32-Bit TX System RISC

TX39 Family

Architecture

TOSHIBA

Preface

Thank you for your new or continued patronage of Toshiba semiconductor products. This is the 2000 edition of the databook for the TX39 Family of 32-bit RISC microprocessors, entitled 32-Bit TX System RISC TX39 Family Architecture.

This databook is written so as to be accessible to engineers who may be designing a Toshiba microprocessor into their products for the first time. No prior knowledge of these devices is assumed. The databook includes a review of the architecture of the processor family, a description of the TX39 instruction set and sections dedicated to various other relevant topics, such as the memory management unit (MMU) and CPU exceptions.

Toshiba continually updates its technical information. Your comments and suggestions concerning this and other Toshiba documents are sincerely appreciated and may be used in subsequent editions. For updates to this document or for additional information about the product, please contact your nearest Toshiba office or authorized Toshiba dealer.

January 2000

CONTENTS

Handling Precautions

Architecture

Handling Precautions

1. Using Toshiba Semiconductors Safely

TOSHIBA is continually working to improve the quality and the reliability of its products.

Nevertheless, semiconductor devices in general can malfunction or fail due to their inherent electrical sensitivity and vulnerability to physical stress. It is the responsibility of the buyer, when utilizing TOSHIBA products, to observe standards of safety, and to avoid situations in which a malfunction or failure of a TOSHIBA product could cause loss of human life, bodily injury or damage to property.

In developing your designs, please ensure that TOSHIBA products are used within specified operating ranges as set forth in the most recent products specifications. Also, please keep in mind the precautions and conditions set forth in the TOSHIBA Semiconductor Reliability Handbook.

2. Safety Precautions

This section lists important precautions which users of semiconductor devices (and anyone else) should observe in order to avoid injury and damage to property, and to ensure safe and correct use of devices.

Please be sure that you understand the meanings of the labels and the graphic symbol described below before you move on to the detailed descriptions of the precautions.

[Explanation of labels]

⚠DANGER	Indicates an imminently hazardous situation which will result in death or serious injury if you do not follow instructions.
⚠WARNING	Indicates a potentially hazardous situation which could result in death or serious injury if you do not follow instructions.
⚠CAUTION	Indicates a potentially hazardous situation which if not avoided, may result in minor injury or moderate injury.

[Explanation of graphic symbol]

Graphic symbol	Meaning
	Indicates that caution is required (laser beam is dangerous to eyes).

2.1 General Precautions regarding Semiconductor Devices

⚠CAUTION
Do not use devices under conditions exceeding their absolute maximum ratings (e.g. current, voltage, power dissipation or temperature). This may cause the device to break down, degrade its performance, or cause it to catch fire or explode resulting in injury. Do not insert devices in the wrong orientation. Make sure that the positive and negative terminals of power supplies are connected correctly. Otherwise the rated maximum current or power dissipation may be exceeded and the device may break down or undergo performance degradation, causing it to catch fire or explode and resulting in injury. When power to a device is on, do not touch the device's heat sink. Heat sinks become hot, so you may burn your hand. Do not touch the tips of device leads. Because some types of device have leads with pointed tips, you may prick your finger. When conducting any kind of evaluation, inspection or testing, be sure to connect the testing equipment's electrodes or probes to the pins of the device under test before powering it on. Otherwise, you may receive an electric shock causing injury. Before grounding an item of measuring equipment or a soldering iron, check that there is no electrical leakage from it. Electrical leakage may cause the device which you are testing or soldering to break down, or could give you an electric shock. Always wear protective glasses when cutting the leads of a device with clippers or a similar tool. If you do not, small bits of metal flying off the cut ends may damage your eyes.

2.2 Precautions Specific to Each Product Group

2.2.1 Optical semiconductor devices

⚠ DANGER

When a visible semiconductor laser is operating, do not look directly into the laser beam or look through the optical system. This is highly likely to impair vision, and in the worst case may cause blindness.
If it is necessary to examine the laser apparatus, for example to inspect its optical characteristics, always wear the appropriate type of laser protective glasses as stipulated by IEC standard IEC825-1.

⚠ WARNING

Ensure that the current flowing in an LED device does not exceed the device's maximum rated current.
This is particularly important for resin-packaged LED devices, as excessive current may cause the package resin to blow up, scattering resin fragments and causing injury.

When testing the dielectric strength of a photocoupler, use testing equipment which can shut off the supply voltage to the photocoupler. If you detect a leakage current of more than 100 μA, use the testing equipment to shut off the photocoupler's supply voltage; otherwise a large short-circuit current will flow continuously, and the device may break down or burst into flames, resulting in fire or injury.

When incorporating a visible semiconductor laser into a design, use the device's internal photodetector or a separate photodetector to stabilize the laser's radiant power so as to ensure that laser beams exceeding the laser's rated radiant power cannot be emitted.
If this stabilizing mechanism does not work and the rated radiant power is exceeded, the device may break down or the excessively powerful laser beams may cause injury.

2.2.2 Power devices

⚠ DANGER

Never touch a power device while it is powered on. Also, after turning off a power device, do not touch it until it has thoroughly discharged all remaining electrical charge.
Touching a power device while it is powered on or still charged could cause a severe electric shock, resulting in death or serious injury.

When conducting any kind of evaluation, inspection or testing, be sure to connect the testing equipment's electrodes or probes to the device under test before powering it on.
When you have finished, discharge any electrical charge remaining in the device.
Connecting the electrodes or probes of testing equipment to a device while it is powered on may result in electric shock, causing injury.

⚠WARNING

Do not use devices under conditions which exceed their absolute maximum ratings (current, voltage, power dissipation, temperature etc.).
This may cause the device to break down, causing a large short-circuit current to flow, which may in turn cause it to catch fire or explode, resulting in fire or injury.

Use a unit which can detect short-circuit currents and which will shut off the power supply if a short-circuit occurs.
If the power supply is not shut off, a large short-circuit current will flow continuously, which may in turn cause the device to catch fire or explode, resulting in fire or injury.

When designing a case for enclosing your system, consider how best to protect the user from shrapnel in the event of the device catching fire or exploding.
Flying shrapnel can cause injury.

When conducting any kind of evaluation, inspection or testing, always use protective safety tools such as a cover for the device.
Otherwise you may sustain injury caused by the device catching fire or exploding.

Make sure that all metal casings in your design are grounded to earth.
Even in modules where a device's electrodes and metal casing are insulated, capacitance in the module may cause the electrostatic potential in the casing to rise.
Dielectric breakdown may cause a high voltage to be applied to the casing, causing electric shock and injury to anyone touching it.

When designing the heat radiation and safety features of a system incorporating high-speed rectifiers, remember to take the device's forward and reverse losses into account.
The leakage current in these devices is greater than that in ordinary rectifiers; as a result, if a high-speed rectifier is used in an extreme environment (e.g. at high temperature or high voltage), its reverse loss may increase, causing thermal runaway to occur.
This may in turn cause the device to explode and scatter shrapnel, resulting in injury to the user.

A design should ensure that, except when the main circuit of the device is active, reverse bias is applied to the device gate while electricity is conducted to control circuits, so that the main circuit will become inactive.
Malfunction of the device may cause serious accidents or injuries.

⚠CAUTION

When conducting any kind of evaluation, inspection or testing, either wear protective gloves or wait until the device has cooled properly before handling it.
Devices become hot when they are operated. Even after the power has been turned off, the device will retain residual heat which may cause a burn to anyone touching it.

2.2.3 Bipolar ICs (for use in automobiles)

⚠CAUTION

If your design includes an inductive load such as a motor coil, incorporate diodes or similar devices into the design to prevent negative current from flowing in.
The load current generated by powering the device on and off may cause it to function erratically or to break down, which could in turn cause injury.

Ensure that the power supply to any device which incorporates protective functions is stable.
If the power supply is unstable, the device may operate erratically, preventing the protective functions from working correctly. If

protective functions fail, the device may break down causing injury to the user.

3. General Safety Precautions and Usage Considerations

This section is designed to help you gain a better understanding of semiconductor devices, so as to ensure the safety, quality and reliability of the devices which you incorporate into your designs.

3.1 From Incoming to Shipping

3.1.1 Electrostatic discharge (ESD)

When handling individual devices (which are not yet mounted on a printed circuit board), be sure that the environment is protected against electrostatic electricity. Operators should wear anti-static clothing, and containers and other objects which come into direct contact with devices should be made of anti-static materials and should be grounded to earth via an 0.5- to 1.0-MΩ protective resistor.

Please follow the precautions described below; this is particularly important for devices which are marked “Be careful of static.”.

(1) Work environment

- When humidity in the working environment decreases, the human body and other insulators can easily become charged with static electricity due to friction. Maintain the recommended humidity of 40% to 60% in the work environment, while also taking into account the fact that moisture-proof-packed products may absorb moisture after unpacking.

- Be sure that all equipment, jigs and tools in the working area are grounded to earth.

- Place a conductive mat over the floor of the work area, or take other appropriate measures, so that the floor surface is protected against static electricity and is grounded to earth. The surface resistivity should be 10^4 to 10^8 Ω/sq and the resistance between surface and ground, 7.5×10^5 to 10^8 Ω

- Cover the workbench surface also with a conductive mat (with a surface resistivity of 10^4 to 10^8 Ω/sq, for a resistance between surface and ground of 7.5×10^5 to 10^8 Ω) . The purpose of this is to disperse static electricity on the surface (through resistive components) and ground it to earth. Workbench surfaces must not be constructed of low-resistance metallic materials that allow rapid static discharge when a charged device touches them directly.

- Pay attention to the following points when using automatic equipment in your workplace:

  (a) When picking up ICs with a vacuum unit, use a conductive rubber fitting on the end of the pick-up wand to protect against electrostatic charge.

  (b) Minimize friction on IC package surfaces. If some rubbing is unavoidable due to the device’s mechanical structure, minimize the friction plane or use material with a small friction coefficient and low electrical resistance. Also, consider the use of an ionizer.

  (c) In sections which come into contact with device lead terminals, use a material which dissipates static electricity.

  (d) Ensure that no statically charged bodies (such as work clothes or the human body) touch the devices.

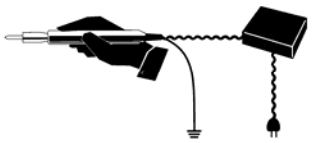
(e) Make sure that sections of the tape carrier which come into contact with installation devices or other electrical machinery are made of a low-resistance material.

(f) Make sure that jigs and tools used in the assembly process do not touch devices.

(g) In processes in which packages may retain an electrostatic charge, use an ionizer to neutralize the ions.

- Make sure that CRT displays in the working area are protected against static charge, for example by a VDT filter. As much as possible, avoid turning displays on and off. Doing so can cause electrostatic induction in devices.
- Keep track of charged potential in the working area by taking periodic measurements.
- Ensure that work chairs are protected by an anti-static textile cover and are grounded to the floor surface by a grounding chain. (Suggested resistance between the seat surface and grounding chain is 7.5×10^5 to $10^{12}\Omega$.)
- Install anti-static mats on storage shelf surfaces. (Suggested surface resistivity is 10^4 to 10^8 Ω/sq; suggested resistance between surface and ground is 7.5×10^5 to 10^8 Ω.)
- For transport and temporary storage of devices, use containers (boxes, jigs or bags) that are made of anti-static materials or materials which dissipate electrostatic charge.
- Make sure that cart surfaces which come into contact with device packaging are made of materials which will conduct static electricity, and verify that they are grounded to the floor surface via a grounding chain.
- In any location where the level of static electricity is to be closely controlled, the ground resistance level should be Class 3 or above. Use different ground wires for all items of equipment which may come into physical contact with devices.

(2) Operating environment



- Operators must wear anti-static clothing and conductive shoes (or a leg or heel strap).
- Operators must wear a wrist strap grounded to earth via a resistor of about 1 MΩ.
- Soldering irons must be grounded from iron tip to earth, and must be used only at low voltages (6 V to 24 V).
- If the tweezers you use are likely to touch the device terminals, use anti-static tweezers and in particular avoid metallic tweezers. If a charged device touches a low-resistance tool, rapid discharge can occur. When using vacuum tweezers, attach a conductive chucking pat to the tip, and connect it to a dedicated ground used especially for anti-static purposes (suggested resistance value: 10^4 to 10^8 Ω).
- Do not place devices or their containers near sources of strong electrical fields (such as above a CRT).

- When storing printed circuit boards which have devices mounted on them, use a board container or bag that is protected against static charge. To avoid the occurrence of static charge or discharge due to friction, keep the boards separate from one other and do not stack them directly on top of one another.

- Ensure, if possible, that any articles (such as clipboards) which are brought to any location where the level of static electricity must be closely controlled are constructed of anti-static materials.

- In cases where the human body comes into direct contact with a device, be sure to wear anti-static finger covers or gloves (suggested resistance value: 10^8 Ω or less).

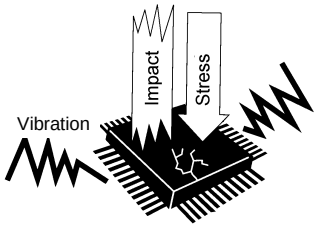
- Equipment safety covers installed near devices should have resistance ratings of 10^9 Ω or less.

- If a wrist strap cannot be used for some reason, and there is a possibility of imparting friction to devices, use an ionizer.

- The transport film used in TCP products is manufactured from materials in which static charges tend to build up. When using these products, install an ionizer to prevent the film from being charged with static electricity. Also, ensure that no static electricity will be applied to the product's copper foils by taking measures to prevent static occuring in the peripheral equipment.

3.1.2 Vibration, impact and stress

Handle devices and packaging materials with care. To avoid damage to devices, do not toss or drop packages. Ensure that devices are not subjected to mechanical vibration or shock during transportation. Ceramic package devices and devices in canister-type packages which have empty space inside them are subject to damage from vibration and shock because the bonding wires are secured only at their ends.



