(fd, fmt)
vfs ← ValueFPR(fs, fmt)
vft ← ValueFPR(ft, fmt)
StoreFPR(fd, fmt, (vfs * vft)-vfd)

Exception:

Coprocessor Unusable
Reserved Instruction
Floating-Point
Inexact Unimplemented Operation Unimplemented Operation
Invalid Operation Overflow Overflow
Underflow

B.3 NMADD.fmt - floating-point negative multiply add



Format: NMADD.S fd, fs, ft
NMADD.D fd, fs, ft

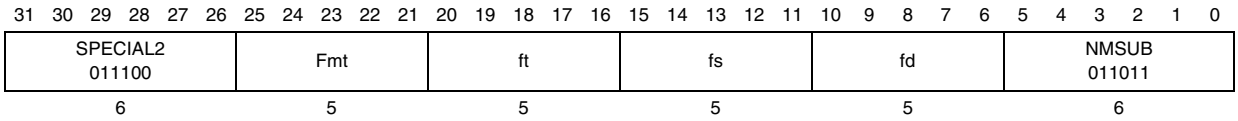
Purpose: To negate a combined multiply-then-add of FP values.

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))
```

Exception: Coprocessor Unusable
Reserved Instruction
Floating-Point
Inexact Unimplemented Operation Unimplemented Operation
Invalid Operation Overflow Overflow
Underflow

B.4 NMSUB.fmt - floating-point negative multiply subtract



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))
```

Exception: Coprocessor Unusable
Reserved Instruction
Floating-Point
Inexact Unimplemented Operation Unimplemented Operation
Invalid Operation Overflow Overflow
Underflow

Appendix C STLS2F01 multimedia technology

C.1 Overview

The media extensions for the Loongson Architecture were designed to enhance performance of advanced media and communication applications. The Loongson Multimedia technology provides a new level of performance to computer platforms by adding new instructions and defining new 64-bit data types, while preserving compatibility with software and operating systems developed for the Loongson Architecture. The Loongson Multimedia technology introduces new general-purpose instructions. These instructions operate in parallel on multiple data elements packed into 64-bit quantities. They perform arithmetic and logical operations on the different data types. These instructions accelerate the performance of applications with compute-intensive algorithms that perform localized, recurring operations on small native data. This includes applications such as motion video, combined graphics with video, image processing, audio synthesis, speech synthesis and compression, telephony, video conferencing, 2D graphics, and 3D graphics.

The Loongson Multimedia instruction set has a simple and flexible software model with no new mode or operating-system visible state. The Loongson Multimedia instruction set is fully compatible with all Loongson Architecture microprocessors. All existing software continues to run correctly, without modification, on microprocessors that incorporate the Loongson Multimedia technology, as well as in the presence of existing and new applications that incorporate this technology.

The Loongson Multimedia technology uses the Single Instruction, Multiple Data (SIMD) technique. This technique speeds up software performance by processing multiple data elements in parallel, using a single instruction. The Loongson Multimedia technology supports parallel operations on byte, halfword, and word data elements, and doubleword integer data type.

Modern media, communications, and graphics applications now include sophisticated algorithms that perform recurring operations on small data types. The Loongson Multimedia technology directly addresses the need of these applications. For example, most audio data is represented in 16-bit (halfword) quantities. The Loongson Multimedia instructions can operate on four of these words simultaneously with one instruction. Video and graphics information is commonly represented as palletized 8-bit (byte) quantities; one Loongson Multimedia instruction can operate on eight of these bytes simultaneously.

C.2 Instruction syntax

Instructions vary by:

- Data type: packed bytes, packed half words, packed words or doublewords
- Signed - Unsigned numbers
- Wraparound - Saturate arithmetic

A typical Loongson Multimedia instruction has this syntax:

- Prefix: **P** for Packed
- Instruction operation: for example - ADD, CMP, or XOR
- Suffix:
  --**US** for Unsigned Saturation
  --**S** for Signed saturation

--**B, H, W, D** for the data type: packed byte, packed halfword, packed word, or doubleword.

Instructions that have different input and output data elements have two data-type suffixes. For example, the conversion instruction converts from one data type to another. It has two suffixes: one for the original data type and the second for the converted data type.

This is an example of an instruction mnemonic syntax:

PADDUSW (Packed Add Unsigned with Saturation for Word)

P = Packed

ADD = the instruction operation

US = Unsigned Saturation

W = Word

C.3 Saturation and wraparound modes

When performing integer arithmetic, an operation may result in an out-of-range condition, where the true result cannot be represented in the destination format. For example, when performing arithmetic on signed halfword integers, positive overflow can occur causing the true signed result is larger than 16 bits.

The Loongson Multimedia technology provides three ways of handling out-of-range conditions:

- Wraparound arithmetic.
- Signed saturation arithmetic.
- Unsigned saturation arithmetic.

With wraparound arithmetic, a true out-of-range result is truncated (that is, the carry or overflow bit is ignored and only the least significant bits of the result are returned to the destination). Wraparound arithmetic is suitable for applications that control the range of operands to prevent out-of-range results. If the range of operands is not controlled, however, wraparound arithmetic can lead to large errors. For example, adding two large signed numbers can cause positive overflow and produce a negative result.

With signed saturation arithmetic, out-of-range results are limited to the representable range of signed integers for the integer size being operated on. For example, if positive overflow occurs when operating on signed halfword integers, the result is “saturated” to 7FFFH, which is the largest positive integer that can be represented in 16 bits; if negative overflow occurs, the result is saturated to 8000H.

With unsigned saturation arithmetic, out-of-range results are limited to the representable range of unsigned integers for the integer size being operated on. So, positive overflow when operating on unsigned byte integers results in FFH being returned and negative overflow results in 00H being retuned.

Saturation arithmetic provides a more natural answer for many overflow situations. For example, in color calculations, saturation causes a color to remain pure black or pure white without allowing inversion. It also prevents wraparound artifacts from entering into computations, when range checking of source operands it not used.

Loongson Multimedia instructions do not indicate overflow or underflow occurrence by generating exceptions.

C.4 Loongson multimedia instructions

The Loongson Multimedia Technology defines 65 instructions (see *Table 63.*). The instructions are grouped into the following functional categories:

- Arithmetic Instructions
- Comparison Instructions
- Conversion Instructions
- Logical Instructions
- Shift Instructions

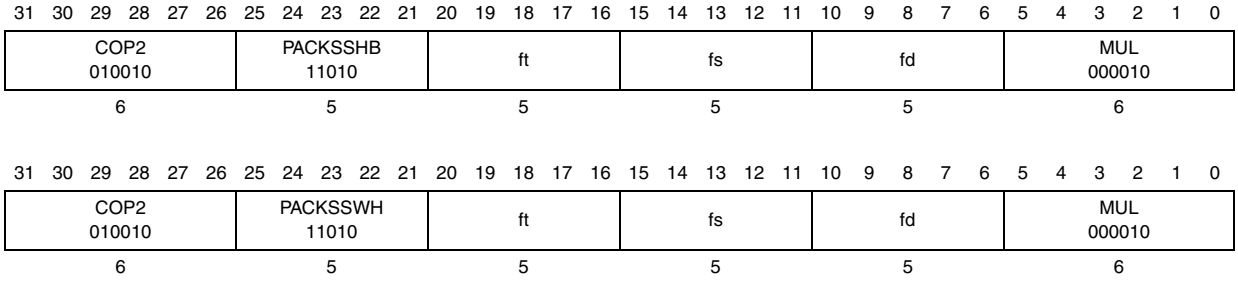
Table 63. Loongson multimedia instruction set summary (opcode = COP2)

FUN Fmt	ADD 000000	SUB 000001	MUL 000010	DIV 000011
24	PADDSH	PSUBSH	PSHUFH	PUNPCKLHW
25	PADDUSH	PSUBUSH	PACKSSWH	PUNPCKHHW
26	PADDH	PSUBH	PACKSSHB	PUNPCKLBH
27	PADDW	PSUBW	PACKUSHB	PUNPCKHBH
28	PADDSB	PSUBSB	Xor	PINSRH_0
29	PADDUSB	PSUBUSB	Nor	PINSRH_1
30	PADDB	PSUBB	And	PINSRH_2
31	PADDD	PSUBD	PANDN	PINSRH_3

Table 64. Loongson multimedia instruction set summary

FUN Fmt	ROUND.L 001000	TRUNC.L 001001	CEIL.L 001010	FLOOR.L 001011
24	PAVGH	PCMPEQW	PSLLW	PSRLW
25	PAVGB	PCMPGTW	PSLLH	PSRLH
26	PMAXSH	PCMPEQH	PMULLH	PSRAW
27	PMINSH	PCMPGTH	PMULHH	PSRAH
28	PMAXUB	PCMPEQB	PMULUW	PUNPCKLWD
29	PMINUB	PCMPGTB	PMULHUH	PUNPCKHWD
24	Addu	Subu	Sll	Srl
25	Or	PASUBUB	Dsll	Dsrl
26	Add	Sub	PEXTRH	Sra
27	Dadd	Dsub	PMADDHW	Dsra
28	Sequ	Sltu	Sleu	BIADD
29	Seq	Slt	Sle	PMOVMASKB

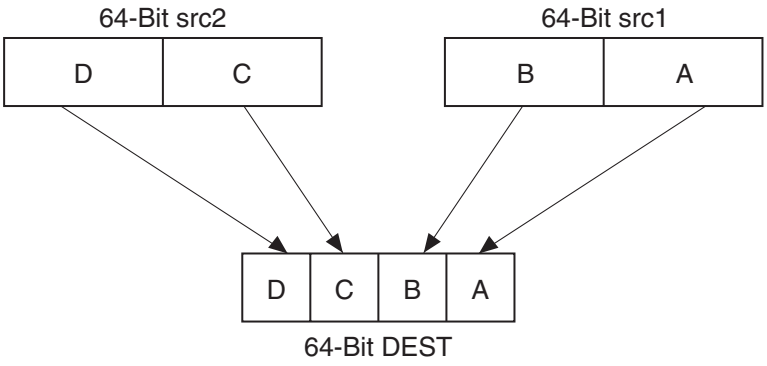
C.5 PACKSSHB/PACKSSWH - pack with signed saturation



Format: PACKSSHB fd,fs,ft
PACKSSWH fd,fs,ft

Description: Converts packed signed halfword integers into packed signed byte integers (PACKSSHB) or converts packed signed word integers into packed signed halfword integers (PACKSSWH), using saturation to handle overflow conditions. See *Figure 24* for an example of the packing operation.

Figure 24. Operation of the PACKSSWH instruction using 64-bit operands



The PACKSSHB instruction converts 4 signed halfword integers from the first operand and 4 signed halfword integers from the second operand into 8 signed byte integers and stores the result in the destination operand. If a signed halfword integer value is beyond the range of a signed byte integer (that is, greater than 7FH for a positive integer or greater than 80H for a negative integer), the saturated signed byte integer value of 7FH or 80H, respectively, is stored in the destination.
The PACKSSWH instruction packs 2 signed words from the first operand and 2 signed words from the second operand into 4 signed half words in the destination operand (see *Figure 24*). If a signed word integer value is beyond the range of a signed halfword (that is, greater than 7FFFH for a positive integer or greater than 8000H for a negative integer), the saturated signed halfword integer value of 7FFFH or 8000H, respectively, is stored into the destination.

Operation:

```
PACKSSHB
fd[7..0]   ← SaturateSignedHalfwordToSignedByte fs[15..0];
fd[15..8]  ← SaturateSignedHalfwordToSignedByte fs[31..16];
fd[23..16] ← SaturateSignedHalfwordToSignedByte fs[47..32];
```

fd[31..24] ← SaturateSignedHalfwordToSignedByte fs[63..48];
fd[39..32] ← SaturateSignedHalfwordToSignedByte ft[15..0];
fd[47..40] ← SaturateSignedHalfwordToSignedByte ft[31..16];
fd[55..48] ← SaturateSignedHalfwordToSignedByte ft[47..32];
fd[63..56] ← SaturateSignedHalfwordToSignedByte ft[63..48];

PACKSSWH
fd[15..0] ← SaturateSignedWordToSignedHalfWord fs[31..0];
fd[31..16] ← SaturateSignedWordToSignedHalfWord fs[63..32];
fd[47..32] ← SaturateSignedWordToSignedHalfWord ft[31..0];
fd[63..48] ← SaturateSignedWordToSignedHalfWord ft[63..32];

Exception: None.

C.6 PACKUSHB - pack with unsigned saturation

31 30 29 28 27 26	25 24 23 22 21	20 19 18 17 16	15 14 13 12 11	10 9 8 7 6	5 4 3 2 1 0
COP2 010010	PACKUSHB 11011	ft	fs	fd	MUL 000010
6	5	5	5	5	6

Format: PACKUSHB fd,fs,ft

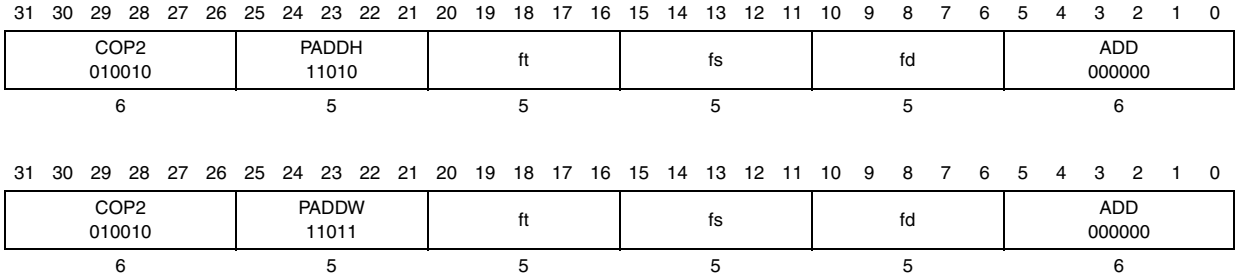
Description: Converts 4 signed halfword integers from the first operand and 4 signed halfword integers from the second operand into 8 unsigned byte integers and stores the result in the destination operand. (See *Figure 24* for an example of the packing operation.) If a signed halfword integer value is beyond the range of an unsigned byte integer (that is, greater than FFH or less than 00H), the saturated unsigned byte integer value of FFH or 00H, respectively, is stored in the destination.
The PACKUSHB instruction operates on 64-bit operands.