Plastic molded devices, on the other hand, have a relatively high level of resistance to vibration and mechanical shock because their bonding wires are enveloped and fixed in resin. However, when any device or package type is installed in target equipment, it is to some extent susceptible to wiring disconnections and other damage from vibration, shock and stressed solder junctions. Therefore when devices are incorporated into the design of equipment which will be subject to vibration, the structural design of the equipment must be thought out carefully.

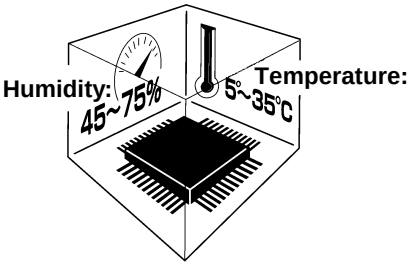
If a device is subjected to especially strong vibration, mechanical shock or stress, the package or the chip itself may crack. In products such as CCDs which incorporate window glass, this could cause surface flaws in the glass or cause the connection between the glass and the ceramic to separate.

Furthermore, it is known that stress applied to a semiconductor device through the package changes the resistance characteristics of the chip because of piezoelectric effects. In analog circuit design attention must be paid to the problem of package stress as well as to the dangers of vibration and shock as described above.

3.2 Storage

3.2.1 General storage

- Avoid storage locations where devices will be exposed to moisture or direct sunlight.
- Follow the instructions printed on the device cartons regarding transportation and storage.
- The storage area temperature should be kept within a temperature range of 5°C to 35°C, and relative humidity should be maintained at between 45% and 75%.



- Do not store devices in the presence of harmful (especially corrosive) gases, or in dusty conditions.
- Use storage areas where there is minimal temperature fluctuation. Rapid temperature changes can cause moisture to form on stored devices, resulting in lead oxidation or corrosion. As a result, the solderability of the leads will be degraded.
- When repacking devices, use anti-static containers.
- Do not allow external forces or loads to be applied to devices while they are in storage.
- If devices have been stored for more than two years, their electrical characteristics should be tested and their leads should be tested for ease of soldering before they are used.

3.2.2 Moisture-proof packing

Moisture-proof packing should be handled with care. The handling procedure specified for each packing type should be followed scrupulously. If the proper procedures are not followed, the quality and reliability of devices may be degraded. This section describes general precautions for handling moisture-proof packing. Since the details may differ from device to device, refer also to the relevant individual datasheets or databook.

(1) General precautions
Follow the instructions printed on the device cartons regarding transportation and storage.

- Do not drop or toss device packing. The laminated aluminum material in it can be rendered ineffective by rough handling.
- The storage area temperature should be kept within a temperature range of 5°C to 30°C, and relative humidity should be maintained at 90% (max). Use devices within 12 months of the date marked on the package seal.

- If the 12-month storage period has expired, or if the 30% humidity indicator shown in Figure 1 is pink when the packing is opened, it may be advisable, depending on the device and packing type, to back the devices at high temperature to remove any moisture. Please refer to the table below. After the pack has been opened, use the devices in a 5°C to 30°C. 60% RH environment and within the effective usage period listed on the moisture-proof package. If the effective usage period has expired, or if the packing has been stored in a high-humidity environment, back the devices at high temperature.

Packing	Moisture removal
Tray	If the packing bears the “Heatproof” marking or indicates the maximum temperature which it can withstand, bake at 125°C for 20 hours. (Some devices require a different procedure.)
Tube	Transfer devices to trays bearing the “Heatproof” marking or indicating the temperature which they can withstand, or to aluminum tubes before baking at 125°C for 20 hours.
Tape	Deviced packed on tape cannot be baked and must be used within the effective usage period after unpacking, as specified on the packing.

- When baking devices, protect the devices from static electricity.

- Moisture indicators can detect the approximate humidity level at a standard temperature of 25°C. 6-point indicators and 3-point indicators are currently in use, but eventually all indicators will be 3-point indicators.

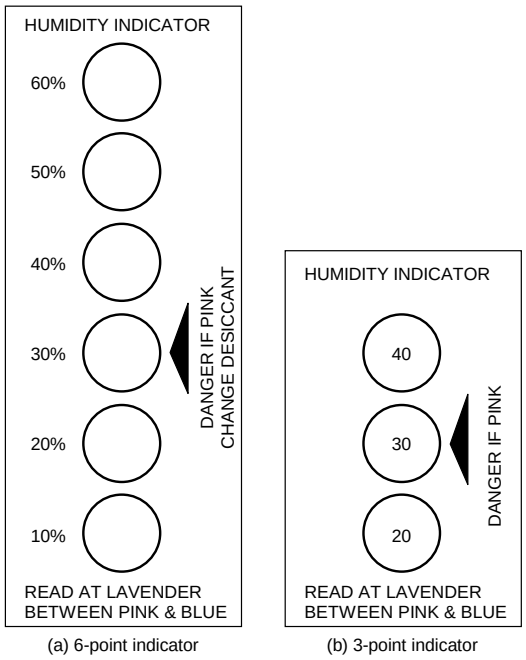


Figure 1 Humidity indicator

3.3 Design

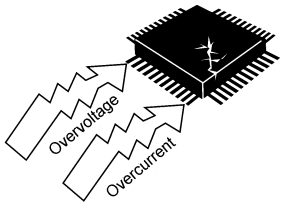
Care must be exercised in the design of electronic equipment to achieve the desired reliability. It is important not only to adhere to specifications concerning absolute maximum ratings and recommended operating conditions, it is also important to consider the overall environment in which equipment will be used, including factors such as the ambient temperature, transient noise and voltage and current surges, as well as mounting conditions which affect device reliability. This section describes some general precautions which you should observe when designing circuits and when mounting devices on printed circuit boards.

For more detailed information about each product family, refer to the relevant individual technical datasheets available from Toshiba.

3.3.1 Absolute maximum ratings

⚠CAUTION Do not use devices under conditions in which their absolute maximum ratings (e.g. current, voltage, power dissipation or temperature) will be exceeded. A device may break down or its performance may be degraded, causing it to catch fire or explode resulting in injury to the user.

The absolute maximum ratings are rated values which must not be exceeded during operation, even for an instant. Although absolute maximum ratings differ from product to product, they essentially concern the voltage and current at each pin, the allowable power dissipation, and the junction and storage temperatures.



If the voltage or current on any pin exceeds the absolute maximum rating, the device's internal circuitry can become degraded. In the worst case, heat generated in internal circuitry can fuse wiring or cause the semiconductor chip to break down.

If storage or operating temperatures exceed rated values, the package seal can deteriorate or the wires can become disconnected due to the differences between the thermal expansion coefficients of the materials from which the device is constructed.

3.3.2 Recommended operating conditions

The recommended operating conditions for each device are those necessary to guarantee that the device will operate as specified in the datasheet.
If greater reliability is required, derate the device's absolute maximum ratings for voltage, current, power and temperature before using it.

3.3.3 Derating

When incorporating a device into your design, reduce its rated absolute maximum voltage, current, power dissipation and operating temperature in order to ensure high reliability.
Since derating differs from application to application, refer to the technical datasheets available for the various devices used in your design.

3.3.4 Unused pins

If unused pins are left open, some devices can exhibit input instability problems, resulting in malfunctions such as abrupt increase in current flow. Similarly, if the unused output pins on a device are connected to the power supply pin, the ground pin or to other output pins, the IC may malfunction or break down.

Since the details regarding the handling of unused pins differ from device to device and from pin to pin, please follow the instructions given in the relevant individual datasheets or databook.

CMOS logic IC inputs, for example, have extremely high impedance. If an input pin is left open, it can easily pick up extraneous noise and become unstable. In this case, if the input voltage level reaches an intermediate level, it is possible that both the P-channel and N-channel transistors will be turned on, allowing unwanted supply current to flow. Therefore, ensure that the unused input pins of a device are connected to the power supply (Vcc) pin or ground (GND) pin of the same device. For details of what to do with the pins of heat sinks, refer to the relevant technical datasheet and databook.

3.3.5 Latch-up

Latch-up is an abnormal condition inherent in CMOS devices, in which Vcc gets shorted to ground. This happens when a parasitic PN-PN junction (thyristor structure) internal to the CMOS chip is turned on, causing a large current of the order of several hundred mA or more to flow between Vcc and GND, eventually causing the device to break down.

Latch-up occurs when the input or output voltage exceeds the rated value, causing a large current to flow in the internal chip, or when the voltage on the Vcc (Vdd) pin exceeds its rated value, forcing the internal chip into a breakdown condition. Once the chip falls into the latch-up state, even though the excess voltage may have been applied only for an instant, the large current continues to flow between Vcc (Vdd) and GND (Vss). This causes the device to heat up and, in extreme cases, to emit gas fumes as well. To avoid this problem, observe the following precautions:

(1) Do not allow voltage levels on the input and output pins either to rise above Vcc (Vdd) or to fall below GND (Vss). Also, follow any prescribed power-on sequence, so that power is applied gradually or in steps rather than abruptly.

(2) Do not allow any abnormal noise signals to be applied to the device.

(3) Set the voltage levels of unused input pins to Vcc (Vdd) or GND (Vss).

(4) Do not connect output pins to one another.

3.3.6 Input/Output protection

Wired-AND configurations, in which outputs are connected together, cannot be used, since this short-circuits the outputs. Outputs should, of course, never be connected to Vcc (Vdd) or GND (Vss).

Furthermore, ICs with tri-state outputs can undergo performance degradation if a shorted output current is allowed to flow for an extended period of time. Therefore, when designing circuits, make sure that tri-state outputs will not be enabled simultaneously.

3.3.7 Load capacitance

Some devices display increased delay times if the load capacitance is large. Also, large charging and discharging currents will flow in the device, causing noise. Furthermore, since outputs are shorted for a relatively long time, wiring can become fused.

Consult the technical information for the device being used to determine the recommended load capacitance.

3.3.8 Thermal design

The failure rate of semiconductor devices is greatly increased as operating temperatures increase. As shown in Figure 2, the internal thermal stress on a device is the sum of the ambient temperature and the temperature rise due to power dissipation in the device. Therefore, to achieve optimum reliability, observe the following precautions concerning thermal design:

(1) Keep the ambient temperature (Ta) as low as possible.

(2) If the device's dynamic power dissipation is relatively large, select the most appropriate circuit board material, and consider the use of heat sinks or of forced air cooling. Such measures will help lower the thermal resistance of the package.

(3) Derate the device's absolute maximum ratings to minimize thermal stress from power dissipation.

$\theta ja = \theta jc + \theta ca$

$\theta ja = (Tj - Ta) / P$

$\theta jc = (Tj - Tc) / P$

$\theta ca = (Tc - Ta) / P$

in which θja = thermal resistance between junction and surrounding air (°C/W)

θjc = thermal resistance between junction and package surface, or internal thermal resistance (°C/W)

θca = thermal resistance between package surface and surrounding air, or external thermal resistance (°C/W)

Tj = junction temperature or chip temperature (°C)

Tc = package surface temperature or case temperature (°C)

Ta = ambient temperature (°C)

P = power dissipation (W)

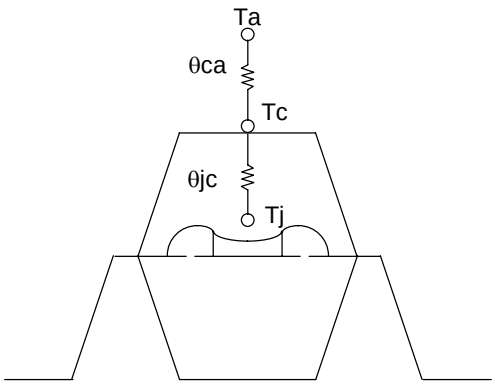


Figure 2 Thermal resistance of package

3.3.9 Interfacing

When connecting inputs and outputs between devices, make sure input voltage (V_{IL}/V_{IH}) and output voltage (V_{OL}/V_{OH}) levels are matched. Otherwise, the devices may malfunction. When connecting devices operating at different supply voltages, such as in a dual-power-supply system, be aware that erroneous power-on and power-off sequences can result in device breakdown. For details of how to interface particular devices, consult the relevant technical datasheets and databooks. If you have any questions or doubts about interfacing, contact your nearest Toshiba office or distributor.

3.3.10 Decoupling

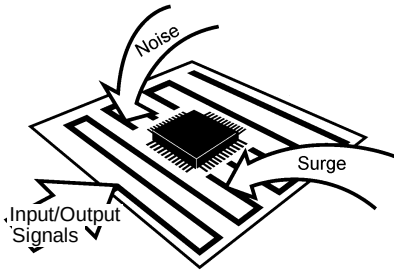
Spike currents generated during switching can cause Vcc (Vdd) and GND (Vss) voltage levels to fluctuate, causing ringing in the output waveform or a delay in response speed. (The power supply and GND wiring impedance is normally 50 Ω to 100 Ω.) For this reason, the impedance of power supply lines with respect to high frequencies must be kept low. This can be accomplished by using thick and short wiring for the Vcc (Vdd) and GND (Vss) lines and by installing decoupling capacitors (of approximately 0.01 μF to 1 μF capacitance) as high-frequency filters between Vcc (Vdd) and GND (Vss) at strategic locations on the printed circuit board.

For low-frequency filtering, it is a good idea to install a 10- to 100-μF capacitor on the printed circuit board (one capacitor will suffice). If the capacitance is excessively large, however, (e.g. several thousand μF) latch-up can be a problem. Be sure to choose an appropriate capacitance value.