Operation:
PACKUSHB
fd[7..0] ← SaturateSignedHalfwordToUnsignedByte fs[15..0];
fd[15..8] ← SaturateSignedHalfwordToUnsignedByte fs [31..16];
fd[23..16] ← SaturateSignedHalfwordToUnsignedByte fs [47..32];
fd[31..24] ← SaturateSignedHalfwordToUnsignedByte fs [63..48];
fd[39..32] ← SaturateSignedHalfwordToUnsignedByte ft[15..0];
fd[47..40] ← SaturateSignedHalfwordToUnsignedByte ft[31..16];
fd[55..48] ← SaturateSignedHalfwordToUnsignedByte ft[47..32];
fd[63..56] ← SaturateSignedHalfwordToUnsignedByte ft[63..48];

Exception: None.

C.7 PADDB/PADDH/PADDW - add packed integers

31 30 29 28 27 26	25 24 23 22 21	20 19 18 17 16	15 14 13 12 11	10 9 8 7 6	5 4 3 2 1 0
COP2 010010	PADDB 11110	ft	fs	fd	ADD 000000
6	5	5	5	5	6



Format:

PADDB	fd,fs,ft
PADDH	fd,fs,ft
PADDW	fd,fs,ft

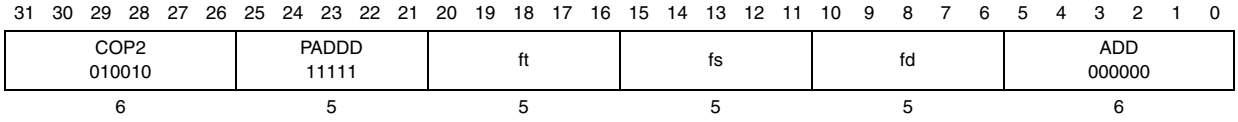
Description: Performs a SIMD add of the packed integers from the first operand and the second operand, and stores the packed integer results in the destination operand. Overflow is handled with wraparound, as described in the following paragraphs.
These instructions operate on 64-bit operands.
The PADDB instruction adds packed byte integers. When an individual result is too large to be represented in 8 bits (overflow), the result is wrapped around and the low 8 bits are written to the destination operand (that is, the carry is ignored).
The PADDH instruction adds packed halfword integers. When an individual result is too large to be represented in 16 bits (overflow), the result is wrapped around and the low 16 bits are written to the destination operand.
The PADDW instruction adds packed word integers. When an individual result is too large to be represented in 32 bits (overflow), the result is wrapped around and the low 32 bits are written to the destination operand.
Note that the PADDB, PADDH, and PADDW instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values operated on.

Operation:

```
PADDB
fd[7..0]    ← fs[7..0] + ft[7..0];
* repeat add operation for 2nd through 7th byte *;
fd[63..56]  ← fs[63..56] + ft[63..56];

PADDH
fd[15..0]   ← fs[15..0] + ft[15..0];
* repeat add operation for 2nd and 3th halfword *;
fd[63..48]  ← fs[63..48] + ft[63..48];

PADDW
fd[31..0]   ← fs[31..0] + ft[31..0];
fd[63..32]  ← fs[63..32] + ft[63..32];
```

Exception: None.

C.8 PADDD - add packed doubleword integers



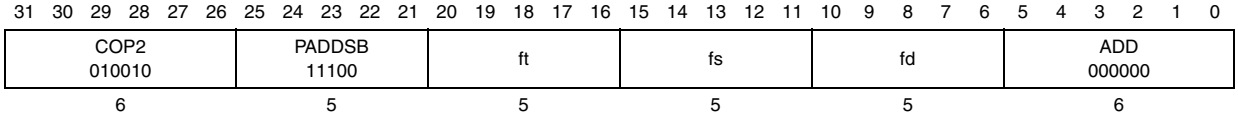
Format: PADDD fd,fs,ft

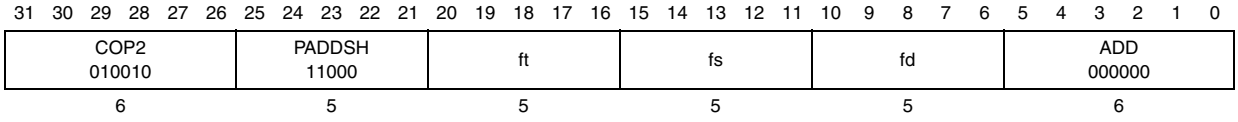
Description: Adds the first operand to the second operand and stores the result in the destination operand. The source operand can be a doubleword integer stored in a 64-bit register. The destination operand can be a doubleword integer stored in a 64-bit register. When a doubleword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).
Note that the PADDD instruction can operate on either unsigned or signed (two's complement notation) integers; however, it does not indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values operated on.

Operation: PADDD
fd[63..0] ← fs[63..0] + ft[63..0];

Exception: None.

C.9 PADDSB/PADDSH - add packed signed integers





Format: PADDSB fd,fs,ft
PADDSH fd,fs,ft

Description: Performs a SIMD add of the packed signed integers from the first operand and the second operand, and stores the packed integer results in the destination operand. Overflow is handled with signed saturation, as described in the following paragraphs. These instructions operate on 64-bit operands.
The PADDSB instruction adds packed signed byte integers. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.
The PADDSH instruction adds packed signed halfword integers. When an individual halfword result is beyond the range of a signed halfword integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

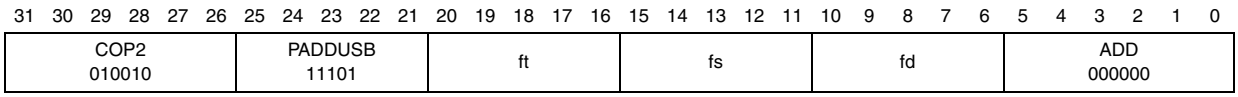
Operation:

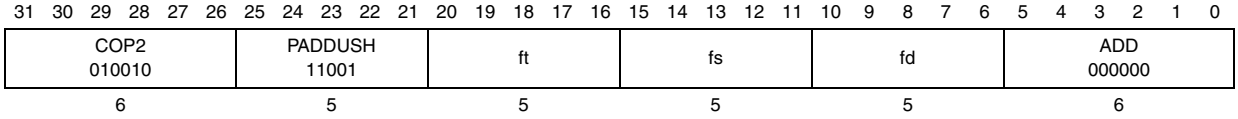
PADDSB
fd[7..0] ← SaturateToSignedByte(fs[7..0] + ft[7..0]);
* repeat add operation for 2nd through 7th bytes *;
fd[63..56] ← SaturateToSignedByte(fs[63..56] + ft[63..56]);

PADDSH
fd[15..0] ← SaturateToSignedHalfword(fs[15..0] + ft[15..0]);
* repeat add operation for 2nd and 7th halfwords *;
fd[63..48] ← SaturateToSignedHalfword(fs[63..48] + ft[63..48]);

Exception: None.

C.10 PADDUSB/PADDUSH - add packed unsigned integers





Format:

PADDUSB fd,fs,ft
PADDUSH fd,fs,ft

Description:

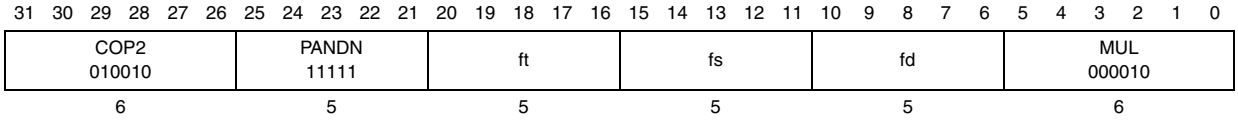
Performs a SIMD add of the packed unsigned integers from the first operand and the second operand, and stores the packed integer results in the destination operand. Overflow is handled with unsigned saturation, as described in the following paragraphs.
These instructions operate on 64-bit operands.
The PADDUSB instruction adds packed unsigned byte integers. When an individual byte result is beyond the range of an unsigned byte integer (that is, greater than FFH), the saturated value of FFH is written to the destination operand.
The PADDUSH instruction adds packed unsigned halfword integers. When an individual halfword result is beyond the range of an unsigned halfword integer (that is, greater than FFFFH), the saturated value of FFFFH is written to the destination operand.

Operation:

PADDUSB
fd[7..0] ← SaturateToUnsignedByte(fs[7..0] + ft[7..0]);
* repeat add operation for 2nd through 7th bytes *;
fd[63..56] ← SaturateToUnsignedByte(fs[63..56] + ft[63..56]);
PADDUSH
fd[15..0] ← SaturateToUnsignedHalfword(fs[15..0] + ft[15..0]);
* repeat add operation for 2nd and 3rd halfwords *;
fd[63..48] ← SaturateToUnsignedHalfword(fs[63..48] + ft[63..48]);

Exception: None.

C.11 PANDN - logical and not



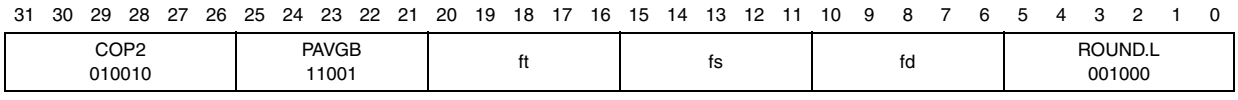
Format: PANDN fd,fs,ft

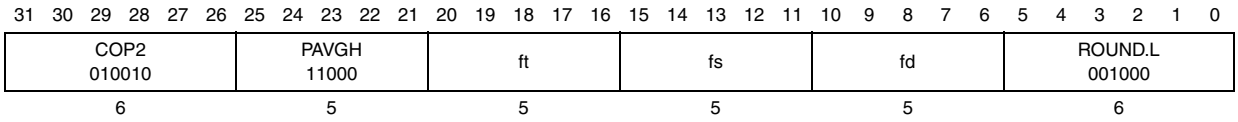
Description: Performs a bitwise logical NOT of the first operand, then performs a bitwise logical AND of the second operand and the inverted destination operand. The result is stored in the destination operand. The source operand can be a 64-bit register. The destination operand can be a 64-bit register. Each bit of the result is set to 1 if the corresponding bit in the first operand is 0 and the corresponding bit in the second operand is 1; otherwise, it is set to 0.

Operation: PANDN
fd ←(NOT fs) AND ft;

Exception: None.

C.12 PAVGB/PAVGH - average packed integers





Format: PAVGB fd,fs,ft
PAVGH fd,fs,ft

Description: Performs a SIMD average of the packed unsigned integers from the first operand and the second operand, and stores the results in the destination operand. For each corresponding pair of data elements in the first and second operands, the elements are added together, a 1 is added to the temporary sum, and that result is shifted right one bit position. The source operand can be a 64-bit register. The destination operand can be a 64-bit register.
The PAVGB instruction operates on packed unsigned bytes and the PAVGH instruction operates on packed unsigned halfwords.

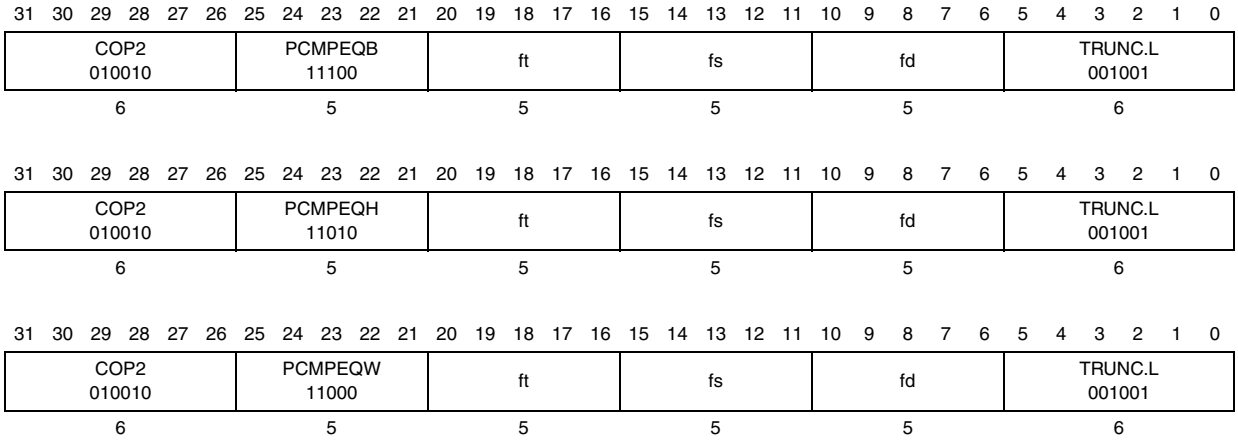
Operation: PAVGB
ft[7-0] ← (fs[7..0] + ft[7..0] + 1) >>; * temp sum before shifting is 9 bits *
* repeat operation performed for bytes 2 through 6 *;
ft[63-56] ← (fs[63..56] + ft[63..56] + 1) >>1;

PAVGH
ft[15-0] ← (fs[15..0] + ft[15..0] + 1) >>1; * temp sum before shifting is 17 bits *

* repeat operation performed for halfwords 2 and 3 *;
ft[63-48] ← (fs[63..48] + ft[63..48] + 1) >>1;

Exception: None.

C.13 PCMPEQB/PCMPEQH/PCMPEQW - compare packed data for equal



Format:

PCMPEQB	fd,fs,ft
PCMPEQH	fd,fs,ft
PCMPEQW	fd,fs,ft

Description: Performs a SIMD compare for equality of the packed bytes, halfwords, or words in the first operand and the second operand. If a pair of data elements is equal, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s. The source operand can be a 64-bit register The destination operand can be a 64-bit register.
The PCMPEQB instruction compares the corresponding bytes in the first and second operands; the PCMPEQH instruction compares the corresponding halfwords in the first and second operands; and the PCMPEQW instruction compares the corresponding words in the first and second operands.

Operation:

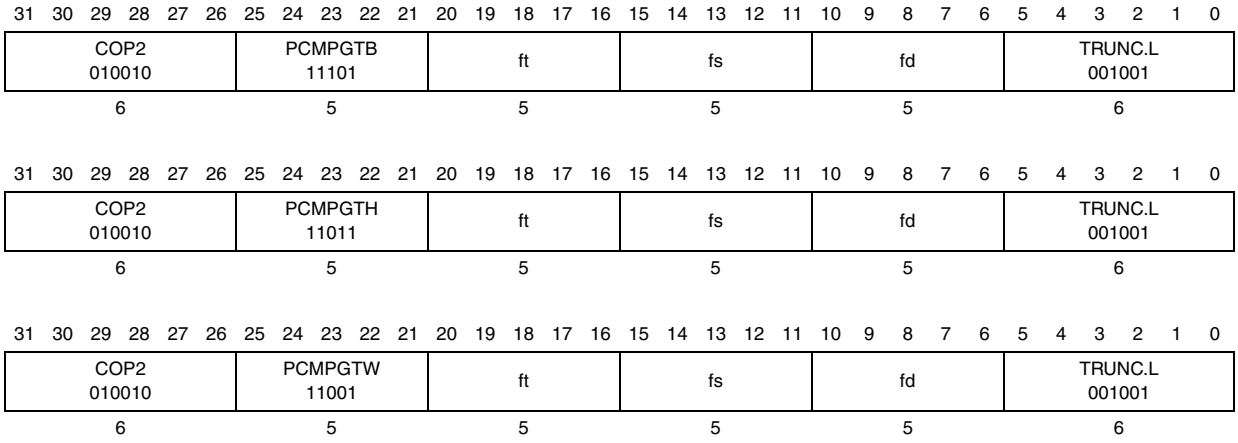
PCMPEQB
IF fs[7..0] =ft[7..0]
THEN fd[7..0] ← FFH;
ELSE fd[7..0] ← 0;
* Continue comparison of 2nd through 7th bytes in fs and ft *
IF fs[63..56] = ft[63..56]
THEN fd[63..56] ← FFH;
ELSE fd[63..56] ← 0;

PCMPEQH
IF fs[15..0] = ft[15..0]
THEN fd[15..0] ← FFFFH;
ELSE fd[15..0] ← 0;
* Continue comparison of 2nd and 3rd halfwords in fs and ft *
IF fs[63..48] = ft[63..48]

```
THEN fd[63..48]   ← FFFFH;
ELSE fd[63..48]   ← 0;

PCMPEQW
IF fs[31..0] = ft[31..0]
THEN fd[31..0]    ← FFFFFFFFH;
ELSE fd[31..0]    ← 0;
IF fs[63..32] = ft[63..32]
THEN fd[63..32]   ← FFFFFFFFH;
ELSE fd[63..32]   ← 0;
```

Exception: None.

C.14 PCMPGTB/PCMPGTH/PCMPGTW - compare packed signed integers



Format:

```
PCMPGTB    fd,fs,ft
PCMPGTH    fd,fs,ft
PCMPGTW    fd,fs,ft
```

Description: Performs a SIMD signed compare for the greater value of the packed byte, halfword, or word integers in the first operand and the second operand. If a data element in the first operand is greater than the corresponding date element in the second operand, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s. The source operand can be a 64-bit register. The destination operand can be a 64-bit register.

The PCMPGTB instruction compares the corresponding signed byte integers in the first and second operands; the PCMPGTH instruction compares the corresponding signed halfword integers in the first and second operands; and the PCMPGTW instruction compares the corresponding signed word integers in the first and second operands.

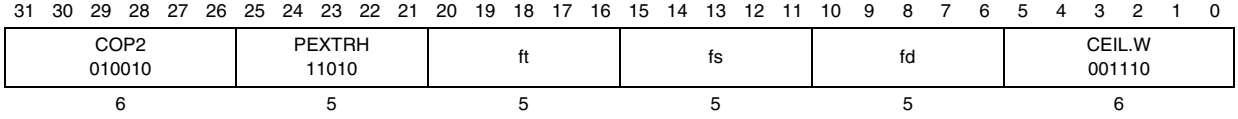
Operation:

```
PCMPGTB
IF fs[7..0] > ft[7..0]
THEN fd[7 0]  ← FFH;
ELSE fd[7..0] ← 0;
* Continue comparison of 2nd through 7th bytes in fs and ft *
```

```
IF fs[63..56] > ft[63..56]
THEN fd[63..56]   ← FFH;
ELSE fd[63..56]   ← 0;

PCMPGTH
IF fs[15..0] > ft[15..0]
THEN fd[15..0]    ← FFFFH;
ELSE fd[15..0]    ← 0;
* Continue comparison of 2nd and 3rd halfwords in fs and ft *
IF fs[63..48] > ft[63..48]
THEN fd[63..48]   ← FFFFH;
ELSE fd[63..48]   ← 0;

PCMPGTW
IF fs[31..0] > ft[31..0]
THEN fd[31..0]    ← FFFFFFFFH;
ELSE fd[31..0]    ← 0;
IF fs[63..32] > ft[63..32]
THEN fd[63..32]   ← FFFFFFFFH;
ELSE fd[63..32]   ← 0;
```

Exception: None.

C.15 PEXTRH - extract halfword



Format: PEXTRH fd,fs,ft

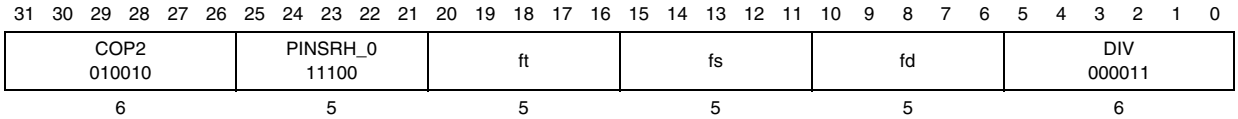
Description: Copies the halfword in the first operand specified by the second operand to the destination operand. The high halfword of the destination operand is cleared (set to all 0s).

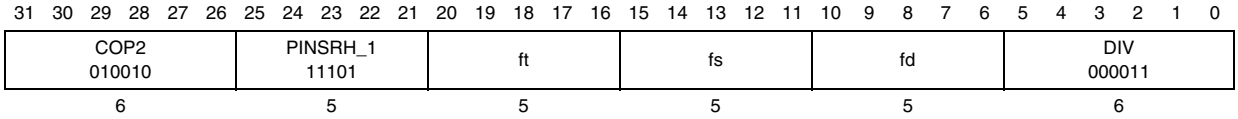
Operation:

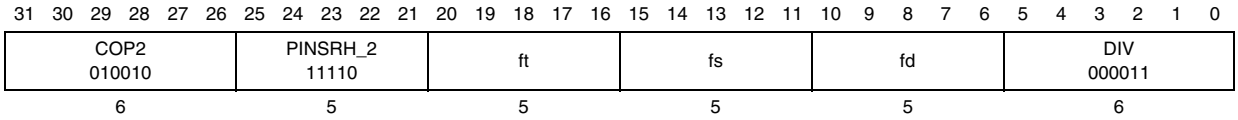
```
PEXTRH
SEL    ← ft AND 3H;
TEMP   ← (fs >> (SEL * 16)) AND FFFFH;
fd[15..0]    ← TEMP[15..0];
fd[63..16]   ← 00000000H;
```

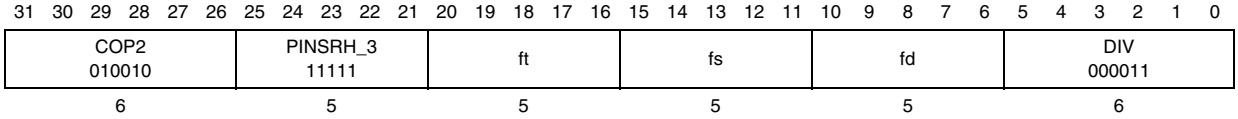
Exception: None.

C.16 PINSRH - insert halfword









Format:

PINSRH_0	fd,fs,ft
PINSRH_1	fd,fs,ft
PINSRH_2	fd,fs,ft
PINSRH_3	fd,fs,ft

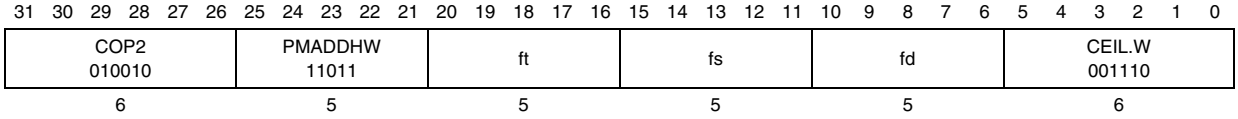
Description: Copies a halfword from the second operand and inserts it in the first operand at the location specified with the number of the instruction name. (The other halfwords in the first register are left untouched.)

Operation:

```
PINSRH_0
MASK ← 000000000000FFFFH;
fd   ← (fs AND NOT MASK) OR (((ft << (0 *16)) AND MASK);

PINSRH_1
MASK ← 00000000FFFF0000H;
fd   ← (fs AND NOT MASK) OR (((ft <<(1 *16)) AND MASK);

PINSRH_2
MASK ← 0000FFFF00000000H;
fd   ← (fs AND NOT MASK) OR (((ft <<(2 *16)) AND MASK);

PINSRH_3
MASK ← FFFF000000000000H;
fd   ← (fs AND NOT MASK) OR (((ft <<(3 *16)) AND MASK);
```

Exception: None.

C.17 PMADDHW - multiply and add packed integers

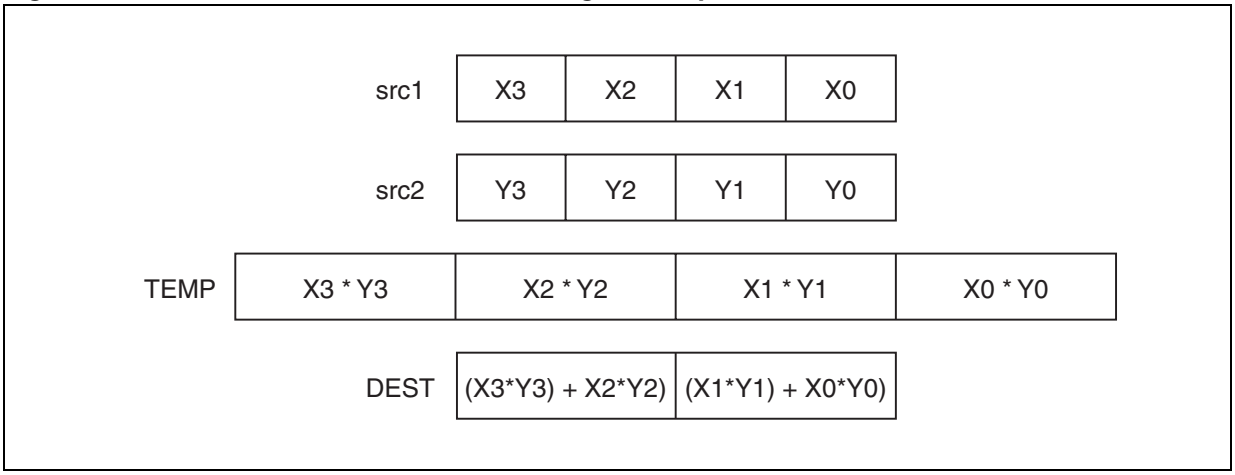


Format: PMADDHW fd,fs,ft

Description: Description:Multiplies the individual signed halfwords of the first operand by the corresponding signed halfwords of the second operand, producing temporary signed,

and word results. The adjacent word results are then summed and stored in the destination operand. For example, the corresponding low-order halfwords (15-0) and (31-16) in the first and second operands are multiplied by one another and the word results are added together and stored in the low word of the destination register (31-0). The same operation is performed on the other pairs of adjacent halfwords. (*Figure 25* shows this operation when using 64-bit operands.) The source operands can be a 64-bit register. The destination operand can be a 64-bit register.
The PMADDHW instruction wraps around only in one situation: when the 2 pairs of halfwords being operated on in a group are all 8000H. In this case, the result wraps around to 80000000H.

Figure 25. PMADDHW Execution model using 64-bit operands



Operation:
PMADDHW
fd[31..0] ← (fs[15..0] * ft[15..0]) + (fs[31..16] * ft[31..16]);
fd[63..32] ← (fs[47..32] * ft[47..32]) + (fs[63..48] * ft[63..48]);

Exception: None.

C.18 PMAXSH - maximum of packed signed halfword integers

31–26	25–21	20–16	15–11	10–6	5–0
COP2 010010	PMAXSH 11010	ft	fs	fd	ROUND.L 00100
6	5	5	5	5	6

Format: PMAXSH fd,fs,ft

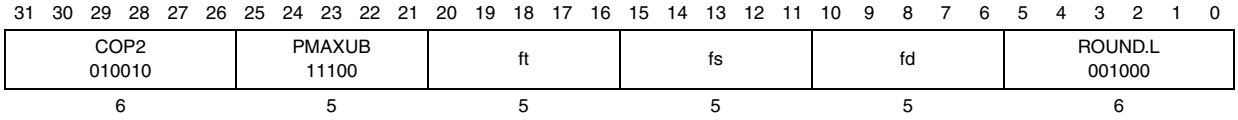
Description: Performs a SIMD compare of the packed signed halfword integers in the first operand and the second operand, and returns the maximum value for each pair of halfword integers to the destination operand. The source operands can be a 64-bi register. The destination operand can be a 64-bi register.

Operation:
PMAXSH
IF (fs[15..0] > ft[15..0]) THEN
fd[15..0] ← fs[15..0];
ELSE
fd[15..0] ← ft[15..0];

```
FI
* repeat operation for 2nd and 3rd halfwords in first and second operands *
IF (fs[63..48] >ft[63..48]) THEN
fd[63..48]  ← fs[63..48];
ELSE
fd[63..48]  ← ft[63..48];
FI
```

Exception: None.

C.19 PMAXUB - maximum of packed unsigned byte integers



Format: PMAXUB fd,fs,ft

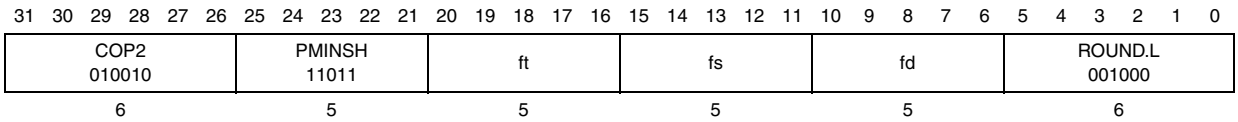
Description: Performs a SIMD compare of the packed unsigned byte integers in the first operand and the second operand, and returns the maximum value for each pair of byte integers to the destination operand. The source operands can be a 64-bit register. The destination operand can be a 64-bit register.

Operation:

```
PMAXUB
IF (fs[7..0] > ft[7..0]) THEN
fd[7..0]    ← fs[7..0];
ELSE
fd[7..0]    ← ft[7..0];
FI
* repeat operation for 2nd through 7th bytes in first and second operands *
IF (fs[63..56] > ft[63..56]) THEN
fd[63..56]  ← fs[63..56];
ELSE
fd[63..56]  ← ft[63..56];
FI
```

Exception: None.