An important point about wiring is that, in the case of high-speed logic ICs, noise is caused mainly by reflection and crosstalk, or by the power supply impedance. Reflections cause increased signal delay, ringing, overshoot and undershoot, thereby reducing the device's safety margins with respect to noise. To prevent reflections, reduce the wiring length by increasing the device mounting density so as to lower the inductance (L) and capacitance (C) in the wiring. Extreme care must be taken, however, when taking this corrective measure, since it tends to cause crosstalk between the wires. In practice, there must be a trade-off between these two factors.

3.3.11 External noise

Printed circuit boards with long I/O or signal pattern lines are vulnerable to induced noise or surges from outside sources. Consequently, malfunctions or breakdowns can result from overcurrent or overvoltage, depending on the types of device used. To protect against noise, lower the impedance of the pattern line or insert a noise-canceling circuit. Protective measures must also be taken against surges.



For details of the appropriate protective measures for a particular device, consult the relevant databook.

3.3.12 Electromagnetic interference

Widespread use of electrical and electronic equipment in recent years has brought with it radio and TV reception problems due to electromagnetic interference. To use the radio spectrum effectively and to maintain radio communications quality, each country has formulated regulations limiting the amount of electromagnetic interference which can be generated by individual products.

Electromagnetic interference includes conduction noise propagated through power supply and telephone lines, and noise from direct electromagnetic waves radiated by equipment. Different measurement methods and corrective measures are used to assess and counteract each specific type of noise.

Difficulties in controlling electromagnetic interference derive from the fact that there is no method available which allows designers to calculate, at the design stage, the strength of the electromagnetic waves which will emanate from each component in a piece of equipment. For this reason, it is only after the prototype equipment has been completed that the designer can take measurements using a dedicated instrument to determine the strength of electromagnetic interference waves. Yet it is possible during system design to incorporate some measures for the prevention of electromagnetic interference, which can facilitate taking corrective measures once

the design has been completed. These include installing shields and noise filters, and increasing the thickness of the power supply wiring patterns on the printed circuit board. One effective method, for example, is to devise several shielding options during design, and then select the most suitable shielding method based on the results of measurements taken after the prototype has been completed.

3.3.13 Peripheral circuits

In most cases semiconductor devices are used with peripheral circuits and components. The input and output signal voltages and currents in these circuits must be chosen to match the semiconductor device's specifications. The following factors must be taken into account.

(1) Inappropriate voltages or currents applied to a device's input pins may cause it to operate erratically. Some devices contain pull-up or pull-down resistors. When designing your system, remember to take the effect of this on the voltage and current levels into account.

(2) The output pins on a device have a predetermined external circuit drive capability. If this drive capability is greater than that required, either incorporate a compensating circuit into your design or carefully select suitable components for use in external circuits.

3.3.14 Safety standards

Each country has safety standards which must be observed. These safety standards include requirements for quality assurance systems and design of device insulation. Such requirements must be fully taken into account to ensure that your design conforms to the applicable safety standards.

3.3.15 Other precautions

(1) When designing a system, be sure to incorporate fail-safe and other appropriate measures according to the intended purpose of your system. Also, be sure to debug your system under actual board-mounted conditions.

(2) If a plastic-package device is placed in a strong electric field, surface leakage may occur due to the charge-up phenomenon, resulting in device malfunction. In such cases take appropriate measures to prevent this problem, for example by protecting the package surface with a conductive shield.

(3) With some microcomputers and MOS memory devices, caution is required when powering on or resetting the device. To ensure that your design does not violate device specifications, consult the relevant databook for each constituent device.

(4) Ensure that no conductive material or object (such as a metal pin) can drop onto and short the leads of a device mounted on a printed circuit board.

3.4 Inspection, Testing and Evaluation

3.4.1 Grounding

⚠CAUTION Ground all measuring instruments, jigs, tools and soldering irons to earth. Electrical leakage may cause a device to break down or may result in electric shock.

3.4.2 Inspection Sequence

⚠CAUTION

① Do not insert devices in the wrong orientation. Make sure that the positive and negative electrodes of the power supply are correctly connected. Otherwise, the rated maximum current or maximum power dissipation may be exceeded and the device may break down or undergo performance degradation, causing it to catch fire or explode, resulting in injury to the user.

② When conducting any kind of evaluation, inspection or testing using AC power with a peak voltage of 42.4 V or DC power exceeding 60 V, be sure to connect the electrodes or probes of the testing equipment to the device under test before powering it on. Connecting the electrodes or probes of testing equipment to a device while it is powered on may result in electric shock, causing injury.

(1) Apply voltage to the test jig only after inserting the device securely into it. When applying or removing power, observe the relevant precautions, if any.

(2) Make sure that the voltage applied to the device is off before removing the device from the test jig. Otherwise, the device may undergo performance degradation or be destroyed.

(3) Make sure that no surge voltages from the measuring equipment are applied to the device.

(4) The chips housed in tape carrier packages (TCPs) are bare chips and are therefore exposed. During inspection take care not to crack the chip or cause any flaws in it.
Electrical contact may also cause a chip to become faulty. Therefore make sure that nothing comes into electrical contact with the chip.

3.5 Mounting

There are essentially two main types of semiconductor device package: lead insertion and surface mount. During mounting on printed circuit boards, devices can become contaminated by flux or damaged by thermal stress from the soldering process. With surface-mount devices in particular, the most significant problem is thermal stress from solder reflow, when the entire package is subjected to heat. This section describes a recommended temperature profile for each mounting method, as well as general precautions which you should take when mounting devices on printed circuit boards. Note, however, that even for devices with the same package type, the appropriate mounting method varies according to the size of the chip and the size and shape of the lead frame. Therefore, please consult the relevant technical datasheet and databook.

3.5.1 Lead forming

⚠CAUTION

① Always wear protective glasses when cutting the leads of a device with clippers or a similar tool. If you do not, small bits of metal flying off the cut ends may damage your eyes.

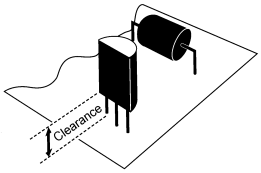
② Do not touch the tips of device leads. Because some types of device have leads with pointed tips, you may prick your finger.

Semiconductor devices must undergo a process in which the leads are cut and formed before the devices can be mounted on a printed circuit board. If undue stress is applied to the interior of a device during this process, mechanical breakdown or performance degradation can result. This is attributable primarily to differences between the stress on the device's external leads and the stress on the internal leads. If the relative difference is great enough, the device's internal leads, adhesive properties or sealant can be damaged. Observe these precautions during the lead-forming process (this does not apply to surface-mount devices):

(1) Lead insertion hole intervals on the printed circuit board should match the lead pitch of the device precisely.

(2) If lead insertion hole intervals on the printed circuit board do not precisely match the lead pitch of the device, do not attempt to forcibly insert devices by pressing on them or by pulling on their leads.

(3) For the minimum clearance specification between a device and a printed circuit board, refer to the relevant device's datasheet and databook. If necessary, achieve the required clearance by forming the device's leads appropriately. Do not use the spacers which are used to raise devices above the surface of the printed circuit board during soldering to achieve clearance. These spacers normally continue to expand due to heat, even after the solder has begun to solidify; this applies severe stress to the device.



(4) Observe the following precautions when forming the leads of a device prior to mounting.

- Use a tool or jig to secure the lead at its base (where the lead meets the device package) while bending so as to avoid mechanical stress to the device. Also avoid bending or stretching device leads repeatedly.
- Be careful not to damage the lead during lead forming.
- Follow any other precautions described in the individual datasheets and databooks for each device and package type.

3.5.2 Socket mounting

(1) When socket mounting devices on a printed circuit board, use sockets which match the inserted device's package.

(2) Use sockets whose contacts have the appropriate contact pressure. If the contact pressure is insufficient, the socket may not make a perfect contact when the device is repeatedly inserted and removed; if the pressure is excessively high, the device leads may be bent or damaged when they are inserted into or removed from the socket.

(3) When soldering sockets to the printed circuit board, use sockets whose construction prevents flux from penetrating into the contacts or which allows flux to be completely cleaned off.

(4) Make sure the coating agent applied to the printed circuit board for moisture-proofing purposes does not stick to the socket contacts.

(5) If the device leads are severely bent by a socket as it is inserted or removed and you wish to repair the leads so as to continue using the device, make sure that this lead correction is only performed once. Do not use devices whose leads have been corrected more than once.

(6) If the printed circuit board with the devices mounted on it will be subjected to vibration from external sources, use sockets which have a strong contact pressure so as to prevent the sockets and devices from vibrating relative to one another.

3.5.3 Soldering temperature profile

The soldering temperature and heating time vary from device to device. Therefore, when specifying the mounting conditions, refer to the individual datasheets and databooks for the devices used.

(1) Using a soldering iron

Complete soldering within ten seconds for lead temperatures of up to 260°C, or within three seconds for lead temperatures of up to 350°C.

(2) Using medium infrared ray reflow

- Heating top and bottom with long or medium infrared rays is recommended (see Figure 3).

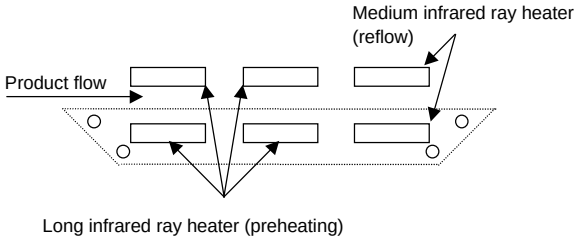


Figure 3 Heating top and bottom with long or medium infrared rays

- Complete the infrared ray reflow process within 30 seconds at a package surface temperature of between 210°C and 240°C.
- Refer to Figure 4 for an example of a good temperature profile for infrared or hot air reflow.

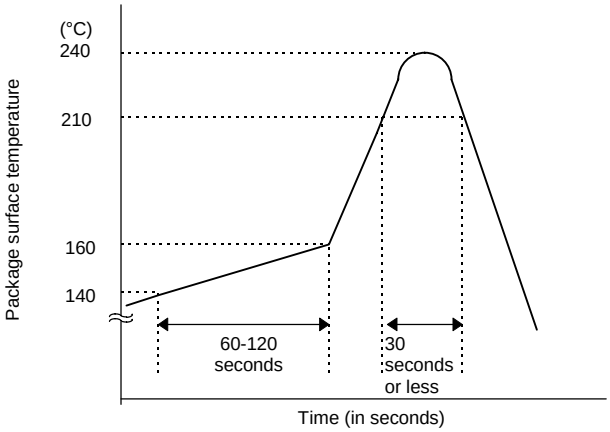


Figure 4 Sample temperature profile for infrared or hot air reflow

(3) Using hot air reflow

- Complete hot air reflow within 30 seconds at a package surface temperature of between 210°C and 240°C.
- For an example of a recommended temperature profile, refer to Figure 4 above.

(4) Using solder flow

- Apply preheating for 60 to 120 seconds at a temperature of 150°C.
- For lead insertion-type packages, complete solder flow within 10 seconds with the temperature at the stopper (or, if there is no stopper, at a location more than 1.5 mm from the body) which does not exceed 260°C.

- For surface-mount packages, complete soldering within 5 seconds at a temperature of 250°C or less in order to prevent thermal stress in the device.
- Figure 5 shows an example of a recommended temperature profile for surface-mount packages using solder flow.

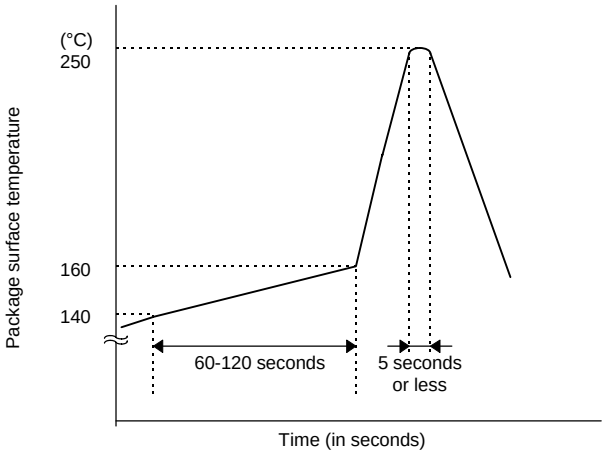


Figure 5 Sample temperature profile for solder flow

3.5.4 Flux cleaning and ultrasonic cleaning

(1) When cleaning circuit boards to remove flux, make sure that no residual reactive ions such as Na or Cl remain. Note that organic solvents react with water to generate hydrogen chloride and other corrosive gases which can degrade device performance.

(2) Washing devices with water will not cause any problems. However, make sure that no reactive ions such as sodium and chlorine are left as a residue. Also, be sure to dry devices sufficiently after washing.

(3) Do not rub device markings with a brush or with your hand during cleaning or while the devices are still wet from the cleaning agent. Doing so can rub off the markings.

(4) The dip cleaning, shower cleaning and steam cleaning processes all involve the chemical action of a solvent. Use only recommended solvents for these cleaning methods. When immersing devices in a solvent or steam bath, make sure that the temperature of the liquid is 50°C or below, and that the circuit board is removed from the bath within one minute.

(5) Ultrasonic cleaning should not be used with hermetically-sealed ceramic packages such as a leadless chip carrier (LCC), pin grid array (PGA) or charge-coupled device (CCD), because the bonding wires can become disconnected due to resonance during the cleaning process. Even if a device package allows ultrasonic cleaning, limit the duration of ultrasonic cleaning to as short a time as possible, since long hours of ultrasonic cleaning degrade the adhesion between the mold resin and the frame material. The following ultrasonic cleaning conditions are recommended:

Frequency: 27 kHz ~ 29 kHz

Ultrasonic output power: 300 W or less (0.25 W/cm^2 or less)

Cleaning time: 30 seconds or less

Suspend the circuit board in the solvent bath during ultrasonic cleaning in such a way that the ultrasonic vibrator does not come into direct contact with the circuit board or the device.