C.20 PMINSH - minimum of packed signed halfword integers



Format: PMINSH fd,fs,ft

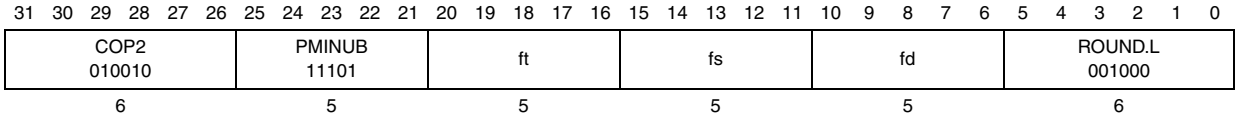
Description: Performs a SIMD compare of the packed signed halfword integers in the first operand and the second operand, and returns the minimum value for each pair of halfword integers to the destination operand. The source operands can be a 64-bit register. The destination operand can be a 64-bit register.

Operation:

```
Operation:PMINSH
IF (fs[15..0] < ft[15..0]) THEN
fd[15..0]    ← fs[15..0];
ELSE
fd[15..0]    ← ft[15..0];
FI
* repeat operation for 2nd and 3rd halfwords in first and second operands *
IF (fs[63..48] < ft[63..48]) THEN
fd[63..48]   ← fs[63..48];
ELSE
fd[63..48]   ← ft[63..48];
FI
```

Exception: None.

C.21 PMINUB - minimum of packed unsigned byte integers



Format: PMINUB fd,fs,ft

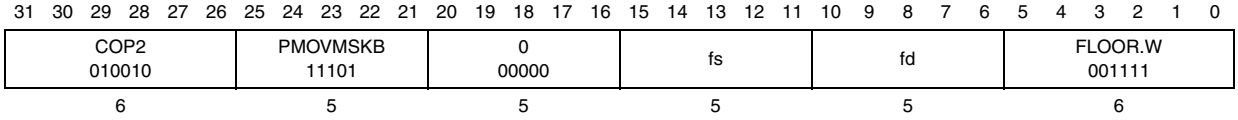
Description: Performs a SIMD compare of the packed unsigned byte integers in the first operand and the second operand, and returns the minimum value for each pair of byte integers to the destination operand. The source operands can be a 64-bit register. The destination operand can be a 64-bit register.

Operation:

```
PMINUB
IF (fs[7..0] < ft[7..0]) THEN
fd[7..0]     ← fs[7..0];
ELSE
fd[7..0]     ← ft[7..0];
FI
* repeat operation for 2nd through 7th bytes in first and second operands *
IF (fs[63..56] < ft [63..56]) THEN
fd[63..56]   ← fs[63..56];
ELSE
fd[63..56]   ← ft[63..56];
FI
```

Exception: None.

C.22 PMOVMSKB - move byte mask



Format: PMOVMSKB fd,fs

Description: Creates a mask made up of the most significant bit of each byte of the first operand and stores the result in the low byte of the destination operand. The source operand is a 64-bit register. When operating on 64-bit operands, the byte mask is 8 bits.

Operation:
```
PMOVMSKB
fd[0]    ← fs[7];
fd[1]    ← fs[15];
* repeat operation for bytes 2 through 6 *
fd[7]    ← fs[63];
fd[63..8] ← 00000000000000H;
```

Exception: None.

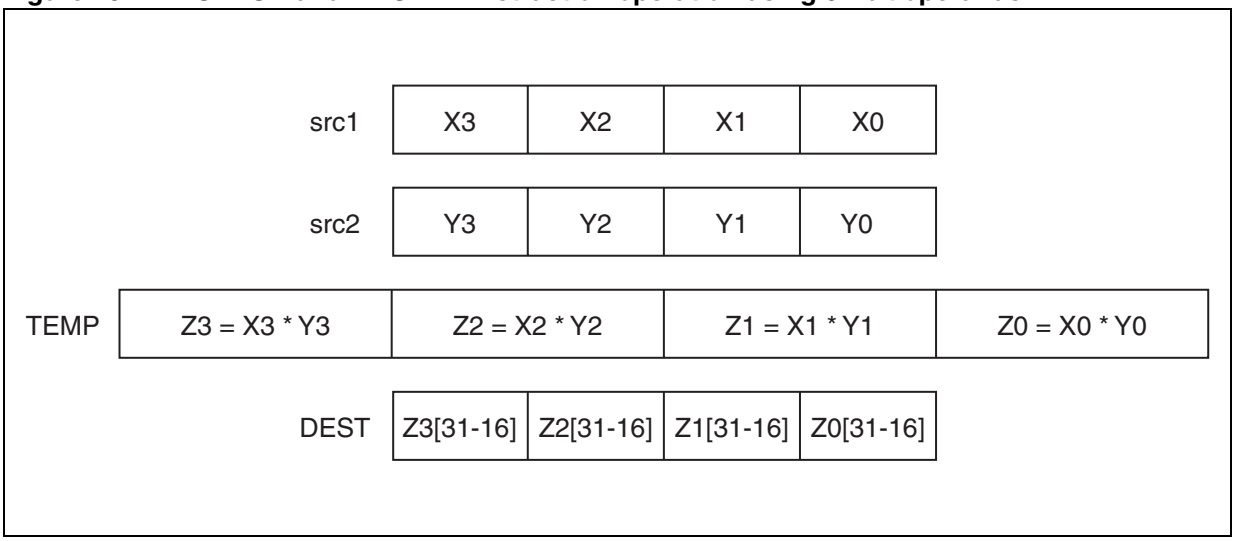
C.23 PMULHUH - multiply packed unsigned integers and store high result

31–26	25–21	20–16	15–11	10–6	5–0
COP2 010010	PMULHUH 11101	ft	fs	fd	CEIL.L 001010
6	5	5	5	5	6

Format: PMULHUH fd,fs,ft

Performs a SIMD unsigned multiply of the packed unsigned halfword integers in the first operand and the second operand, and stores the high 16 bits of each 32-bit intermediate results in the destination operand. (*Figure 26* shows this operation when using 64-bit operands.) The source operands can be a 64-bit register. The destination operand can be a 64-bit register.

Figure 26. PMULHUH and PMULHH instruction operation using 64-bit operands

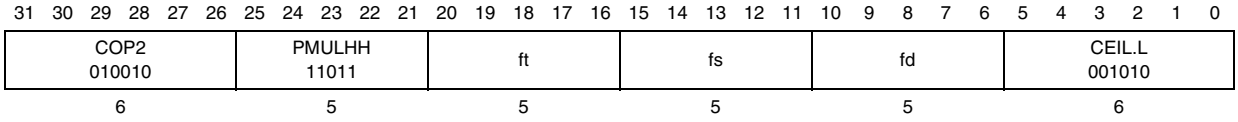


Operation:
```
PMULHUH
TEMP0[31..0] ← fs[15..0] * ft[15..0]; * Unsigned multiplication *
TEMP1[31..0] ← fs[31..16] * ft[31..16];
TEMP2[31..0] ← fs[47..32] * ft[47..32];
TEMP3[31..0] ← fs[63..48] * ft[63..48];
```

```
fd[15..0]   ← TEMP0[31..16];
fd[31..16]  ← TEMP1[31..16];
fd[47..32]  ← TEMP2[31..16];
fd[63..48]  ← TEMP3[31..16];
```

Exception: None.

C.24 PMULHH - multiply packed signed integers and store high result



Format: PMULHH fd,fs,ft

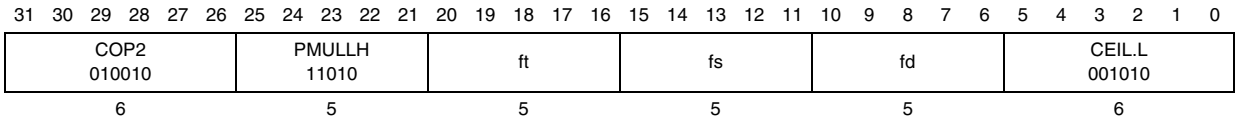
Description: Performs a SIMD signed multiply of the packed signed halfword integers in the first operand and the second operand, and stores the high 16 bits of each intermediate 32-bit result in the destination operand. (*Figure 26* shows this operation when using 64-bit operands.) The source operands can be a 64-bit register. The destination operand can be a 64-bit register.

Operation:

```
PMULHH
TEMP0[31..0] ← fs[15..0] * ft[15..0]; * Signed multiplication *
TEMP1[31..0] ← fs[31..16] * ft[31..16];
TEMP2[31..0] ← fs[47..32] * ft[47..32];
TEMP3[31..0] ← fs[63..48] * ft[63..48];
fd[15..0]   ← TEMP0[31..16];
fd[31..16]  ← TEMP1[31..16];
fd[47..32]  ← TEMP2[31..16];
fd[63..48]  ← TEMP3[31..16];
```

Exception: None

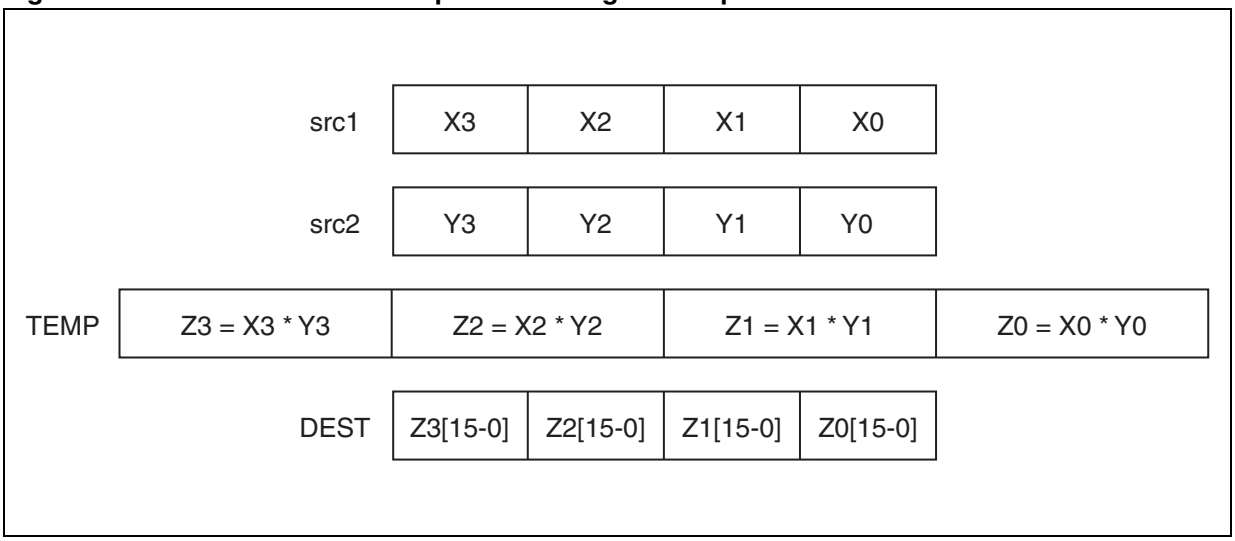
C.25 PMULLH - multiply packed signed integers and store low result



Format: PMULLH fd,fs,ft

Performs a SIMD signed multiply of the packed signed halfword integers in the first operand and the second operand, and stores the low 16 bits of each intermediate 32-bit result in the destination operand. (*Figure 26* shows this operation when using 64-bit operands.) The source operand can be a 64-bit register. The destination operand can be a 64-bit register.

Figure 27. PMULLH instruction operation using 64-bit operands



Operation:

```
PMULLH
TEMP0[31..0] ← fs[15..0] * ft[15..0]; * Signed multiplication *
TEMP1[31..0] ← fs[31..16] * ft[31..16];
TEMP2[31..0] ← fs[47..32] * ft[47..32];
TEMP3[31..0] ← fs[63..48] * ft[63..48];
fd[15..0]    ← TEMP0[15..0];
fd[31..16]   ← TEMP1[15..0];
fd[47..32]   ← TEMP2[15..0];
fd[63..48]   ← TEMP3[15..0];
```

Exception: None.

C.26 PMULUW - multiply packed unsignedword integers

31–26	25–21	20–16	15–11	10–6	5–0
COP2 010010	PMULUW 11100	ft	fs	fd	CEIL.L 001010
6	5	5	5	5	6

Format: PMULUW fd,fs,ft

Description: Multiplies the first operand by the second operand and stores the result in the destination operand. The source operands can be an unsigned word integer stored in the low word of a 64-bit register. The result is an unsigned doubleword integer stored in the destination a 64-bit register. When a doubleword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

Operation:

```
PMULUW
fd[63..0]    ← fs[31..0] * ft[31..0];
```

Exception: None.

C.27 PSADBH - compute sum of absolute differences

31–26	25–21	20–16	15–11	10–6	5–0
COP2 010010	PASUBUB 11001	ft	fs	fd	TRUNC.W 001101
6	5	5	5	5	6

31–26	25–21	20–16	15–11	10–6	5–0
COP2 010010	BIADD 11100	0 00000	fs	fd	FLOOR.W 001111
6	5	5	5	5	6

Format: PASUBUB fd,fs,ft
BIADD fd,fs

Description: PSADBH instruction computes the absolute value of the difference of 8 unsigned byte integers from the first operand and from the second operand. These 8 differences are then summed to produce an unsigned halfword integer result that is stored in the destination operand. The source operand can be a 64-bit register. The destination operand can be a 64-bit register. *Figure 28* shows the operation of the PSADBH instruction when using 64-bit operands. When operating on 64-bit operands, the halfword integer result is stored in the low halfword of the destination operand, and the remaining bytes in the destination operand are cleared to all 0s.

Figure 28. PSADBH instruction operation using 64-bit operands

src1	X7	X6	X5	X4	X3	X2	X1	X0
src2	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
TEMP	ABS(X7-Y7)	ABS(X6-Y6)	ABS(X5-Y5)	ABS(X4-Y4)	ABS(X3-Y3)	ABS(X2-Y2)	ABS(X1-Y1)	ABS(X0-Y0)
DEST	00H	00H	00H	00H	00H	00H	SUM(TEMP7...TEMP0)	

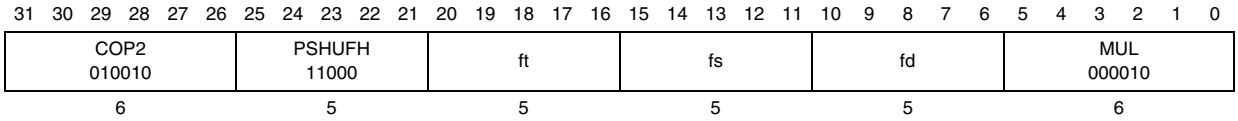
Note: PSADBH instruction is divided into two instructions, PASUBUB and BIADD. PASUBUB instruction computes the absolute value of the difference of 8 unsigned byte integers from the first operand and from the second operand. BIADD computes the sum of 8 unsigned byte integers of the source operand.