3.5.5 No cleaning

If analog devices or high-speed devices are used without being cleaned, flux residues may cause minute amounts of leakage between pins. Similarly, dew condensation, which occurs in environments containing residual chlorine when power to the device is on, may cause between-lead leakage or migration. Therefore, Toshiba recommends that these devices be cleaned. However, if the flux used contains only a small amount of halogen (0.05W% or less), the devices may be used without cleaning without any problems.

3.5.6 Mounting tape carrier packages (TCPs)

(1) When tape carrier packages (TCPs) are mounted, measures must be taken to prevent electrostatic breakdown of the devices.

(2) If devices are being picked up from tape, or outer lead bonding (OLB) mounting is being carried out, consult the manufacturer of the insertion machine which is being used, in order to establish the optimum mounting conditions in advance and to avoid any possible hazards.

(3) The base film, which is made of polyimide, is hard and thin. Be careful not to cut or scratch your hands or any objects while handling the tape.

(4) When punching tape, try not to scatter broken pieces of tape too much.

(5) Treat the extra film, reels and spacers left after punching as industrial waste, taking care not to destroy or pollute the environment.

(6) Chips housed in tape carrier packages (TCPs) are bare chips and therefore have their reverse side exposed. To ensure that the chip will not be cracked during mounting, ensure that no mechanical shock is applied to the reverse side of the chip. Electrical contact may also cause a chip to fail. Therefore, when mounting devices, make sure that nothing comes into electrical contact with the reverse side of the chip.
If your design requires connecting the reverse side of the chip to the circuit board, please consult Toshiba or a Toshiba distributor beforehand.

3.5.7 Mounting chips

Devices delivered in chip form tend to degrade or break under external forces much more easily than plastic-packaged devices. Therefore, caution is required when handling this type of device.

(1) Mount devices in a properly prepared environment so that chip surfaces will not be exposed to polluted ambient air or other polluted substances.

(2) When handling chips, be careful not to expose them to static electricity.
In particular, measures must be taken to prevent static damage during the mounting of chips. With this in mind, Toshiba recommend mounting all peripheral parts first and then mounting chips last (after all other components have been mounted).

(3) Make sure that PCBs (or any other kind of circuit board) on which chips are being mounted do not have any chemical residues on them (such as the chemicals which were used for etching the PCBs).

(4) When mounting chips on a board, use the method of assembly that is most suitable for maintaining the appropriate electrical, thermal and mechanical properties of the semiconductor devices used.

* For details of devices in chip form, refer to the relevant device's individual datasheets.

3.5.8 Circuit board coating

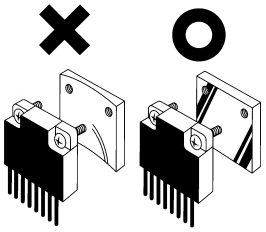
When devices are to be used in equipment requiring a high degree of reliability or in extreme environments (where moisture, corrosive gas or dust is present), circuit boards may be coated for protection. However, before doing so, you must carefully consider the possible stress and contamination effects that may result and then choose the coating resin which results in the minimum level of stress to the device.

3.5.9 Heat sinks

(1) When attaching a heat sink to a device, be careful not to apply excessive force to the device in the process.

(2) When attaching a device to a heat sink by fixing it at two or more locations, evenly tighten all the screws in stages (i.e. do not fully tighten one screw while the rest are still only loosely tightened). Finally, fully tighten all the screws up to the specified torque.

(3) Drill holes for screws in the heat sink exactly as specified. Smooth the surface by removing burrs and protrusions or indentations which might interfere with the installation of any part of the device.



(4) A coating of silicone compound can be applied between the heat sink and the device to improve heat conductivity. Be sure to apply the coating thinly and evenly; do not use too much. Also, be sure to use a non-volatile compound, as volatile compounds can crack after a time, causing the heat radiation properties of the heat sink to deteriorate.

(5) If the device is housed in a plastic package, use caution when selecting the type of silicone compound to be applied between the heat sink and the device. With some types, the base oil separates and penetrates the plastic package, significantly reducing the useful life of the device.
Two recommended silicone compounds in which base oil separation is not a problem are YG6260 from Toshiba Silicone.

(6) Heat-sink-equipped devices can become very hot during operation. Do not touch them, or you may sustain a burn.

3.5.10 Tightening torque

(1) Make sure the screws are tightened with fastening torques not exceeding the torque values stipulated in individual datasheets and databooks for the devices used.

(2) Do not allow a power screwdriver (electrical or air-driven) to touch devices.

3.5.11 Repeated device mounting and usage

Do not remount or re-use devices which fall into the categories listed below; these devices may cause significant problems relating to performance and reliability.

(1) Devices which have been removed from the board after soldering

(2) Devices which have been inserted in the wrong orientation or which have had reverse current applied

(3) Devices which have undergone lead forming more than once

3.6 Protecting Devices in the Field

3.6.1 Temperature

Semiconductor devices are generally more sensitive to temperature than are other electronic components. The various electrical characteristics of a semiconductor device are dependent on the ambient temperature at which the device is used. It is therefore necessary to understand the temperature characteristics of a device and to incorporate device derating into circuit design. Note also that if a device is used above its maximum temperature rating, device deterioration is more rapid and it will reach the end of its usable life sooner than expected.

3.6.2 Humidity

Resin-molded devices are sometimes improperly sealed. When these devices are used for an extended period of time in a high-humidity environment, moisture can penetrate into the device and cause chip degradation or malfunction. Furthermore, when devices are mounted on a regular printed circuit board, the impedance between wiring components can decrease under high-humidity conditions. In systems which require a high signal-source impedance, circuit board leakage or leakage between device lead pins can cause malfunctions. The application of a moisture-proof treatment to the device surface should be considered in this case. On the other hand, operation under low-humidity conditions can damage a device due to the occurrence of electrostatic discharge. Unless damp-proofing measures have been specifically taken, use devices only in environments with appropriate ambient moisture levels (i.e. within a relative humidity range of 40% to 60%).

3.6.3 Corrosive gases

Corrosive gases can cause chemical reactions in devices, degrading device characteristics. For example, sulphur-bearing corrosive gases emanating from rubber placed near a device (accompanied by condensation under high-humidity conditions) can corrode a device's leads. The resulting chemical reaction between leads forms foreign particles which can cause electrical leakage.

3.6.4 Radioactive and cosmic rays

Most industrial and consumer semiconductor devices are not designed with protection against radioactive and cosmic rays. Devices used in aerospace equipment or in radioactive environments must therefore be shielded.

3.6.5 Strong electrical and magnetic fields

Devices exposed to strong magnetic fields can undergo a polarization phenomenon in their plastic material, or within the chip, which gives rise to abnormal symptoms such as impedance changes or increased leakage current. Failures have been reported in LSIs mounted near malfunctioning deflection yokes in TV sets. In such cases the device's installation location must be changed or the device must be shielded against the electrical or magnetic field. Shielding against magnetism is especially necessary for devices used in an alternating magnetic field because of the electromotive forces generated in this type of environment.

3.6.6 Interference from light (ultraviolet rays, sunlight, fluorescent lamps and incandescent lamps)

Light striking a semiconductor device generates electromotive force due to photoelectric effects. In some cases the device can malfunction. This is especially true for devices in which the internal chip is exposed. When designing circuits, make sure that devices are protected against incident light from external sources. This problem is not limited to optical semiconductors and EPROMs. All types of device can be affected by light.

3.6.7 Dust and oil

Just like corrosive gases, dust and oil can cause chemical reactions in devices, which will adversely affect a device's electrical characteristics. To avoid this problem, do not use devices in dusty or oily environments. This is especially important for optical devices because dust and oil can affect a device's optical characteristics as well as its physical integrity and the electrical performance factors mentioned above.

3.6.8 Fire

Semiconductor devices are combustible; they can emit smoke and catch fire if heated sufficiently. When this happens, some devices may generate poisonous gases. Devices should therefore never be used in close proximity to an open flame or a heat-generating body, or near flammable or combustible materials.

3.7 Disposal of devices and packing materials

When discarding unused devices and packing materials, follow all procedures specified by local regulations in order to protect the environment against contamination.

4. Precautions and Usage Considerations

This section describes matters specific to each product group which need to be taken into consideration when using devices. If the same item is described in Sections 3 and 4, the description in Section 4 takes precedence.

4.1 Microcontrollers

4.1.1 Design

(1) Using resonators which are not specifically recommended for use

Resonators recommended for use with Toshiba products in microcontroller oscillator applications are listed in Toshiba databooks along with information about oscillation conditions. If you use a resonator not included in this list, please consult Toshiba or the resonator manufacturer concerning the suitability of the device for your application.

(2) Undefined functions

In some microcontrollers certain instruction code values do not constitute valid processor instructions. Also, it is possible that the values of bits in registers will become undefined. Take care in your applications not to use invalid instructions or to let register bit values become undefined.

Architecture

Chapter 1 Introduction

1.1 Features

The TX39 Processor Core is a high-performance 32-bit microprocessor core developed by Toshiba based on the R3000A RISC (Reduced Instruction Set Computer) microprocessor. The R3000A was developed by MIPS Technologies, Inc.

Toshiba develops ASSPs (Application Specific Standard Products) using the TX39 Processor Core and provides the TX39 as a processor core in Embedded Array or Cell-based ICs. The low power consumption and high cost-performance ratio of this processor make it especially well-suited to embedded control applications in products such as PDAs (Personal Digital Assistants) and game equipment.

1.1.1 High-performance RISC techniques

- R3000A architecture
  - R3000A upward compatible instruction set (excluding TLB (translation lookaside buffer) instructions and some coprocessor instructions)
  - Five-stage pipeline
- Built-in cache memory
  - Separate instruction and data caches
  - Data cache snoop function: Invalidatation of data in the data cache to maintain cache memory and main memory consistency on DMA transfer cycles
- Nonblocking load
  - Execute the following instruction regardless of a cache miss caused by a preceding load instruction
- DSP function
  - Multiply/Add (32-bit x 32-bit + 64-bit) in one clock cycle.

1.1.2 Functions for embedded applications

- Small code size
  - Branch Likely instruction: The branch delay slot accepts an instruction to be executed at the branch target
  - Hardware Interlock: Stall the pipeline at the load delay slot when the instruction in the slot depends on the data to be loaded
- Real-time performance
  - Cache Lock Function: Lock one set of the two-way set associative cache memory to keep data in cache memory
- Debug support
  - Breakpoint
  - Single step execution
- Real-time debug system interface

1.1.3 Low power consumption

- Power Down mode
  - Prepare for Reduced Frequency mode: Control the clock frequency of the TX39 Processor Core with a clock generator
  - Halt and Doze mode: Stop TX39 Processor Core operations
- Clock can be stopped
  - Clock signal can be stopped at high state

1.1.4 Development environment for embedded arrays and cell-based ICs

- Compact core
- Easy-to-design peripheral circuits
  - Single direction separate bus: Bus configuration suitable for core
  - Built-in cache memory: No need to consider cache operation timing
- ASIC Process
- Sufficient Development Environment

1.2 Notation Used in This Manual

Mathematical notation

- Hexadecimal numbers are expressed as follows (example shown for decimal number 42)

  0x2A
- A K(kilo)byte is 2^{10} = 1,024 bytes, a M(mega)byte is 2^{20} = 1,024 x 1,024 = 1,048,576 bytes, and a G(giga)byte is 2^{30} = 1,024 x 1,024 x 1,024 = 1,073,741,824 bytes.

Data notation

- Byte: 8 bits
- Halfword: 2 contiguous bytes (16 bits)
- Word: 4 contiguous bytes (32 bits)
- Doubleword: 8 contiguous bytes (64 bits)

Signal notation

- Low active signals are indicated by an asterisk (*) at the end of the signal name (e.g.: RESET*).
- Changing a signal to active level is to “assert” a signal, while changing it to a non-active level is to “de-assert” the signal.

Chapter 2 Architecture

2.1 Overview

A block diagram of the TX39 Processor Core is shown in Figure 2-1. It includes the CPU core, an instruction cache and a data cache. You can select an optimum data and instruction cache configuration for your system from among a variety of possible configurations.

The CPU Core comprises the following blocks:

- CPU registers: General-purpose register, HI/LO register and program counter (PC).
- CP0 registers: Registers for system control coprocessor (CP0) functions.
- ALU/Shifter: Computational unit.
- MAC: Computational unit for multiply/add.
- Bus interface unit: Control bus interface between CPU core and external circuit.
- Memory management unit: Direct segment mapping memory management unit.

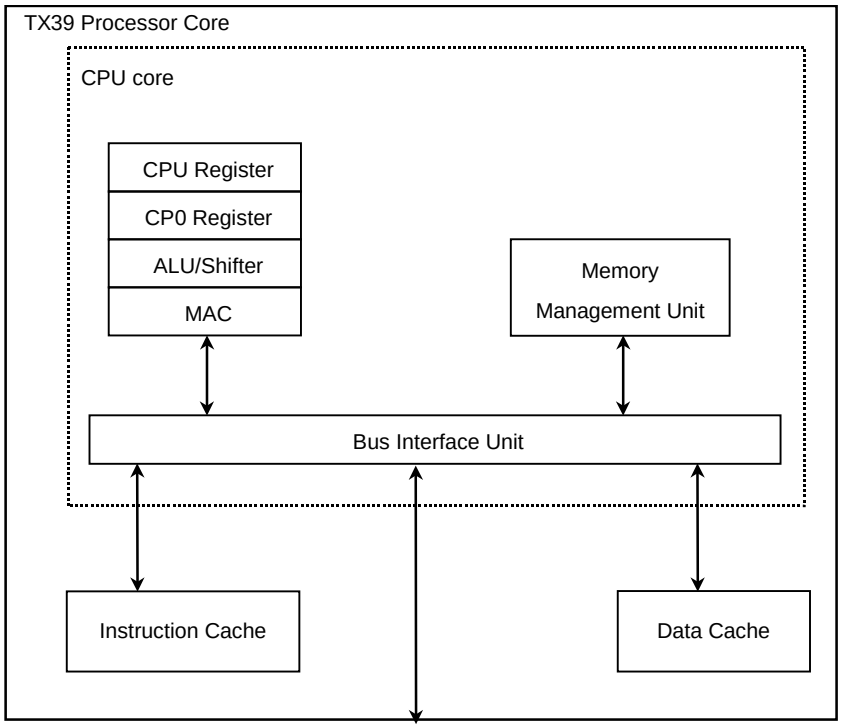


Figure 2-1. Block Diagram of the TX39 Processor Core

2.2 Registers

2.2.1 CPU registers

The TX39 Processor Core has the following 32-bit registers.