Operation:

```
PASUBUB
fd[7..0]    ← ABS(fs[7..0] - ft[7..0]);
* repeat operation for bytes 2 through 6 *
fd[63..56]  ← ABS(fs[63..56] - ft[63..56]);

BIADD
fd[15..0]   ← SUM(fs[7..0]... fs[63..56]);
fd[63..16]  ← 000000000000H;
```

Exception: None.

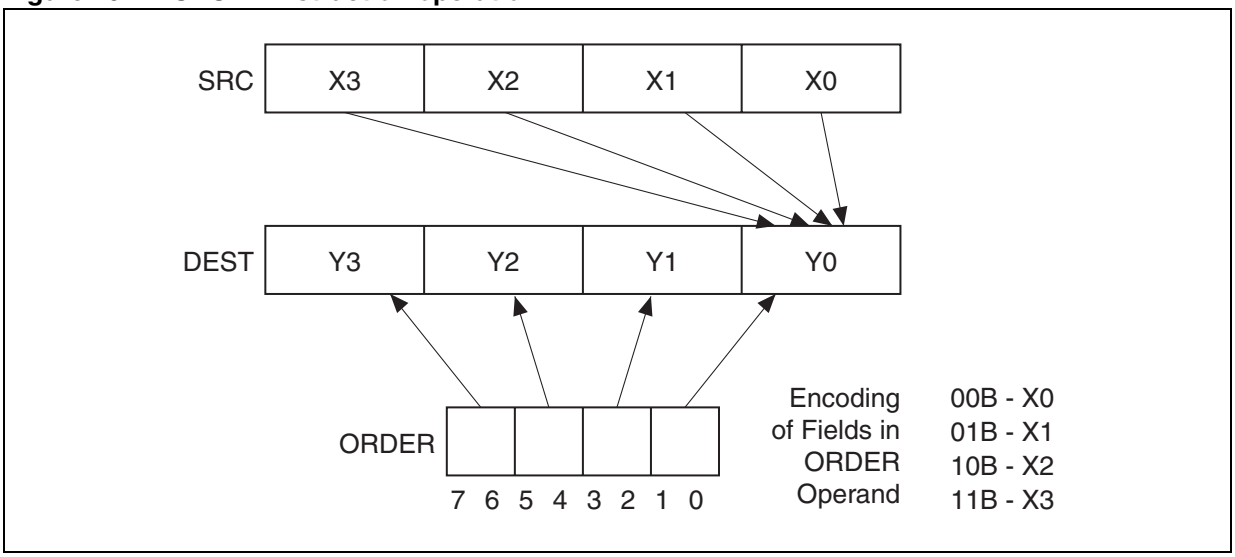
C.28 PSHUFH - shuffle packed halfwords



Format: PSHUFH fd,fs,ft

Description: Copies halfwords from the first operand and inserts them in the destination operand at halfword locations selected with the second operand(order operand). This operation is illustrated in *Figure 29*. For the PSHUFH instruction, each 2-bit field in the second operand selects the contents of one halfword location in the destination operand. The encodings of the second operand fields select halfwords from the first operand to be copied to the destination operand.
The first operand can be a 64-bit register. The destination operand is a 64-bit register. The order operand is a 64-bit register.
Note that this instruction permits a halfword in the first operand to be copied to more than one halfword location in the destination operand.

Figure 29. PSHUFH Instruction operation

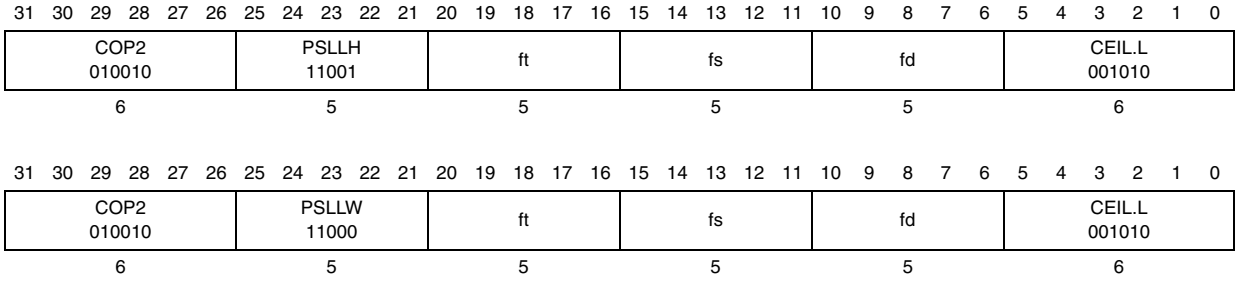


Operation: PSHUFH

```
fd[15..0]   ← (fs >> (ft[1..0] * 16) )[15..0]
fd[31..16]  ← (fs >> (ft[3..2] * 16) )[15..0]
fd[47..32]  ← (fs >> (ft[5..4] * 16) )[15..0]
fd[63..48]  ← (fs >> (ft[7..6] * 16) )[15..0]
```

Exception: None.

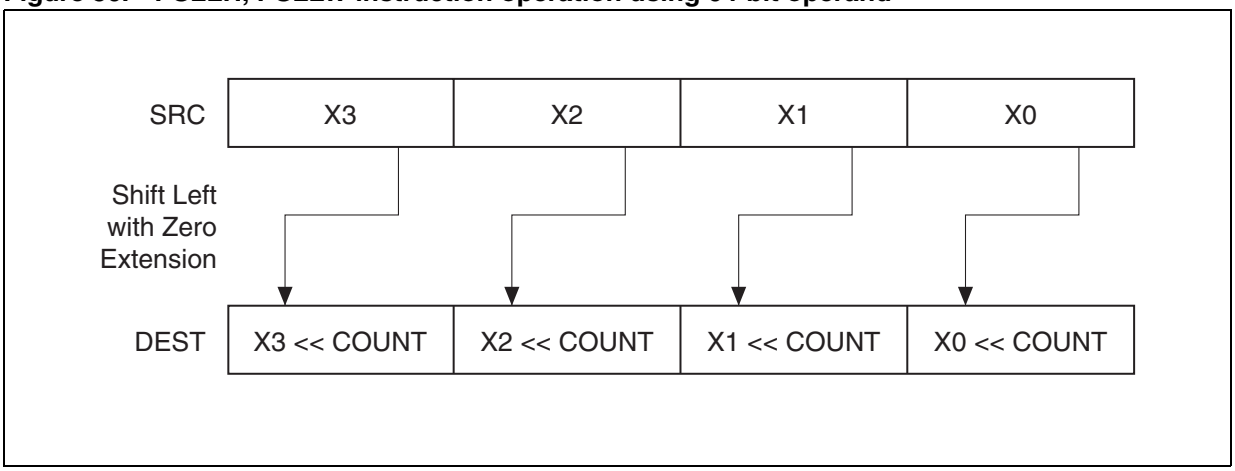
C.29 PSLLH/PSLLW - shift packed data left logical



Format: PSLLH fd,fs,ft
PSLLW fd,fs,ft

Description: Shifts the bits in the individual data elements (halfwords, words) in the first operand to the left by the number of bits specified in the second operand (count operand). As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for halfwords), 31 (for words), then the destination operand is set to all 0s. (*Figure 30* gives an example of shifting words in a 64-bit operand.).

Figure 30. PSLLH, PSLLW instruction operation using 64-bit operand



The PSLLH instruction shifts each of the halfwords in the first operand to the left by the number of bits specified in the count operand; the PSLLW instruction shifts each of the words in the first operand.

Operation:

```
PSLLH
IF (ft[6..0] > 15)
THEN
fd[64..0]    ← 0000000000000000H
ELSE
fd[15..0]    ← ZeroExtend(fs[15..0] << ft[6..0]);
* repeat shift operation for 2nd and 3rd words *;
fd[63..48]   ← ZeroExtend(fs[63..48] << ft[6..0]);
FI;

PSLLW
```

```
IF (ft[6..0] > 31)
THEN
fd[64..0]    ← 0000000000000000H
ELSE
fd[31..0]    ← ZeroExtend(fs[31..0] << ft[6..0]);
fd[63..32]   ← ZeroExtend(fs[63..32] << ft[6..0]);
FI;
```

Exception: None.

C.30 PSRAH/PSRAW - shift packed data right arithmetic

31–26	25–21	20–16	15–11	10–6	5–0
COP2 010010	PSRAH 11011	ft	fs	fd	FLOOR.L 001011
6	5	5	5	5	6

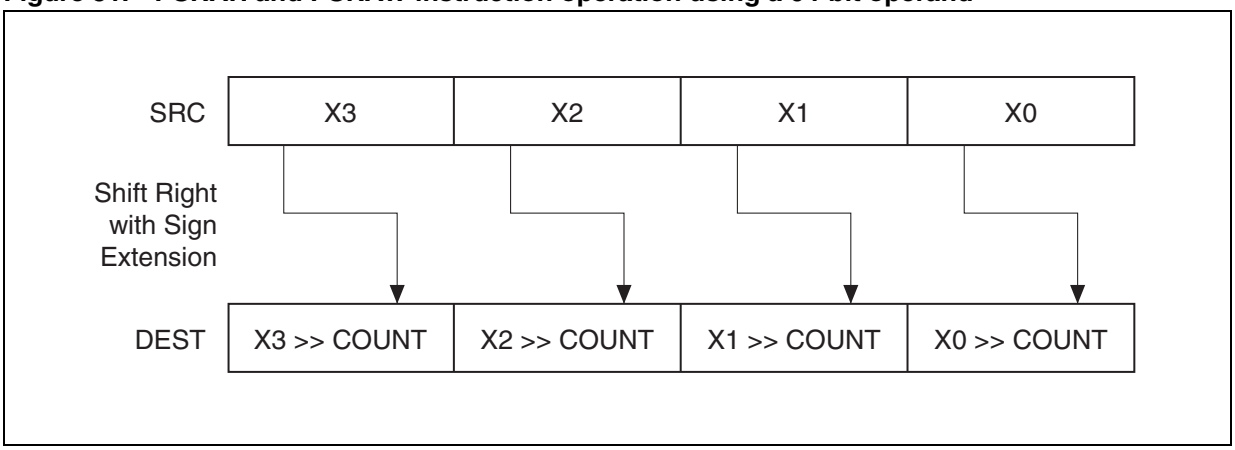
31–26	25–21	20–16	15–11	10–6	5–0
COP2 010010	PSRAW 11010	ft	fs	fd	FLOOR.L 001011
6	5	5	5	5	6

Format:

PSRAH	fd,fs,ft
PSRAW	fd,fs,ft

Description: Shifts the bits in the individual data elements (halfwords or words) in the first operand to the right by the number of bits specified in the second operand (count operand). As the bits in the data elements are shifted right, the empty high-order bits are filled with the initial value of the sign bit of the data element. If the value specified by the count operand is greater than 15 (for halfwords) or 31 (for words), each destination data element is filled with the initial value of the sign bit of the element. (*Figure 31* gives an example of shifting halfwords in a 64-bit operand.)

Figure 31. PSRAH and PSRAW instruction operation using a 64-bit operand

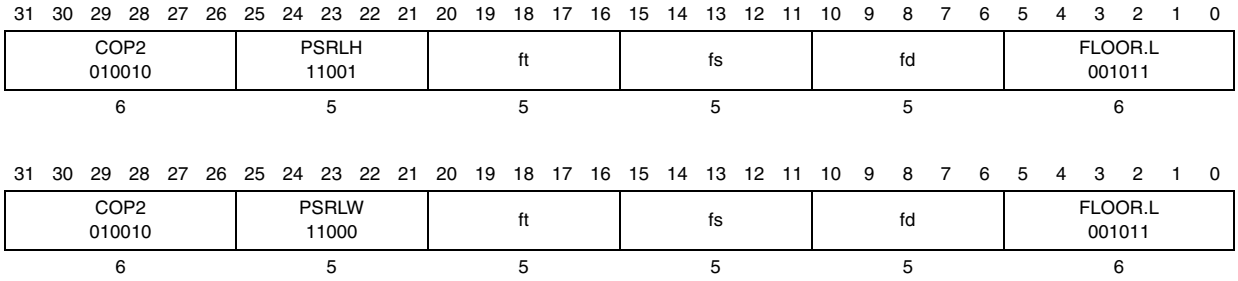


The PSRAH instruction shifts each of the halfwords in the first operand to the right by the number of bits specified in the count operand, and the PSRAW instruction shifts each of the words in the first operand.

Operation:

```
PSRAH
IF (ft[6..0] > 15)
THEN ft[6..0] ← 16;
FI;
fd[15..0] ← SignExtend(fs[15..0] >> ft[6..0]);
* repeat shift operation for 2nd and 3rd halfwords *;
fd[63..48] ← SignExtend(fs[63..48] >> ft[6..0]);

PSRAW
IF (ft[6..0] > 31)
THEN ft[6..0] ←32;
FI;
fd[31..0] ← SignExtend(fs[31..0] >> ft[6..0]);
fd[63..32] ← SignExtend(fs[63..32] >>ft[6..0]);
None.
```

C.31 PSRLH/PSRLW - shift packed data right logical

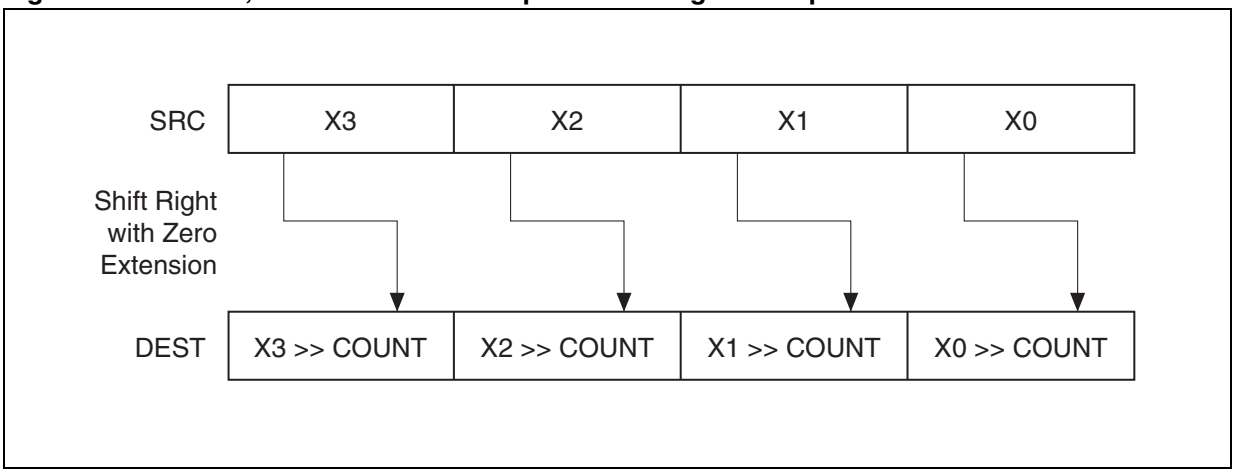


Format:

PSRLH	fd,fs,ft
PSRLW	fd,fs,ft

Description: Shifts the bits in the individual data elements (halfwords, words) in the first operand to the right by the number of bits specified in the second operand (count operand). As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for halfwords), 31 (for words), then the destination operand is set to all 0s. (*Figure 32* gives an example of shifting halfwords in a 64-bit operand.)