- Thirty-two general-purpose registers
- A program counter (PC)
- HI/LO registers for storing the result of multiply and divide operations

The configuration of the registers is shown in Figure 2-2.

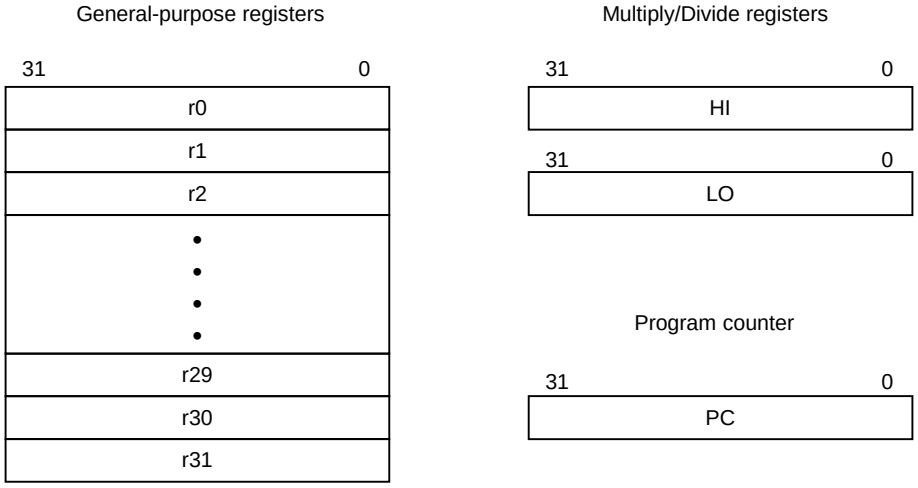


Figure 2-2. TX39 Processor Core registers

The r0 and r31 registers have special functions.

- Register r0 always contains the value 0. It can be a target register of an instruction whose operation result is not needed. Or, it can be a source register of an instruction that requires a value of 0.
- Register r31 is the link register for the Jump And Link instruction. The address of the instruction after the delay slot is placed in r31.

The TX39 Processor Core has the following three special registers that are used or modified implicitly by certain instructions.

PC: Program counter

HI: High word of the multiply/divide registers

LO: Low word of the multiply/divide registers

The multiply/divide registers (HI, LO) store the double-word (64-bit) result of integer multiply operations. In the case of integer divide operations, the quotient is stored in LO and the remainder in HI.

2.2.2 System control coprocessor (CP0) registers

The TX39 Processor Core can be connected to as many as three coprocessors, referred to as CP1, CP2 and CP3. The TX39 also has built-in system control coprocessor (CP0) functions for exception handling and for configuring the system. Figure 2-3 shows the functional breakdown of the CP0 registers.

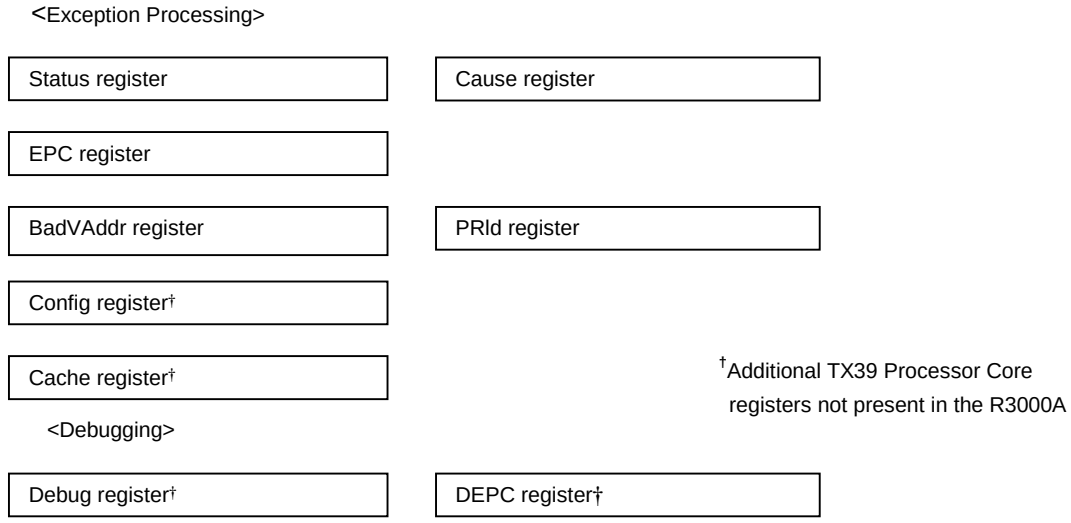


Figure 2-3. CP0 registers

Table 2-1 lists the CP0 registers built into the TX39 Processor Core. Some of these registers are reserved for use by an external memory management unit.

Table 2-1. List of system control coprocessor (CP0) registers

No	Mnemonic	Description
0	-	(reserved) †
1	-	(reserved) †
2	-	(reserved) †
3	Config††	Hardware configuration
4	-	(reserved) †
5	-	(reserved) †
6	-	(reserved) †
7	Cache††	Cache lock function
8	BadVAddr	Last virtual address triggering error
9	-	(reserved) †
10	-	(reserved) †
11	-	(reserved) †
12	Status	Information on mode, interrupt enabled, diagnostic status
13	Cause	Indicates nature of last exception
14	EPC	Exception program counter
15	PRId	Processor revision ID
16	Debug†††	Debug exception control
17	DEPC†††	Program counter for debug exception
18 | 31	-	(reserved) †

† Reserved for external memory management unit, when direct segment mapping MMU is not used.

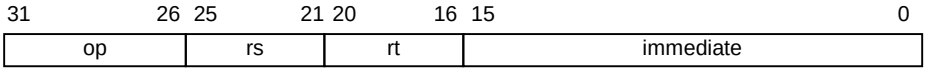
†† Additional TX39 Processor Core register not present in R3000A.

††† Additional TX39 Processor Core Debug register not present in R3000A.

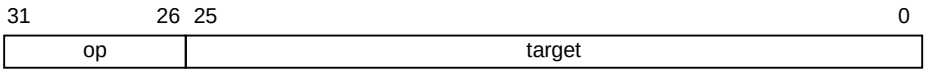
2.3 Instruction Set Overview

All TX39 Processor Core instructions are 32 bits in length. There are three instruction formats: immediate (I-type), jump (J-type) and register (R-type), as shown in Figure 2-4. Having just three instruction formats simplifies instruction decoding. If more complex functions or addressing modes are required, they can be produced with the compiler using combinations of the instructions.

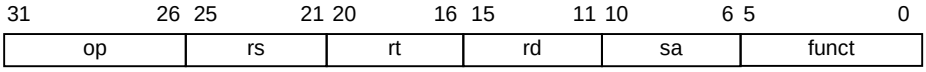
I-type (Immediate)



J-type (Jump)



R-type (Register)



op	Operation code (6 bits)
rs	Source register (5 bits)
rt	Target (source or destination) register, or branch condition (5 bits)
rd	Destination register (5 bits)
immediate	Immediate, branch displacement, address displacement (16 bits)
target	Branch target address (26 bits)
sa	Shift amount (5 bits)
funct	Function (6 bits)

Figure 2-4. Instruction formats and subfield mnemonics

The instruction set is classified as follows.

(1) Load/store

These instructions transfer data between memory and general registers. All instructions in this group are I-type. "Base register + 16 bit signed immediate offset" is the only supported addressing mode.

(2) Computational

These instructions perform arithmetic, logical and shift operations on register values. The format can be R-type (when both operands and the result are register values) or I-type (when one operand is 16-bit immediate data).

(3) Jump/branch

These instructions change the program flow. A jump is always made to a 32 bit address contained in a register (R-type format), or to a paged absolute address constructed by combining a 26-bit target address with the upper 4 bits of the program counter (J-type format). In a branch instruction, the target address is made up of the program counter value plus a 16 bit offset.

(4) Coprocessor

These instructions execute coprocessor operations. Each coprocessor has its own format for computational instructions.

Note: Coprocessor load instruction LWCz and coprocessor store instruction SWCz are not supported by the TX39 Processor Core. An attempt to execute either of these instructions will trigger a Reserved Instruction exception.

(5) Coprocessor 0

These instructions are used for operations with system control coprocessor (CP0) registers, processor memory management and exception handling.

Note: TLB (Translation Lookaside Buffer) instructions (TLBR, TLBWJ, TLBWR and TLBP) are not supported by the TX39 Processor Core. These instructions will be treated by the TX39 as NOP(no operation).

(6) Special

These instructions support system calls and breakpoint functions. The format is always R-type.

The instruction set supported by all MIPS R-Series processors is listed in Table 2-2. Table 2-3 shows extended instructions supported by the TX39 Processor Core, and Table 2-4 lists coprocessor 0 (CP0) instructions.

Table 2-5 shows R3000A instructions not supported by the TX39 Processor Core.

Table 2-2. Instructions supported by MIPS R-Series processors (ISA)

Instruction	Description
Load/Store Instructions	
LB	Load Byte
LBU	Load Byte Unsigned
LH	Load Halfword
LHU	Load Halfword Unsigned
LW	Load Word
LWL	Load Word Left
LWR	Load Word Right
SB	Store Byte
SH	Store Halfword
SW	Store Word
SWL	Store Word Left
SWR	Store Word Right
Computational Instructions	
(ALU Immediate)	
ADDI	Add Immediate
ADDIU	Add Immediate Unsigned
SLTI	Set on Less Than Immediate
SLTIU	Set on Less Than Immediate Unsigned
ANDI	AND Immediate
ORI	OR Immediate
XORI	XOR Immediate
LUI	Load Upper Immediate
(ALU 3-operand, register type)	
ADD	Add
ADDU	Add Unsigned
SUB	Subtract
SUBU	Subtract Unsigned
SLT	Set on Less Than
SLTU	Set on Less Than Unsigned
AND	AND
OR	OR
XOR	XOR
NOR	NOR
(Shift)	
SLL	Shift Left Logical
SRL	Shift Right Logical
SRA	Shift Right Arithmetic
SLLV	Shift Left Logical Variable
SRLV	Shift Right Logical Variable
SRAV	Shift Right Arithmetic Variable
(Multiply/Divide)	
MULT	Multiply
MULTU	Multiply Unsigned
DIV	Divide
DIVU	Divide Unsigned
MFHI	Move from HI
MTHI	Move to HI
MFLO	Move from LO
MTLO	Move to LO

Table 2-2(cont.). Instructions supported by MIPS R-Series processors (ISA)

Instruction	Description
Jump/Branch Instructions	
J	Jump
JAL	Jump And Link
JR	Jump Register
JALR	Jump And Link Register
BEQ	Branch on Equal
BNE	Branch on Not Equal
BLEZ	Branch on Less than or Equal to Zero
BGTZ	Branch on Greater Than Zero
BLTZ	Branch on Less Than Zero
BGEZ	Branch on Greater than or Equal to Zero
BLTZAL	Branch on Less Than Zero And Link
BGEZAL	Branch on Greater than or Equal to Zero And Link
Coprocessor Instructions	
MTCz	Move to Coprocessor z
MFCz	Move from Coprocessor z
CTCz	Move Control Word to Coprocessor z
CFCz	Move control Word from Coprocessor z
COPz	Coprocessor Operation z
BCzT	Branch on Coprocessor z True
BCzF	Branch on Coprocessor z False
Special Instructions	
SYSCALL	System Call
BREAK	Breakpoint

Table 2-3. TX39 extended instructions

Instruction	Description
Load/Store Instruction	
SYNC	Sync
Computational Instructions	
MULT	Multiply (3-operand instruction)
MULTU	Multiply Unsigned (3-operand instruction)
MADD	Multiply/ADD
MADDU	Multiply/ADD Unsigned
Jump/Branch Instructions	
BEQL	Branch on Equal Likely
BNEL	Branch on Not Equal Likely
BLEZL	Branch on Less than or Equal to Zero Likely
BGTZL	Branch on Greater Than Zero Likely
BLTZL	Branch on Less Than Zero Likely
BGEZL	Branch on Greater than or Equal to Zero Likely
BLTZALL	Branch on Less Than Zero And Link Likely
BGEZALL	Branch on Greater than or Equal to Zero And Link Likely
Coprocessor Instructions	
BCzTL	Branch on Coprocessor z True Likely
BCzFL	Branch on Coprocessor z False Likely
Special Instruction	
SDBBP	Software Debug Breakpoint

Table 2-4. CP0 instructions

Instruction	Description
CP0 Instructions	
MTC0	Move to CP0
MFC0	Move from CP0
RFE	Restore from Exception
DERET	Debug Exception Return
CACHE	Cache Operation

Table 2-5. R3000A instructions not supported by the TX39

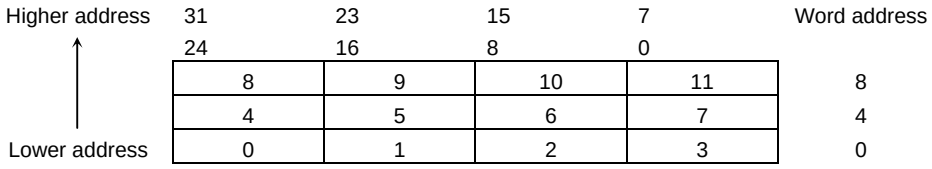
Instruction	Description	Operation
Coprocessor Instructions		
LWCz	Load Word from Coprocessor	Reserved Instruction Exception
SWCz	Store Word to Coprocessor	Reserved Instruction Exception
CP0 Instructions		
TLBR	Read indexed TLB entry	no operation(nop)
TLBWJ	Write indexed TLB entry	no operation(nop)
TLBWR	Write Random TLB entry	no operation(nop)
TLBP	Probe TLB for matching entry	no operation(nop)

2.4 Data Formats and Addressing

This section explains how data is organized in TX39 registers and memory.

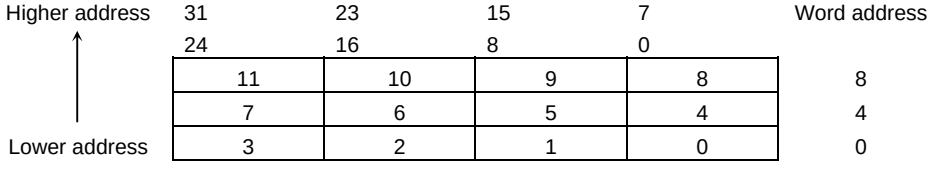
The TX39 uses the following data formats: 64-bit doubleword, 32-bit word, 16-bit halfword and 8-bit byte. The byte order can be set to either big endian or little endian.

Figure 2-5 shows how bytes are ordered in words, and how words are ordered in multiple words, for both the big-endian and little-endian formats.



- Byte 0 is the most significant byte (bit 31-24).
- A word is addressed beginning with the most significant byte.

(a) Big endian



- Byte 0 is the least significant byte (bit 7-0).
- A word is addressed beginning with the most significant byte.

(b) Little endian

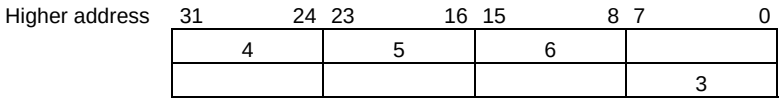
Figure 2-5. Big endian and little endian formats

In this document (bit 0 is always the rightmost bit).

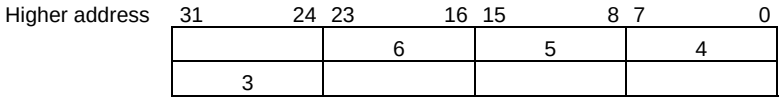
Byte addressing is used with the TX39 Processor Core, but there are alignment restrictions for halfword and word access. Halfword access is aligned on an even byte boundary (0, 2, 4...) and word access on a byte boundary divisible by 4 (0, 4, 8...) .

The address of multiple-byte data, as shown in Figure 2-5 above, begins at the most significant byte for the big endian format and at the least significant byte for the little endian format.

There are special instructions (LWL, LWR, SWL, SWR) for accessing words not aligned on a word boundary. They are used in pairs for addressing misaligned words, but involve an extra instruction cycle which is wasted if used with properly aligned words. Figure 2-6 shows the byte arrangement when a misaligned word is addressed at byte address 3 for the big and little endian formats.



(a) Big endian



(a) Little endian

Figure 2-6. Byte addresses of a misaligned word

2.5 Pipeline Processing Overview

The TX39 Processor Core executes instructions in five pipeline stages (F: instruction fetch; D: decode; E: execute; M: memory access; W: register write-back). Each pipeline stage is executed in one clock cycle. When the pipeline is fully utilized, five instructions are executed at the same time resulting in an instruction execution rate of one instruction per cycle.

With the TX39 Processor Core an instruction that immediately follows a load instruction can use the result of that load instruction. Execution of the following instruction is delayed by hardware interlock until the result of the load instruction becomes available. The instruction position immediately following the load instruction is called the "load delay slot."

In the case of branch instructions, a one-cycle delay is required to generate the branch target address. This delayed cycle is referred to as the "branch delay slot." An instruction placed immediately after a branch instruction (in the branch delay slot) can be executed prior to the branch while the branch target address is being generated.

The TX39 Processor Core provides a Branch Likely instruction whereby an instruction to be executed at the branch target can be placed in the delay slot of the Branch Likely instruction and executed only if the conditions of the branch instruction are met. If the conditions are not met, and the branch is not taken, the instruction in the delay slot is treated as a NOP. This makes it possible to place an instruction that would normally be executed at the branch target into the delay slot for quick execution (if the conditions of the branch are met).

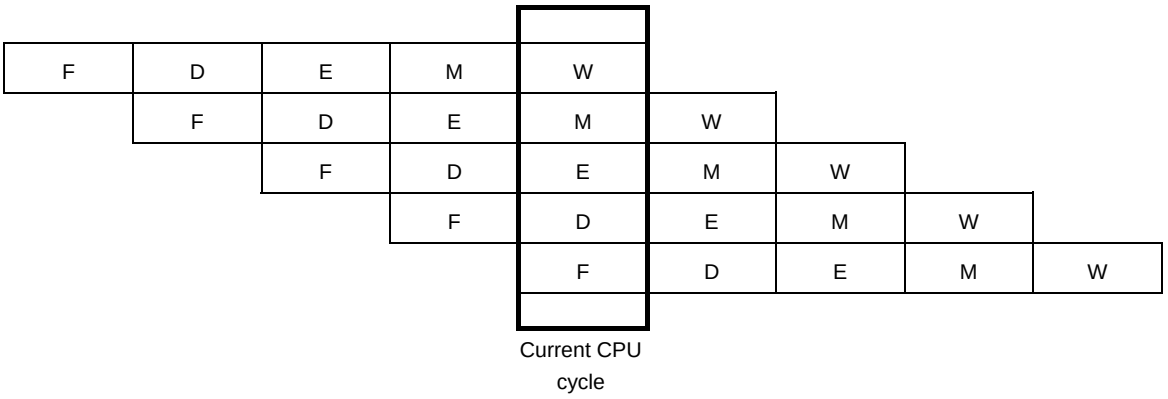


Figure 2-7. Pipeline stages for execution of TX39 Processor Core instructions

2.6 Memory Management Unit (MMU)

2.6.1 TX39 Processor Core operating modes

The TX39 Processor Core has two operating modes, user mode and kernel mode. Normally the processor operates in user mode. It switches to kernel mode if an exception is detected. Once in kernel mode, it remains there until an RFE (Restore From Exception) instruction is executed.

(1) User mode

User mode makes available one of the two 2 Gbyte virtual address spaces (kuseg). In this mode the most significant bit of each kuseg address in the memory map is 0. Attempting to access an address whose MSB is 1 while in user mode returns an Address Error exception.

(2) Kernel mode

Kernel mode makes available a second 2 Gbyte virtual address space (kseg), in addition to the kuseg accessible in user mode. The MSB of each kseg address in the memory map is 1.

2.6.2 Direct segment mapping

The TX39 Processor Core includes a direct segment mapping MMU. The following virtual address spaces are available depending on the processor mode (Figure 2-8 shows the address mapping).

(1) User mode

One 2 Gbyte virtual address space (kuseg) is available. Virtual addresses from 0x0000 0000 to 0x7FFF FFFF are translated to physical addresses 0x4000 0000 to 0xBFFF FFFF, respectively.

(2) Kernel mode

The kernel mode address space is treated as four virtual address segments. One of these is the same as the kuseg space in user mode; the remaining three are the kernel segments kseg0, kseg1 and kseg2.

(a) kuseg

This is the same as the virtual address space available in user mode. Address translation is also the same as in user mode. The upper 16 Mbytes of kuseg is reserved for on-chip resources and is not cacheable.

(b) kseg0

This is a 512 Mbyte segment spanning virtual addresses 0x8000 0000 to 0x9FFF FFFF. Fixed mapping of this segment is made to physical addresses 0x0000 0000 to 0x1FFF FFFF, respectively. This area is cacheable.

(c) kseg1

This is a 512 Mbyte segment from virtual address 0xA000 0000 to 0xBFFF FFFF. Fixed mapping of this segment is made to physical address 0x0000 0000 to 0x1FFF FFFF, respectively. Unlike kseg0, this area is not cacheable.

(d) kseg2

This is a 1 Gbyte linear address space from virtual addresses 0xC000 0000 to 0xFFFF FFFF. The upper 16 Mbytes of kseg2 are reserved for on-chip resources and are not cacheable. Of this reserved area, 0xFF20 0000 to 0xFF3F FFFF is a 2 Mbyte reserved area intended for use as a debugging monitor area and for testing.

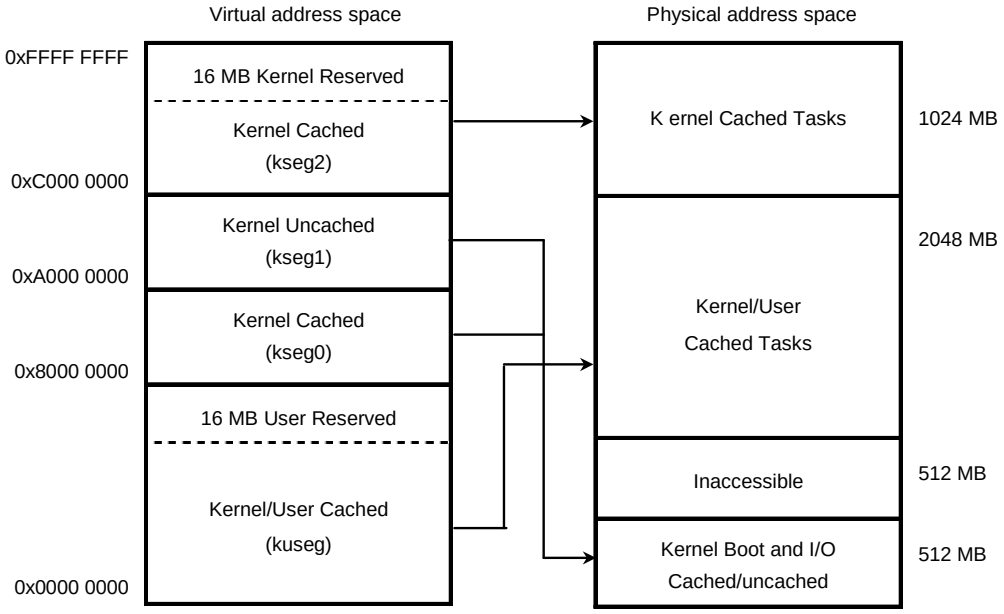


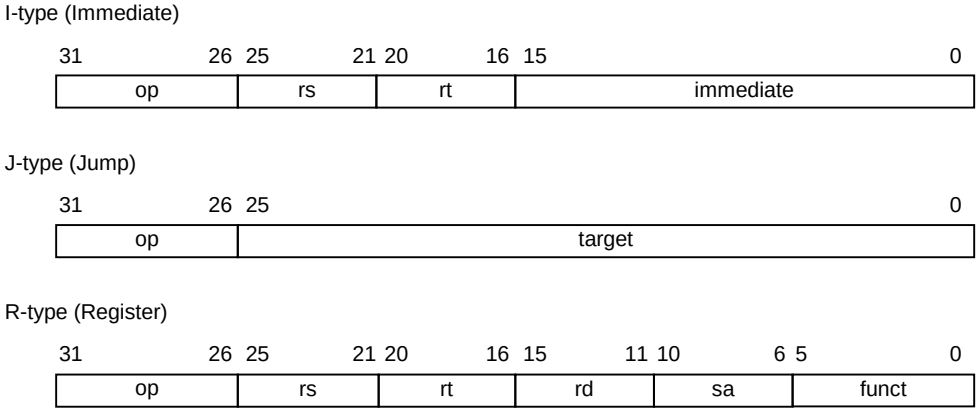
Figure 2-8. Address mapping

Chapter 3 Instruction Set Overview

This chapter summarizes each of the TX39 Processor Core instruction types in table format and explains each instruction briefly. Details of individual instructions are given in Appendix A.

3.1 Instruction Formats

Each of the TX39 Processor Core instructions is aligned on a word boundary and has a 32-bit (single-word) length. There are only three instruction formats, as shown in Figure 3-1. As a result, instruction decoding is simplified. Less frequently used and more complex functions or addressing modes can be realized by combining these instructions.



op	Operation code (6 bits)
rs	Source register (5 bits)
rt	Target (source or destination) register, or branch condition (5 bits)
rd	Destination register (5 bits)
immediate	Immediate, branch displacement, address displacement (16 bits)
target	Branch target address (26 bits)
sa	Shift amount (5 bits)
funct	Function (6 bits)

Figure 3-1. Instruction Formats and subfield mnemonics

3.2 Instruction Notation

All variable subfields in the instruction formats used here are written in lower-case letters (rs, rt, immediate, etc.). Also, an alias is sometimes used for a subfield name, for the sake of clarity. For example, rs in a load/store instruction may be referred to as “base”. When such an alias refers to a subfield that can take a variable value, it is likewise written in lower-case letters.

With specific instructions, the instruction subfields “op” and “funct” have fixed 6-bit values. These values are thus written as equates in upper-case letters. In the Load Byte instruction, for example, op = LB; and in the ADD instruction, op = SPECIAL and function = ADD.

3.3 Load and Store Instructions

Load and Store instructions move data between memory and general registers and are all I-type instructions. The only directly supported addressing mode is "base register plus 16-bit signed immediate offset."

With the TX39 Processor Core, the result of a load instruction can be used by the immediately following instruction. Execution of the following instruction is delayed by hardware interlock until the load result becomes available. The instruction position immediately following the load instruction is referred to as the "load delay slot". In the case of the LWL (Load Word Left) and LWR (Load Word Right) instructions, however, it is possible to use the destination register of an immediately preceding load instruction as the target register of the LWL or LWR instruction.