Figure 32. PSRLH, PSRLW instruction operation using 64-bit operand



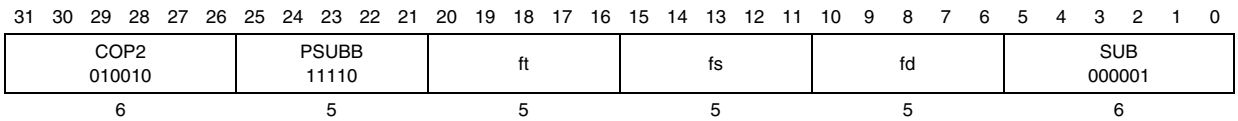
The PSRLH instruction shifts each of the halfwords in the first operand to the right by the number of bits specified in the count operand; the PSRLW instruction shifts each of the words in the first operand.

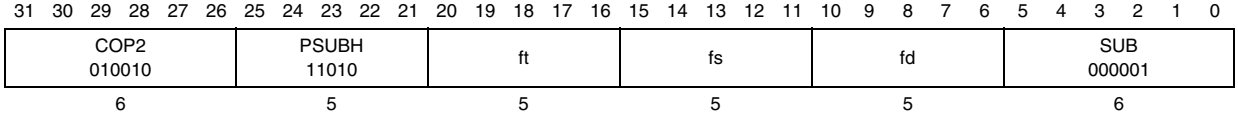
Operation:

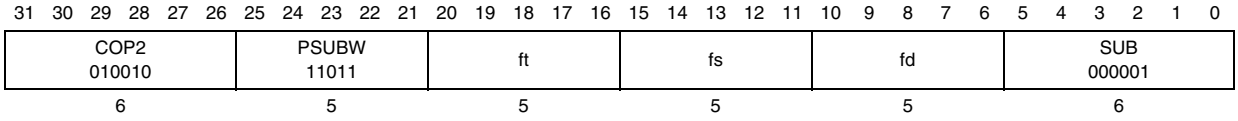
```
PSRLH
IF (ft[6..0] > 15)
THEN
fd[64..0]    ← 0000000000000000H
ELSE
fd[15..0]    ← ZeroExtend(fs[15..0] >> ft[6..0]);
* repeat shift operation for 2nd and 3rd halfwords *;
fd[63..48]   ← ZeroExtend(fs[63..48] >> ft[6..0]);
FI;

PSRLW
IF (COUNT > 31)
THEN
fd[64..0]    ← 0000000000000000H
ELSE
fd[31..0]    ← ZeroExtend(fs[31..0] >> ft[6..0]);
fd[63..32]   ← ZeroExtend(fs[63..32] zz ft[6..0]);
FI;
```

Exception: None.

C.32 PSUBB/PSUBH/PSUBW - subtract packed integers







Format:

PSUBB	fd,fs,ft
PSUBH	fd,fs,ft
PSUBW	fd,fs,ft

Description: Performs a SIMD subtract of the packed integers of the second operand from the packed integers of the first operand, and stores the packed integer results in the destination operand. Overflow is handled with wraparound, as described in the following paragraphs. These instructions operate on 64-bit operands.

The PSUBB instruction subtracts packed byte integers. When an individual result is too large or too small to be represented in a byte, the result is wrapped around and the low 8 bits are written to the destination element.

The PSUBH instruction subtracts packed halfword integers. When an individual result is too large or too small to be represented in a halfword, the result is wrapped around and the low 16 bits are written to the destination element.

The PSUBW instruction subtracts packed word integers. When an individual result is too large or too small to be represented in a word, the result is wrapped around and the low 32 bits are written to the destination element.

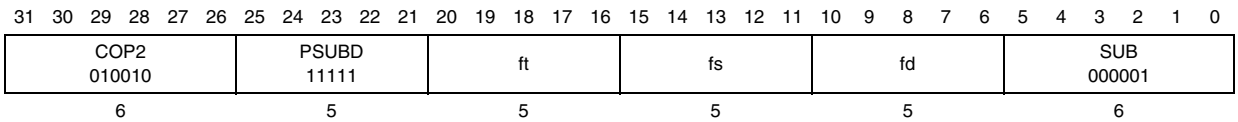
Note that the PSUBB, PSUBW, and PSUBD instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values operated on.

Operation:

```
PSUBB
fd[7..0]    ← fs[7..0] - ft[7..0];
* repeat subtract operation for 2nd through 7th byte *;
fd[63..56]  ← fs[63..56] - ft[63..56];

PSUBH
fd[15..0]   ← fs[15..0] - ft[15..0];
* repeat subtract operation for 2nd and 3rd halfword *;
fd[63..48]  ← fs[63..48] - ft[63..48];

PSUBW
fd[31..0]   ← fs[31..0] - ft[31..0];
fd[63..32]  ← fs[63..32] - ft[63..32];
```

Exception: None.

C.33 PSUBD - subtract packed doubleword integers



Format:

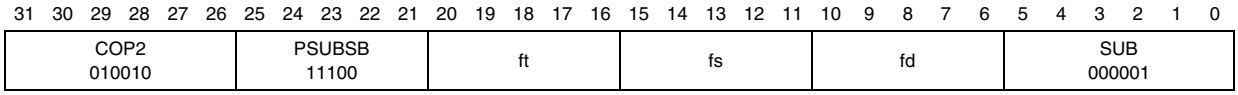
PSUBD	fd,fs,ft

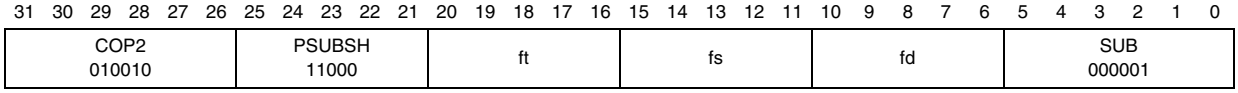
Description: Subtracts the second operand from the first operand and stores the result in the destination operand. When packed doubleword operands are used, a SIMD subtract is performed. When a doubleword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).
Note that the PSUBD instruction can operate on either unsigned or signed (two's complement notation) integers; however, it does not indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values operated on.

Operation:
```
PSUBD
fd[63..0]  ← fs[63..0] - ft[63..0];
```

Exception: None.

C.34 PSUBSB/PSUBSH - subtract packed signed integers





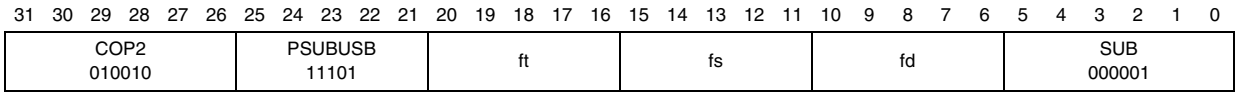
Format:
```
PSUBSB    fd,fs,ft
PSUBSH    fd,fs,ft
```

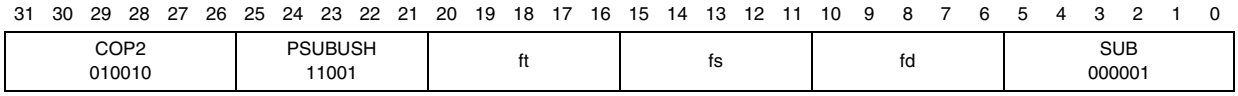
Description: Performs a SIMD subtract of the packed signed integers of the second operand from the packed signed integers of the first operand, and stores the packed integer results in the destination operand. Overflow is handled with signed saturation, as described in the following paragraphs. These instructions operate on 64-bit.
The PSUBSB instruction subtracts packed signed byte integers. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.
The PSUBSH instruction subtracts packed signed halfword integers. When an individual halfword result is beyond the range of a signed halfword integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

Operation:
```
PSUBSB
fd[7..0]   ← SaturateToSignedByte(fs[7..0] - ft[7..0]) ;
* repeat subtract operation for 2nd through 7th bytes *;
fd[63..56] ← SaturateToSignedByte(fs[63..56] - ft[63..56] );

PSUBSH
fd[15..0]  ← SaturateToSignedHalfword(fs[15..0] - ft[15..0] );
* repeat subtract operation for 2nd and 7th halfwords *;
fd[63..48] ← SaturateToSignedHalfword(fs[63..48] - ft[63..48] );
```

Exception: None.

C.35 PSUBUSB/PSUBUSH - subtract packed unsigned integers





Format: PSUBUSB fd,fs,ft
PSUBUSH fd,fs,ft

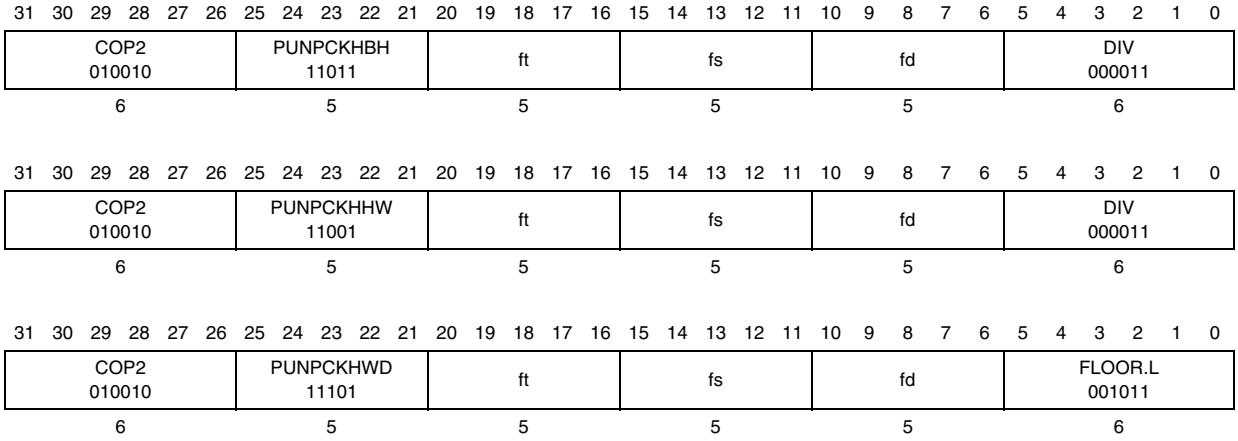
Description: Performs a SIMD subtract of the packed unsigned integers of thesecond operand from the packed unsigned integers of the first operand, and stores the packed unsigned integer results in the destination operand. Overflow is handled with unsigned saturation, as described in the following paragraphs. These instructions operate on 64-bit operands.
The PSUBUSB instruction subtracts packed unsigned byte integers. When an individual byte result is less than zero, the saturated value of 00H is written to the destination operand.
The PSUBUSH instruction subtracts packed unsigned halfword integers. When an individual halfword result is less than zero, the saturated value of 0000H is written to the destination operand.

Operation:

```
PSUBUSB
fd[7..0]    ← SaturateToUnsignedByte(fs[7..0] - ft[7..0]) ;
* repeat add operation for 2nd through 7th bytes *;
fd[63..56]  ← SaturateToUnsignedByte(fs[63..56] - ft[63..56] );

PSUBUSH
fd[15..0]   ← SaturateToUnsignedHalfword(fs[15..0] - ft[15..0] );
* repeat add operation for 2nd and 3rd halfwords *;
fd[63..48]  ← SaturateToUnsignedHalfword(fs[63..48] - ft[63..48] );
```

Exception: None.

C.36 PUNPCKHBH/PUNPCKHHW/PUNPCKHWD - unpack high data

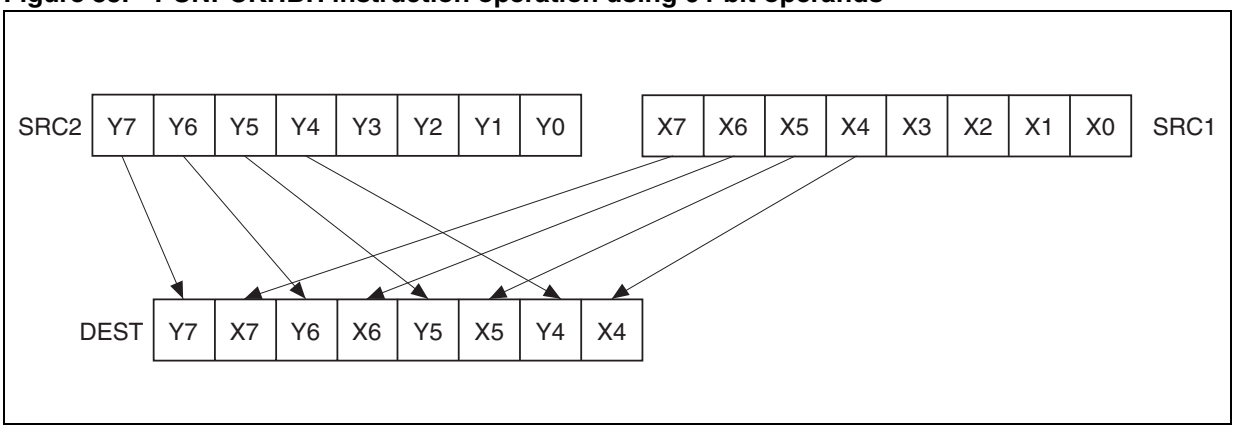


Format:

PUNPCKHBH	fd,fs,ft
PUNPCKHHW	fd,fs,ft
PUNPCKHWD	fd,fs,ft

Description: Unpacks and interleaves the high-order data elements (bytes,halfwords, words) of the first operand and second operand into the destination operand. (*Figure 33* shows the unpack operation for bytes in 64-bit operands.). The low-order data elements are ignored.

Figure 33. PUNPCKHBH instruction operation using 64-bit operands



The PUNPCKHBH instruction interleaves the high-order bytes of the first and second operands, the PUNPCKHHW instruction interleaves the high-order halfwords of the first and second operands, the PUNPCKHWD instruction interleaves the high-order word (or words) of first and second operands.

These instructions can be used to convert bytes to halfwords, halfwords to words, words to doublewords, respectively, by placing all 0s in the second operand. Here, if the second operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the first operand. For example, with the PUNPCKHBH instruction the high-order bytes are zero extended (that is, unpacked into unsigned halfword integers), and with the

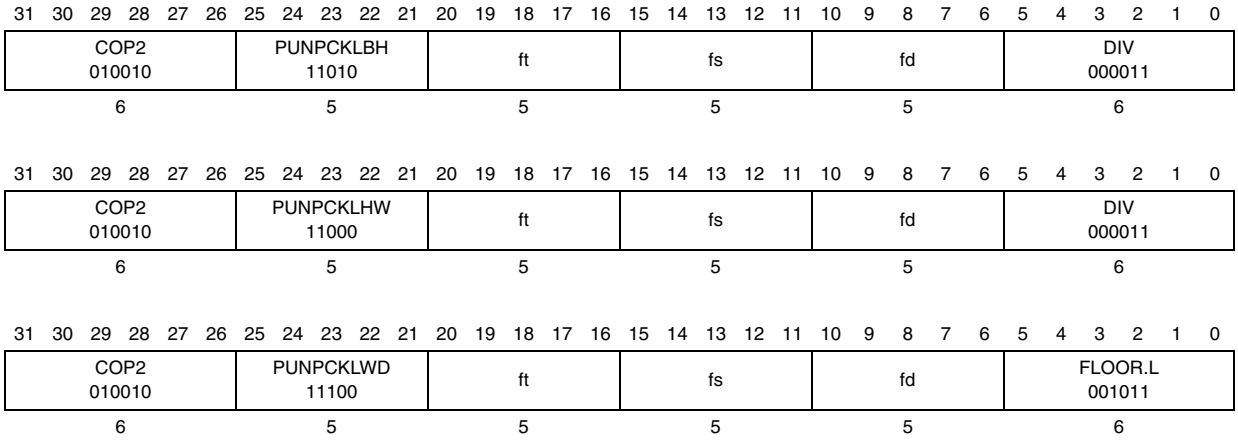
PUNPCKHHW instruction, the high-order halfwords are zero extended (unpacked into unsigned word integers).

Operation:

```
PUNPCKHBH
fd[7..0]   ← fs[39..32];
fd[15..8]  ← ft[39..32];
fd[23..16] ← fs[47..40];
fd[31..24] ← ft[47..40];
fd[39..32] ← fs[55..48];
fd[47..40] ← ft[55..48];
fd[55..48] ← fs[63..56];
fd[63..56] ← ft[63..56];

PUNPCKHHW
fd[15..0]  ← fs[47..32];
fd[31..16] ← ft[47..32];
fd[47..32] ← fs[63..48];
fd[63..48] ← ft[63..48];

PUNPCKHWD
fd[31..0]  ← fs[63..32]
fd[63..32] ← ft[63..32];
```

Exception: None.

C.37 PUNPCKLBH/PUNPCKLHW/PUNPCKLWD - unpack low data

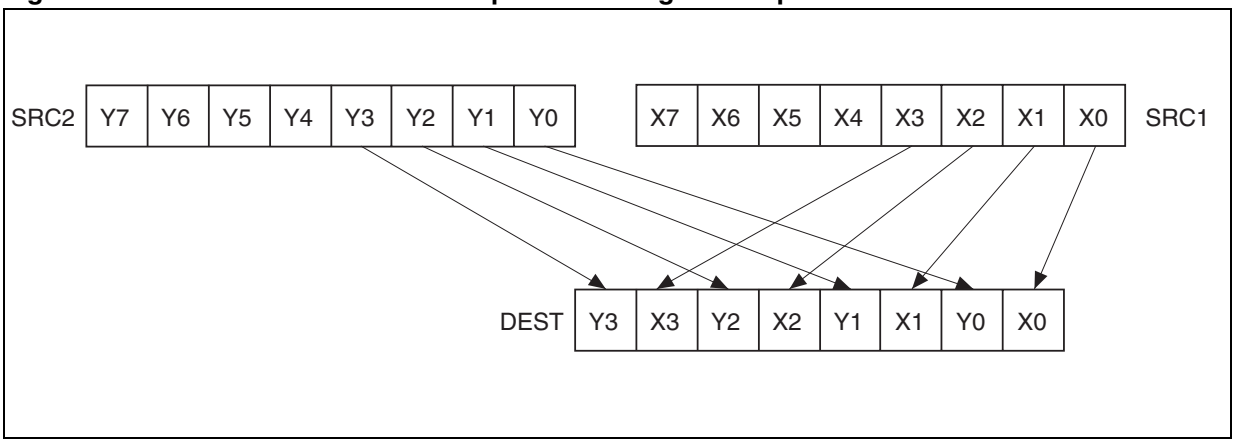


Format:

```
PUNPCKLBH   fd,fs,ft
PUNPCKLHW   fd,fs,ft
PUNPCKLWD   fd,fs,ft
```

Description: Unpacks and interleaves the low-order data elements (bytes, halfwords, words) of the first operand and second operand into the destination operand. (*Figure 34* shows the unpack operation for bytes in 64-bit operands.). The high-order data elements are ignored.

Figure 34. PUNPCKLBH instruction operation using 64-bit operands



The PUNPCKLBH instruction interleaves the low-order bytes of the first and second operands, the PUNPCKLHW instruction interleaves the low-order halfwords of the first and second operands, the PUNPCKLWD instruction interleaves the low-order word of the first and second operands.
These instructions can be used to convert bytes to halfwords, halfwords to words, words to doublewords, respectively, by placing all 0s in the second operand. Here, if the second operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the first operand. For example, with the PUNPCKLBH instruction the high-order bytes are zero extended (that is, unpacked into unsigned halfword integers), and with the PUNPCKLHW instruction, the high-order halfwords are zero extended (unpacked into unsigned word integers).

Operation:

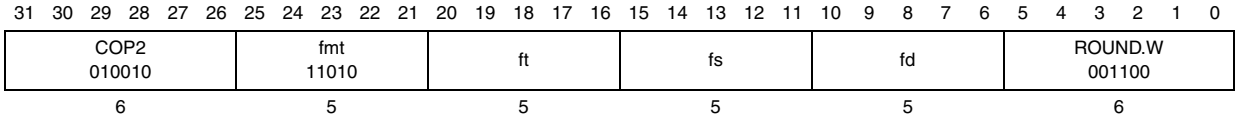
PUNPCKLBH
fd[63..56] ← ft[31..24];
fd[55..48] ← fs[31..24];
fd[47..40] ← ft[23..16];
fd[39..32] ← fs[23..16];
fd[31..24] ← ft[15..8];
fd[23..16] ← fs[15..8];
fd[15..8] ← ft[7..0];
fd[7..0] ← fs [7..0];

PUNPCKLHW
fd[63..48] ← ft[31..16];
fd[47..32] ← fs[31..16];
fd[31..16] ← ft[15..0];
fd[15..0] ← fs[15..0];

PUNPCKLWD
fd[63..32] ← ft[31..0];
fd[31..0] ← fs[31..0];

Exception: None.

C.38 Add - add word



Format: Add fd, fs,ft

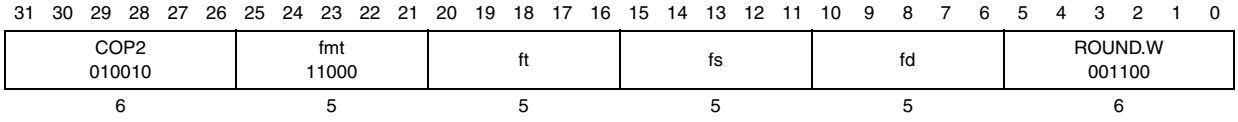
Purpose: To add 32-bit integers. If overflow occurs, then trap

Description: fd ← fs+ft
The 32-bit word value in FPR ft is added to the 32-bit value in FPR fs to produce a 32-bit result. If the addition results in 32-bit 2's complement arithmetic overflow then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 32-bit result is placed into FPR fd.

Operation:
```
If(NotWordValue(FPR[fs]) or NotWordValue(FPR[ft])) then UndefinedResult() endif
temp← FPR [fs]+FPR[ft]
if(32_bit_arithmetic_overflow) then
SignalException(IntegerOverflow)
else
FPR[fd]←sign_extend(temp31..0)
endif
```

Exception: Integer Overflow

C.39 Addu - add unsigned word



Format: Add fd, fs,ft

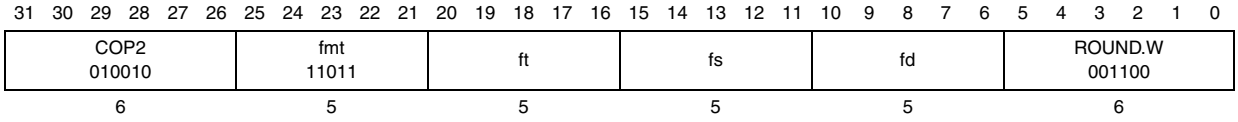
Purpose: To add 32-bit integers.

Description: fd ← fs+ft
The 32-bit word value in FPR ft is added to the 32-bit value in FPR fs and the 32-bit arithmetic result is placed into FPR fd.
No Integer Overflow exception occurs under any circumstances..

Operation:
```
If(NotWordValue(FPR[fs]) or NotWordValue(FPR[ft])) then UndefinedResult() endif
temp← FPR[fs]+FPR[ft]
FPR[fd] ←sign_extend(temp31..0)
```

Exception: None

C.40 Dadd - doubleword ADD



Format: Dadd fd, fs,ft

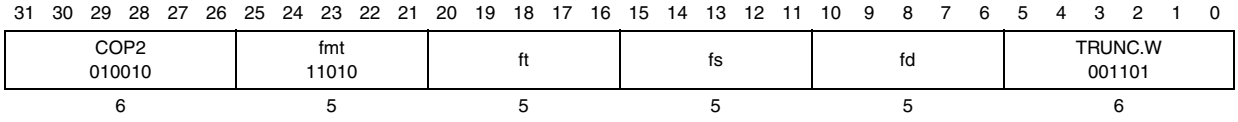
Purpose: To add 64-bit integers. If overflow occurs, then trap

Description: fd ← fs+ft
The 64-bit word value in FPR ft is added to the 64-bit value in FPR fs to produce a 64-bit result. If the addition results in 64-bit 2's complement arithmetic overflow then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 64-bit result is placed into FPR fd.

Operation:
```
64-bit processors
temp← FPR[fs]+FPR[ft]
if(64_bit_arithmetic_overflow) then
SignalException(IntegerOverflow)
else
FPR[fd] ←temp
endif
```

Exception: Integer Overflow

C.41 Sub - sub word



Format: Sub fd, fs,ft

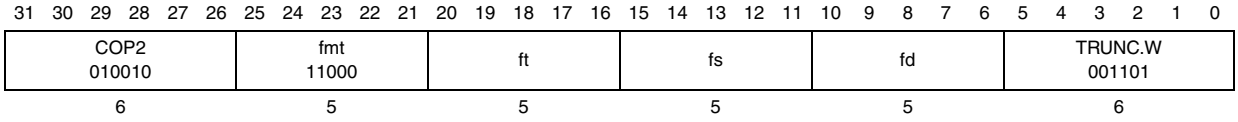
Purpose: To subtract 32-bit integers. If overflow occurs, then trap

Description: fd ← fs - ft
The 32-bit word value in FPR ft is subtracted from the 32-bit value in FPR fs to produce a 32-bit result. If the subtraction results in 32-bit 2's complement arithmetic overflow then the destination register is not modified and an Integer overflow exception occurs. If it does not overflow, the 32-bit result is placed into FPR fd.

Operation:
```
If(NotWordValue(FPR[fs]) or NotWordValue(FPR[ft])) then UndefinedResult() endif
temp← FPR[fs] - FPR[ft]
if(32_bit_arithmetic_overflow) then
SignalException(IntegerOverflow)
else
FPR[fd] ←temp
endif
```

Exception: Integer Overflow

C.42 Subu - sub unsigned word



Format: Sub fd, fs,ft

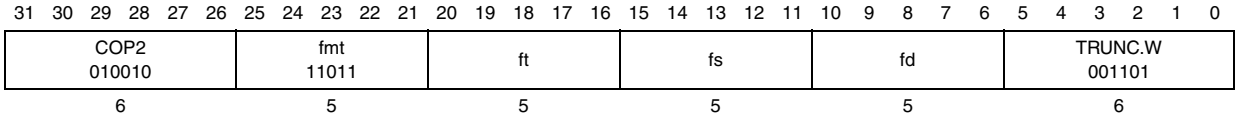
Purpose: To subtract 32-bit integers.

Description: fd ← fs - ft
The 32-bit word value in FPR ft is subtracted from the 32-bit value in FPR fs and the 32-bit arithmetic result is placed into FPR fd.
No Integer Overflow exception occurs under any circumstances.

Operation: If(NotWordValue(FPR[fs]) or NotWordValue(FPR[ft])) then UndefinedResult() endif
temp← FPR[fs]-FPR[ft]
FPR[fd] ←temp

Exception: None

C.43 Dsub - doubleword sub



Format: Dsub fd, fs,ft

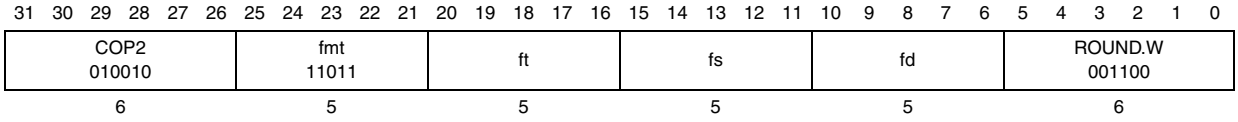
Purpose: To add 64-bit integers. If overflow occurs, then trap

Description: fd ← fs-ft
The 64-bit word value in FPR ft is subtracted from the 64-bit value in FPR fs to produce a 64-bit result. If the addition results in 64-bit 2's complement arithmetic overflow then the destination register is not modified and an Integer overflow exception occurs. If it does not overflow, the 64-bit result is placed into FPR fd.