The access type, which indicates the size of data to be loaded or stored, is determined by the operation code (op) of the load or store instruction. The target address of a load or store is always the smallest byte address of the target data byte string, regardless of the access type or endian. This address is the most significant byte for the big endian format, and the least significant byte for the little endian format.

The position of the accessed data is determined by the access type and the two low-order address bits, as shown in Table 3-1.

Designating a combination other than those shown in Table 3-1 results in an Address Error exception.

Table 3-1. Byte specifications for load and store instructions

Access Type	Low order address bits		Accessed Bytes							
			Big Endian				Little Endian			
	1	0	31			0	31			0
word	0	0	0	1	2	3	3	2	1	0
triple-byte	0	0	0	1	2			2	1	0
	0	1		1	2	3	3	2	1	
halfword	0	0	0	1					1	0
	1	0			2	3	3	2		
byte	0	0	0							0
	0	1		1					1	
	1	0			2			2		
	1	1				3	3			

Table 3-2. Load/store instructions

Instruction	Format and Description: op | base | rt | offset
Load Byte	LB rt, offset (base) Generate the address by sign-extending a 32-bit offset and adding it to the contents of register base. Sign-extend the contents of the addressed byte and load into register rt.
Load Byte Unsigned	LBU rt, offset (base) Generate the address by sign-extending a 32-bit offset and adding it to the contents of register base. Zero-extend the contents of the addressed byte and load into register rt.
Load Halfword	LH rt, offset (base) Generate the address by sign-extending a 32-bit offset and adding it to the contents of register base. Sign-extend the contents of the addressed halfword and load into register rt.
Load Halfword Unsigned	LHU rt, offset (base) Generate the address by sign-extending a 32-bit offset and adding it to the contents of register base. Zero-extend the contents of the addressed halfword and load into register rt.
Load Word	LW rt, offset (base) Generate the address by sign-extending a 32-bit offset and adding it to the contents of register base. Load the contents of the addressed word into register rt.
Load Word Left	LWL rt, offset (base) Generate the address by sign-extending a 32-bit offset and adding it to the contents of register base. This instruction is paired with LWR and used to load word data not aligned with a word boundary. The LWL instruction loads the left part of the word, and LWR loads the right part. LWL shifts the addressed byte to the left, so that it will form the left side of the word, merges it with the contents of register rt and loads the result into rt.
Load Word Right	LWR rt, offset (base) Generate the address by sign-extending a 32-bit offset and adding it to the contents of register base. LWR shifts the addressed byte to the right, so that it will form the right side of the word, merges it with the contents of register rt and loads the result into rt.
Store Byte	SB rt, offset (base) Generate the address by sign-extending a 32-bit offset and adding it to the contents of register base. Store the contents of the least significant byte of register rt at the addressed byte.
Store Halfword	SH rt, offset (base) Generate the address by sign-extending a 32-bit offset and adding it to the contents of register base. Store the contents of the least significant halfword of register rt at the addressed byte.
Store Word	SW rt, offset (base) Generate the address by sign-extending a 32-bit offset and adding it to the contents of register base. Store the contents of the least significant word of register rt at the addressed byte.
Store Word Left	SWL rt, offset (base) Generate the address by sign-extending a 32-bit offset and adding it to the contents of register base. This instruction is used together with SWR to store the contents of a register into four consecutive bytes of memory when the bytes cross a word boundary. The SWL instruction stores the left part of the register, and SWR stores the right part. SWL shifts the contents of register rt to the right so that the leftmost byte of the word aligns with the addressed byte. It then stores the bytes containing the original data in the corresponding bytes at the addressed byte.
Store Word Right	SWR rt, offset (base) Generate the address by sign-extending a 32-bit offset and adding it to the contents of register base. SWR shifts the contents of register rt to the left so that the rightmost byte of the word aligns with the addressed byte. It then stores the bytes containing the original data in the corresponding bytes at the addressed byte.

Table 3-3. Load/store instructions (R3000A extended set)

Instruction	Format and Description: op | 0 | func
SYNC	SYNC Interlock the pipeline while a load or store instruction is executing, until execution is completed.

3.4 Computational Instructions

Computational instructions perform arithmetic, logical or shift operations on values in registers. The instruction format can be R-type or I-type. With R-type instructions, the two operands and the result are register values. With I-type instructions, one of the operands is 16-bit immediate data. Computational instructions can be classified as follows.

- ALU immediate (Table 3-4)
- Three-operand register-type (Table 3-5)
- Shift (Table 3-6)
- Multiply/Divide (Table 3-7,Table3-8)

Table 3-4. ALU immediate instructions

Instruction	Format and Description: op | rs | rt | immediate
Add Immediate	ADDI rt, rs, immediate Add 32-bit sign-extended immediate to the contents of register rs, and store the result in register rt. An exception is raised in the event of a two's-complement overflow.
Add Immediate Unsigned	ADDIU rt, rs, immediate Add 32-bit sign-extended immediate to the contents of register rs, and store the result in register rt. No exception is raised on a two's-complement overflow.
Set on Less Than Immediate	SLTI rt, rs, immediate Compare 32-bit sign-extended immediate with the contents of register rs as signed 32-bit data. If rs is less than immediate, set 1 in rt as the result; otherwise store 0 in rt.
Set on Less Than Unsigned Immediate	SLTUI rt, rs, immediate Compare 32-bit sign-extended immediate with the contents of register rs as unsigned 32-bit data. If rs is less than immediate, set 1 in rt as the result; otherwise store 0 in rt.
AND Immediate	ANDI rt, rs, immediate AND 32-bit zero-extended immediate with the contents of register rs, and store the result in register rt.
OR Immediate	ORI rt, rs, immediate OR 32-bit zero-extended immediate with the contents of register rs, and store the result in register rt.
Exclusive OR Immediate	XORI rt, rs, immediate Exclusive-OR 32-bit zero-extended immediate with the contents of register rs, and store the result in register rt.
Load Upper Immediate	LUI rt, immediate Shift 16-bit immediate left 16 bits, zero-fill the least significant 16 bits of the word, and store the result in register rt.

Table 3-5. Three-operand register-type instructions

Instruction	Format and Description	op	rs	rt	rd	0	funct
Add	ADD rd, rs, rt Add the contents of registers rs and rt, and store the result in register rd. An exception is raised in the event of a two's-complement overflow.						
Add Unsigned	ADDU rd, rs, rt Add the contents of registers rs and rt, and store the result in register rd. No exception is raised on a two's-complement overflow.						
Subtract	SUB rd, rs, rt Subtract the contents of register rt from rs, and store the result in register rd. An exception is raised in the event of a two's-complement overflow.						
Subtract Unsigned	SUBU rd, rs, rt Subtract the contents of register rt from rs, and store the result in register rd. No exception is raised on a two's-complement overflow.						
Set on Less Than	SLT rd, rs, rt Compare the contents of registers rt and rs as 32-bit signed integers. If rs is less than rt, store 1 in rd as the result; otherwise store 0 in rd.						
Set on Less Than Unsigned	SLTU rd, rs, rt Compare the contents of registers rt and rs as 32-bit unsigned integers. If rs is less than rt, store 1 in rd as the result; otherwise store 0 in rd.						
AND	AND rd, rs, rt Bitwise AND the contents of registers rs and rt, and store the result in register rd.						
OR	OR rd, rs, rt Bitwise OR the contents of registers rs and rt, and store the result in register rd.						
Exclusive OR	XOR rd, rs, rt Bitwise Exclusive-OR the contents of registers rs and rt, and store the result in register rd.						
NOR	NOR rd, rs, rt Bitwise NOR the contents of registers rs and rt, and store the result in register rd.						

Table 3-6. Shift instructions

(a) SLL, SRL, SRA

Instruction	Format and Description	op	0	rt	rd	sa	funct
Shift Left Logical	SLL rd, rt, sa Left-shift the contents of register rt by the number of bits indicated in sa (shift amount), and zero-fill the low-order bits. Store the resulting 32 bits in register rd.						
Shift Right Logical	SRL rd, rt, sa Right-shift the contents of register rt by sa bits, and zero-fill the high-order bits. Store the resulting 32 bits in register rd.						
Shift Right Arithmetic	SRA rd, rt, sa Right-shift the contents of register rt by sa bits, and sign-extend the high-order bits. Store the resulting 32 bits in register rd.						

(b) SLLV, SRLV, SRAV

Instruction	Format and Description	op	rs	rt	rd	0	funct
Shift Left Logical Variable	SLLV rd, rt, sa Left-shift the contents of register rt. The number of bits shifted is indicated in the 5 low-order bits of the register rs contents. Zero-fill the low-order bits of rt and store the resulting 32 bits in register rd.						
Shift Right Logical Variable	SRLV rd, rt, sa Right-shift the contents of register rt. The number of bits shifted is indicated in the 5 low-order bits of the register rs contents. Zero-fill the high-order bits of rt and store the resulting 32 bits in register rd.						
Shift Right Arithmetic Variable	SRAV rd, rt, sa Right-shift the contents of register rt. The number of bits shifted is indicated in the 5 low-order bits of the register rs contents. Sign-extend the high-order bits of rt and store the resulting 32 bits in register rd.						

Table 3-7. Multiply/Divide Instructions

(a) MULT, MULTU, DIV, DIVU

Instruction	Format and Description	op	rs	rt	0	funct
Multiply	MULT rs, rt Multiply the contents of registers rs and rt as two's complement integers, and store the doubleword (64-bit) result in multiply/divide registers HI and LO.					
Multiply Unsigned	MULTU rs, rt Multiply the contents of registers rs and rt as unsigned integers, and store the doubleword (64-bit) result in multiply/divide registers HI and LO.					
Divide	DIV rs, rt Divide register rs by register rt as two's complement integers. Store the 32-bit quotient in LO, and the 32-bit remainder in HI.					
Divide Unsigned	DIVU rs, rt Divide register rs by register rt as unsigned integers. Store the 32-bit quotient in LO, and the 32-bit remainder in HI.					

(b) MFHI, MFLO

Instruction	Format and Description	op	rs	rt	0	funct
Move From HI	MFHI rd Store the contents of multiply/divide register HI in register rd.					
Move From LO	MFLO rd Store the contents of multiply/divide register LO in register rd.					

(c) MTHI, MTLO

Instruction	Format and Description	op	rs	0	funct
Move To HI	MTHI rs Store the contents of register rs in multiply/divide register HI.				
Move To LO	MTLO rs Store the contents of register rs in multiply/divide register LO.				

Table 3-8. Multiply, multiply / add instructions (R3000A extended instruction set)

MULT, MULTU, MADD, MADDU (ISA extended set)

Instruction	Format and Description	op	rs	rt	rd	0	funct
Multiply	MULT rd, rs, rt Multiply the contents of registers rs and rt as two's complement integers, and store the doubleword (64-bit) result in multiply/divide registers HI and LO. Also, store the lower 32 bits in register rd.						
Multiply Unsigned	MULTU rd, rs, rt Multiply the contents of registers rs and rt as unsigned integers, and add the doubleword (64-bit) result to multiply/divide registers HI and LO. Also, store the lower 32 bits in register rd.						
Multiply ADD	MADD rd, rs, rt MADD rs, rt Multiply the contents of registers rs and rt as two's complement integers, and add the doubleword (64-bit) result to multiply/divide registers HI and LO. Also, store the lower 32 bits of the add result in register rd. In the MADD rs, rt format, the store operation to a general register is omitted.						
Multiply ADD Unsigned	MADDU rd, rs, rt MADDU rs, rt Multiply the contents of registers rs and rt as unsigned integers, and add the doubleword (64-bit) result to multiply/divide registers HI and LO. Also, store the lower 32 bits of the add result in register rd. In the MADDU rs, rt format, the store operation to a general register is omitted.						

3.5 Jump/Branch Instructions

Jump/branch instructions change the program flow. A jump/branch instruction will delay the pipeline by one instruction cycle, however, an instruction inserted into the delay slot (immediately following a branch instruction) can be executed while the instruction at the branch target address is being fetched.

Jump and Jump And Link instructions, typically used to call subroutines, have the J-type instruction format. The jump target address is generated as follows. The 26-bit target address (target) of the instruction is left-shifted two bits and combined with the high-order four bits of the current PC (program counter) value to form a 32-bit absolute address. This becomes the branch target address of the jump instruction. The PC shows the address of the branch delay slot at that time.

The Jump And Link instruction puts the return address in register r31.

The R-type instruction format is used for returns from subroutines and long-distance jumps beyond one page (Jump Register and Jump And Link Register instructions). The register value in this format is a 32-bit byte address.

Branch instructions use the I-type format. Branching is to an relative address determined by adding a 16-bit signed offset to the program counter.

Table 3-9. Jump instructions

(a) J, JAL

Instruction	Format and Description	op	target
Jump	J target Left-shift the 26-bit target by two bits and, after a one-instruction delay, jump to an address formed by combining this result with the high-order 4 bits of the program counter (PC).		
Jump And Link	JAL target Left-shift the 26-bit target by two bits and, after a one-instruction delay, jump to an address formed by combining the result with the high-order 4 bits of the program counter (PC). Store in r31 (link register) the address of the instruction following the instruction in the delay slot (The instruction in the delay slot is executed during the jump).		

(b) JR

Instruction	Format and Description	op	rs	0	funct
Jump Register	JR rs Jump to the address in register rs after a one-instruction delay.				

(c) JALR

Instruction	Format and Description	op	rs	0	rd	0	funct
Jump And Link Register	JALR rs, rd Jump to the address in register rs after a one-instruction delay. Store in rd the address of the instruction following the instruction in the delay slot (the instruction in the delay slot is executed during the jump).						

The following notes apply to Table 3-10.