Operation: 64-bit processors
temp← FPR[fs]-FPR[ft]
if(64_bit_arithmetic_overflow) then
SignalException(IntegerOverflow)
else
FPR[fd] ←temp
endif

Exception: Integer Overflow

C.44 Or - or



Format: Or fd, fs,ft

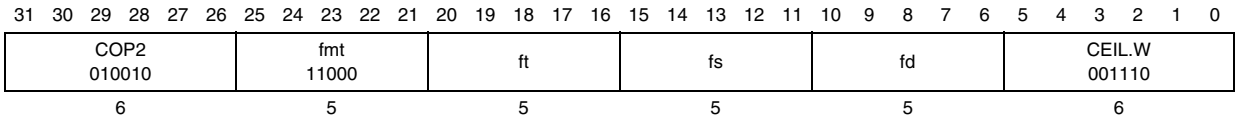
Purpose: To do a bitwise logic or

Description: fd ← fs Or ft
The contents of FPR fs are combined with the contents of FPR ft in a bitwise logic OR operation. The result is placed into FPR fd.

Operation: FPR[fd] ←FPR[fs] or FPR[ft]

Exception: None

C.45 Sll - shift word left logical



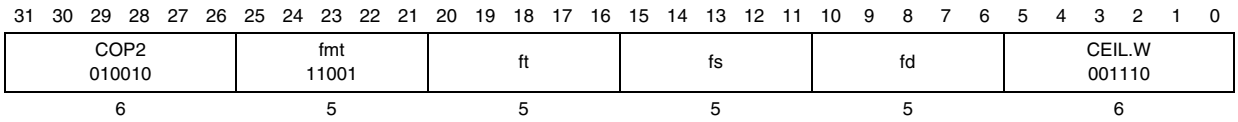
Format: Sll fd,fs,ft

Description: fd ← mfs << Value(FPR[ft])
The contents of the low-order 32-bit word of FPR fs are shifted left, inserting zeros into the emptied bits; the word result is placed in FPR fd. the bit shift count is specified by Value of FPR ft, the result word is sign-extended.

Operation: s ← Value(FPR[ft])
if s >= 32 then
FPR[fd] ← 0^{64}
else
Temp ← $FPR[fs]_{(31-s)..0} || 0^{s}$
FPR[fd]←sign_extend(temp)

Exception: None.

C.46 Dsll - doubleword shift left logical



Format: fd ←fs << Value(FPR[ft])
Sll fd,fs,ft

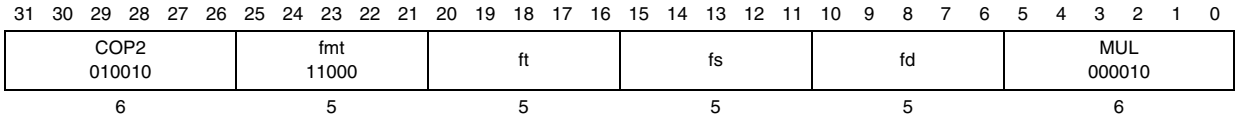
Description: The contents of the low-order 64-bit word of FPR fs are shifted left, inserting zeros into the emptied bits; the result is placed in FPR fd. the bit shift count is specified by value of FPR ft.

Operation:

```
s ← Value(FPR[ft])
if s > = 64 then
FPR[fd] ← 0^64
else
Temp ← FPR[fs]_(63-s)..0 || 0^s
FPR[fd] ← temp
```

Exception: None.

C.47 Xor - xor



Format: Xor fd, fs,ft

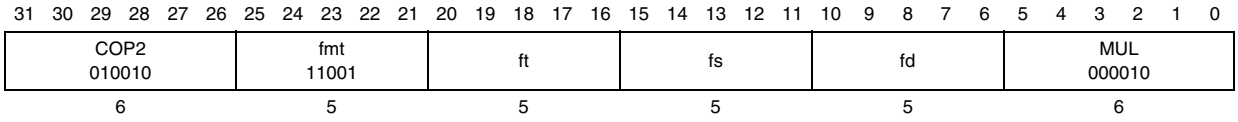
Purpose: To do a bitwise logic Xor

Description: fd ← fs Xor ft
The contents of FPR fs are combined with the contents of FPR ft in a bitwise logic XOR operation. The result is placed into FPR fd.

Operation: FPR[fd] ← FPR[fs] xor FPR[ft]

Exception: None

C.48 Nor - nor



Format: Nor fd, fs,ft

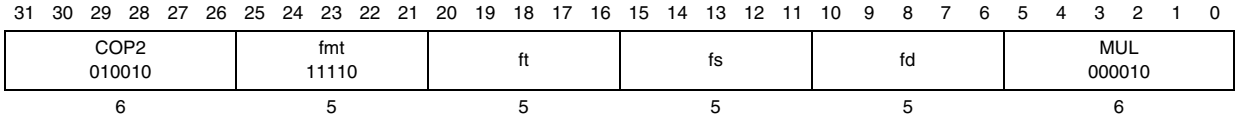
Purpose: To do a bitwise logic Nor

Description: fd ← fs Nor ft
The contents of FPR fs are combined with the contents of FPR ft in a bitwise logic NOR operation. The result is placed into FPR fd.

Operation: FPR[fd] ← FPR[fs] nor FPR[ft]

Exception: None

C.49 And - and



Format: And fd, fs,ft

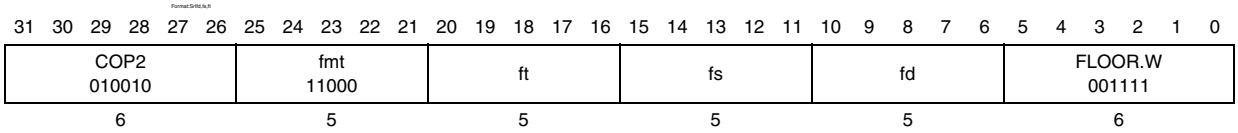
Purpose: To do a bitwise logic And

Description: fd ← fs And ft
The contents of FPR fs are combined with the contents of FPR ft in a bitwise logic AND operation. The result is placed into FPR fd.

Operation: FPR[fd] ← FPR[fs] and FPR[ft]

Exception: None

C.50 Srl - shift word right logical



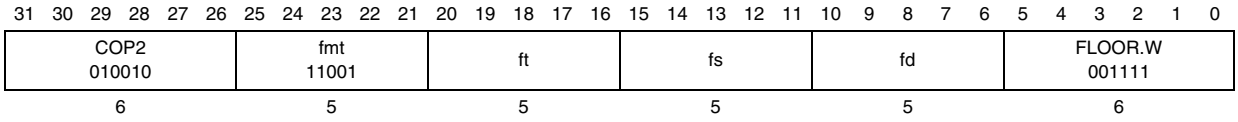
Format: Srl
fd,fs,ft

Description: fd ← fs >> Value(FPR[ft])
The contents of the low-order 32-bit word of FPR ft are shifted right, inserting zeros into the emptied bits; the word result is placed in FPR fd. the bit shift count is specified by Value of FPR ft, the result word is sign-extended.

Operation:
s ← Value(FPR[ft])
if s >= 32 then
FPR[fd] ← 0^{64}
else
Temp ← 0^{s} || $FPR[fs]_{(31-s)..0}$
FPR[fd] ← sign_extend(temp)

Exception: None.

C.51 Dsrl - doubleword shift right logical



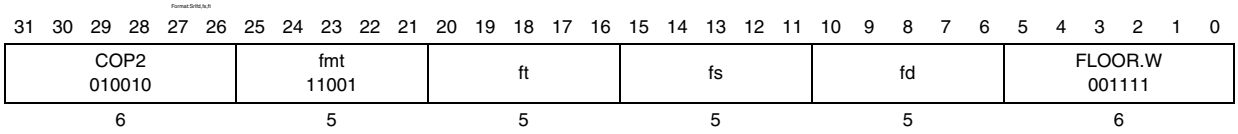
Format: DSrl
fd,fs,ft

Description: fd ← fs >> Value(FPR[ft])
The contents of the low-order 64-bit word of FPR ft are shifted right, inserting zeros into the emptied bits; the result is placed in FPR fd. the bit shift count is specified by Value of FPR ft.

Operation:
s ← Value(FPR[ft])
if s > = 64 then
FPR[fd] ←0^{64}
else
Temp ← 0^{s} || $FPR[ft]_{(63-s)..0}$
FPR[fd] ← temp

Exception: None.

C.52 Sra - shift word right arithmetic



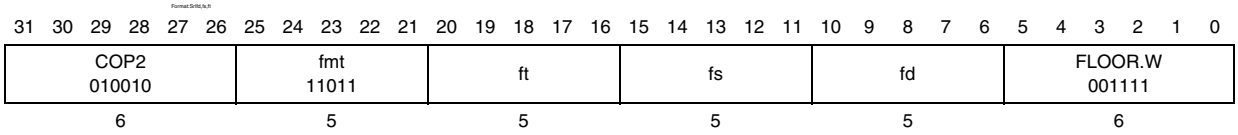
Format: Srl
fd,fs,ft

Description: fd ← fs >> Value(FPR[ft]) (arithmetic)
The contents of the low-order 32-bit word of FPR ft are shifted right, duplicating the sign-bit (bit 31) in the emptied bits; the word result is placed in FPR fd. the bit shift count is specified by Value of FPR ft, the result word is sign-extended.

Operation:
If (NotWordValue(FPR[fs]) then undefinedResult() endif
s ← Value(FPR[ft])
if s > = 32 then
FPR[fd] ← $(FPR[fs]_{31})^{64}$
else
Temp ← $(FPR[fs]_{31})^{s}$ || $FPR[fs]_{(31-s)..0}$
FPR[fd] ← sign_extend(temp)

Exception: None.

C.53 Dsra - doubleword shift right arithmetic



Format: DSra
fd,fs,ft

Description: fd ← fs >> Value(FPR[ft])
The contents of the low-order 64-bit word of FPR ft are shifted right, duplicating the sign-bit (bit 63) in the emptied bits; the result is placed in FPR fd. The bit shift count is specified by Value of FPR ft.

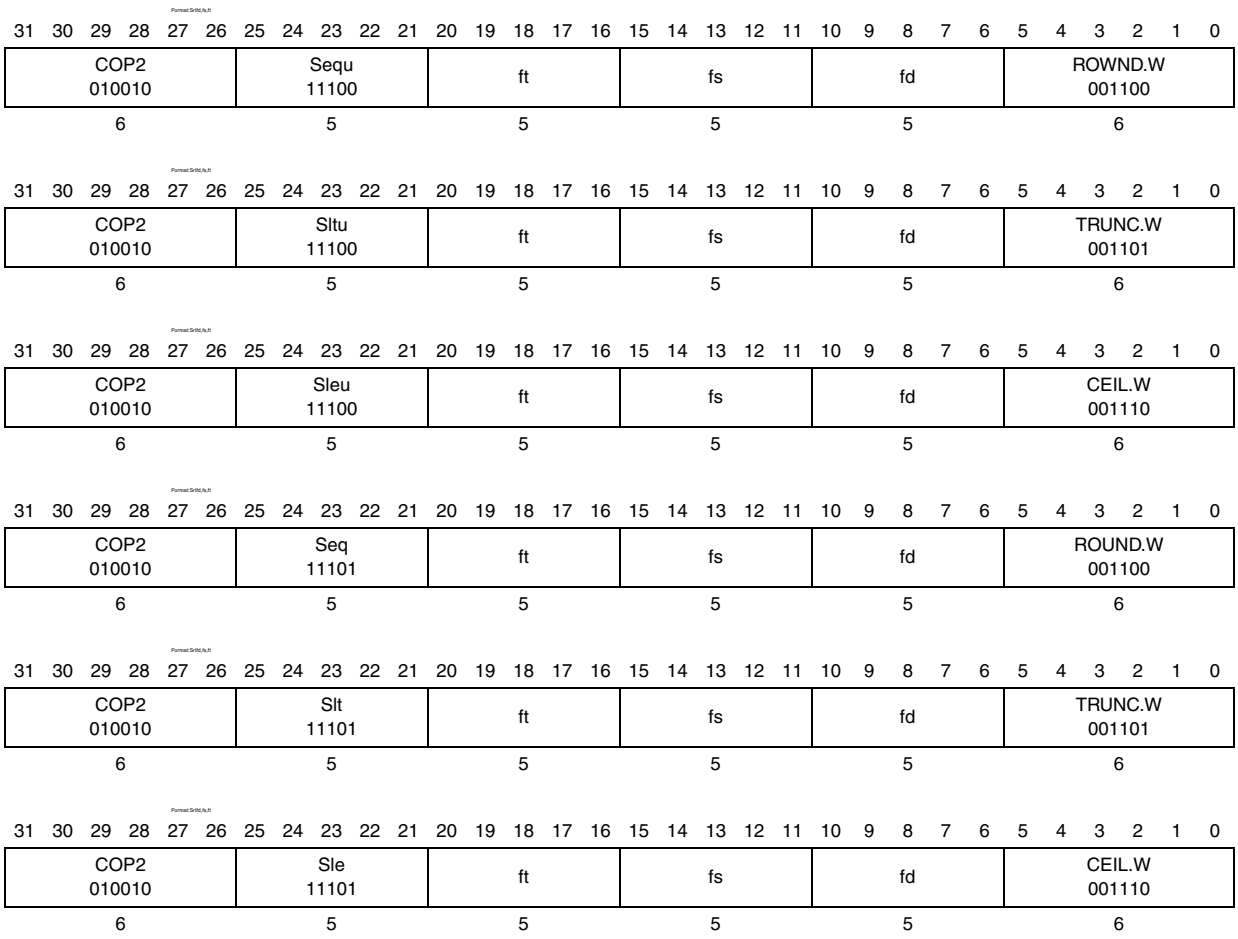
Operation:

s ← Value(FPR[ft])
if s > = 64 then
FPR[fd] ← $(FPR[fs]_{63})^{64}$
else
Temp ← $(FPR[fs]_{63})^{s}$ || $FPR[ft]_{(63-s)..0}$
FPR[fd] ← tempNone.

Exception:

None.

C.54 Sequ/seq/sltu/slt/sleu/sle - fixing-point compare set cc bit



Format:

Sequ
fs, ft
Sltu
fs, ft
Sleu
fs, ft
Seq
fs, ft
Slt
fs, ft

Sle
fs, ft

Description: Description: The value in FPR fs is compared to the value in FPR ft. The comparison is exact and neither overflows nor underflows. The result is written into condition code cc, true is 1 and false is 0.

Operation:

Sequ/seq
condition ← ValueFPR(fs,fmt) = ValueFPR(ft,fmt)
FCC[cc] ← condition

Sltu/slt
condition ← ValueFPR(fs,fmt) < ValueFPR(ft,fmt)
FCC[cc] ← condition

Sleu/sle
condition ← ValueFPR(fs,fmt) <= ValueFPR(ft,fmt)
FCC[cc] ← condition

Exception: None.

14 Revision history

Table 65. Document revision history

Date	Revision	Changes
6-Aug-2007	1	Initial release.