- The target address of a branch instruction is generated by adding the address of the instruction in the delay slot (the instruction to be executed during the branch) to the 16-bit offset (that has been left-shifted two bits and sign-extended to 32 bits). Branch instructions are executed with a one-cycle delay.
- In the case of the Branch Likely instructions in Table 3-10, if the branch condition is not met and the branch is not taken, the instruction in the delay slot is treated as a NOP.

Table 3-10. Branch instructions

(a) BEQ, BNE

Instruction	Format and Description	op	rs	rt	offset
Branch on Equal	BEQ rs, rt, offset Branch to the target if the contents of registers rs and rt are equal.				
Branch on Not Equal	BNE rs, rt, offset Branch to the target if the contents of registers rs and rt are not equal.				

(b) BLEZ, BGTZ

Instruction	Format and Description	op	rs	0	offset
Branch on Less Than or Equal Zero	BLEZ rs, offset Branch to the target if register rs is 0 or less.				
Branch on Greater Than Zero	BGTZ rs, offset Branch to the target if register rs is greater than 0.				

(c) BLTZ, BGEZ, BLTZAL, BGEZAL

Instruction	Format and Description	op	rs	funct	offset
Branch on Less Than Zero	BLTZ rs, offset Branch to the target if register rs is less than zero				
Branch on Greater Than or Equal Zero	BGEZ rs, offset Branch to the target if register rs is 0 or greater.				
Branch on Less Than Zero And Link	BLTZAL rs, offset Store in r31 (link register) the address of the instruction following the instruction in the delay slot (the one to be executed during the branch). If register rs is less than 0, branch to the target.				
Branch on Greater Than or Equal Zero And Link	BGEZAL rs, offset Store in r31 (link register) the address of the instruction following the instruction in the delay slot (the instruction in the delay slot is executed during the branch). If register rs is 0 or greater, branch to the target.				

(d) BEQL, BNEL, BLEZL, BGTZL, BLTZL, BGEZL, BLTZALL, BGEZALL (ISA Extended Set)

Instruction	Format and Description	op	rs	rt	offset
Branch on Equal Likely	BEQL rs, rt, offset Branch to the target if the contents of registers rs and rt are equal.				
Branch on Not Equal Likely	BNEL rs, rt, offset Branch to the target if the contents of registers rs and rt are not equal.				
Branch on Less Than or Equal Zero Likely	BLEZL rs, offset Branch to the target if register rs is 0 or less.				
Branch on Greater Than Zero Likely	BGTZL rs, offset Branch to the target if register rs is greater than 0.				
Instruction	Format and Description	op	rs	funct	offset
Branch on Less Than Zero Likely	BLTZL rs, offset Branch to the target if register rs is less than zero				
Branch on Greater Than or Equal Zero Likely	BGEZL rs, offset Branch to the target if register rs is 0 or greater.				
Branch on Less Than Zero And Link Likely	BLTZALL rs, offset Store in r31 (link register) the address of the instruction following the instruction in the delay slot (the one to be executed during the branch). If register rs is less than 0, branch to the target.				
Branch on Greater Than or Equal Zero And Link Likely	BGEZALL rs, offset Store in r31 (link register) the address of the instruction following the instruction in the delay slot (the instruction in the delay slot is executed during the branch). If register rs is 0 or greater, branch to the target.				

3.6 Special Instructions

There are three special instructions used for software traps. The instruction format is R-type for all three.

Table 3-11. Special instructions

(a) SYSCALL

Instruction	Format and Description	op	code	funct
System Call	SYSCALL code Raise a system call exception, passing control to an exception handler.			

(b) BREAK

Instruction	Format and Description	op	code	funct
Breakpoint	BREAK code Raise a breakpoint exception, passing control to an exception handler.			

(c) SDBBP

Instruction	Format and Description	op	code	funct
Software Debug Breakpoint	SDBBP code Raise a debug exception, passing control to an exception processor.			

3.7 Coprocessor Instructions

Coprocessor instructions invoke coprocessor operations. The format of these instructions depends on which coprocessor is used.

Table 3-12. Coprocessor instructions

(a) MTCz, MFCz, CTCz, CFCz

Instruction	Format and Description	op	funct	rt	rd	0
Move To Coprocessor	MTCz rt, rd Move the contents of CPU general register rt to coprocessor z's coprocessor register rd.					
Move From Coprocessor	MFCz rt, rd Move the contents of coprocessor z's coprocessor register rd to CPU general register rt.					
Move Control To Coprocessor	CTCz rt, rd Move the contents of CPU general register rt to coprocessor z's coprocessor control register rd.					
Move Control From Coprocessor	CFCz rt, rd Move the contents of coprocessor z's coprocessor control register rd to CPU general register rt.					

(b) COPz

Instruction	Format and Description	op	co	cofun
Coprocessor Operation	COPz cofun Execute in coprocessor z the processing indicated in cofun. The CPU state is not changed by the processing executed in the coprocessor.			

(c) BCzT, BCzF

Instruction	Format and Description	op	funct	offset
Branch on Coprocessor z True	BCzT offset Generate the branch target address by adding the address of the instruction in the delay slot (the instruction to be executed during the branch) and the 16-bit offset (after left-shifting two bits and sign-extending to 32 bits). If the coprocessor z condition line is true, branch to the target address after a one-cycle delay.			
Branch on Coprocessor z False	BCzF offset Generate the branch target address by adding the address of the instruction in the delay slot (the instruction to be executed during the branch) and the 16-bit offset (after left-shifting two bits and sign-extending to 32 bits). If the coprocessor z condition line is false, branch to the target address after a one-cycle delay.			

(d) BCzTL, BCzFL (ISA Extended Set)

Instruction	Format and Description	op	funct	offset
Branch on Coprocessor z True Likely	BCzTL offset Generate the branch target address by adding the address of the instruction in the delay slot (the instruction to be executed during the branch) and the 16-bit offset (after left-shifting two bits and sign-extending to 32 bits). If the coprocessor z condition line is true, branch to the target address after a one-cycle delay. If the condition line is false, nullify the instruction in the delay slot.			
Branch on Coprocessor z False Likely	BCzFL offset Generate the branch target address by adding the address of the instruction in the delay slot (the instruction to be executed during the branch) and the 16-bit offset (after left-shifting two bits and sign-extending to 32 bits). If the coprocessor z condition line is false, branch to the target address after a one-cycle delay. If the condition line is true, nullify the instruction in the delay slot.			

3.8 System Control Coprocessor (CP0) Instructions

Coprocessor 0 instructions are used for operations involving the system control coprocessor (CP0) registers, processor memory management and exception handling.

Note: Attempting to execute a CP0 instruction in user mode when the CU0 bit in the status register is not set will return a Coprocessor Unusable exception.

Table 3-13. System Control Coprocessor (CP0) Instructions

(a) MTC0, MFC0

Instruction	Format and Description	op	funct	rt	rd	0
Move To CP0	MTC0 rt, rd Move the contents of CPU general register rt to CP0 coprocessor register rd.					
Move From CP0	MFC0 rt, rd Move the contents of CP0 coprocessor register rd to CPU general register rt.					

(b) RFE, DERET

Instruction	Format and Description	op	co	0	funct
Restore From Exception	RFE Restore the previous mode bit of the Status register and Cache register into the corresponding current mode bit, and restore the old status bit into the corresponding previous mode bit.				
Debug Exception Return	DERET Branch to the value in the CP0 DEPC register.				

(c) CACHE

Instruction	Format and Description	op	base	op	offset
Cache Operation	CACHE op, offset (base) Add the contents of the CPU general registers designated by base and offset to generate a virtual address. The MMU translates this virtual address to a physical address. The cache operation to be performed at this address is contained in op.				

Chapter 4 Pipeline Architecture

4.1 Overview

The TX39 Processor Core executes instructions in five pipeline stages (F: instruction fetch; D: decode; E: execute; M: memory access; W: register write-back). The five stages have the following roles.

F: An instruction is fetched from the instruction cache.

D: The instruction is decoded. Contents of the general-purpose registers are read. If the instruction involves a branch or jump, the target address is generated. The coprocessor condition signal is latched.

E: Arithmetic, logical and shift operations are performed. The execution of multiple/divide instructions is begun.

M: The data cache is accessed in the case of load and store instructions.

W: The result is written to a general register.

Each pipeline stage is executed in one clock cycle. When the pipeline is fully utilized, five instructions are executed at the same time, resulting in an average instruction execution rate of one instruction per cycle as illustrated in Figure 4-1.

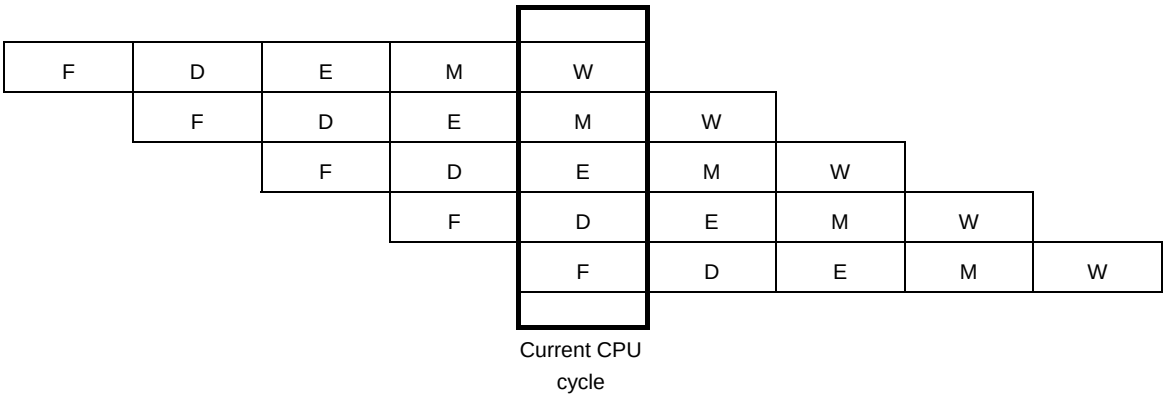


Figure 4-1. Pipeline stages for execution of TX39 Processor Core instructions

4.2 Bypassing (Forwarding) and Hazards

In the instruction example shown in Figure 4-2, the arithmetic result of the ADD instruction is used by the subsequent ADDU instruction. Although the ADD instruction writes the arithmetic result to general-purpose register r1 at the W stage, this is too late for the arithmetic operation (at the E stage) of the subsequent ADDU instruction. To avoid this problem, the TX39 core directly passes the arithmetic result of a preceding instruction from output of its arithmetic unit to the input of the arithmetic unit of the subsequent instruction. This is called bypassing (or forwarding).

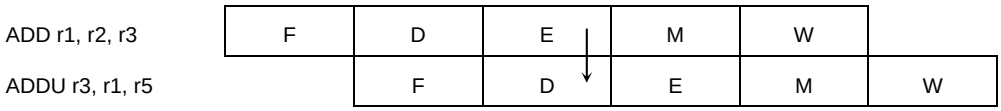


Figure 4-2. Bypassing (Forwarding)

In the instruction example shown in Figure 4-3, the MFHI instruction result (written to general-purpose register r1) is used by the subsequent ADDIU instruction. Because the MFHI instruction reads the HI register at the M stage, even if bypassing is performed, it is too late for the arithmetic operation (at the E stage) of the subsequent ADDIU instruction. Therefore, the TX39 core stalls the ADDIU instruction pipeline and waits for the necessary data to become available.

A problem caused by dependency of data shared between instructions is called pipeline hazard (hazard). Stalling the pipeline to avoid a hazard is called pipeline stall (stall).

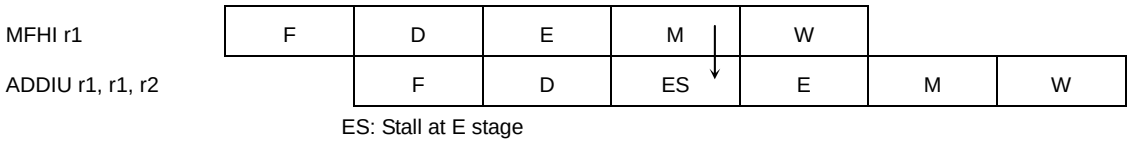


Figure 4-3. Pipeline stall

4.3 Delay Slot

Some TX39 Processor Core instructions are executed with a delay of one instruction cycle. The cycle in which an instruction is delayed is called a delay slot. A delay occurs with load instructions and branch/jump instructions.

4.3.1 Delayed load

With load instructions, a one-cycle delay occurs while waiting for the data being loaded to become available for use by another instruction. The TX39 Processor Core checks the instruction in the delay slot (the instruction immediately following the load instruction) to see if that instruction needs to use the load result; if so, it stalls the pipeline (see Figure 4-4).

With the R3000A, if the instruction following a load instruction required access to the loaded data, then a NOP had to be inserted immediately after the load instruction. The delay load feature in the TX39 Processor Core eliminates the need for a NOP instruction, resulting in smaller code size than with the R3000A.

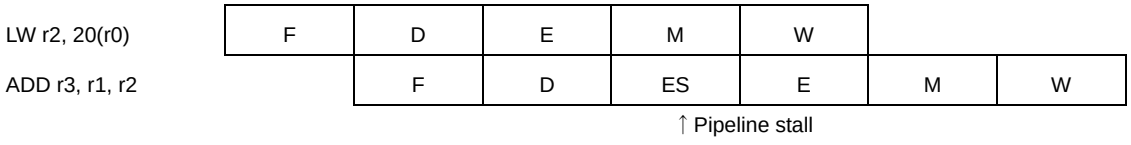


Figure 4-4. Load delay slot and pipeline stall

4.3.2 Delayed branching

Figure 4-5 shows the pipeline flow for jump/branch instructions. The branch target address that must be generated for these type of instructions does not become available until the E stage – too late to be used by the instruction in the branch delay slot. The branch target instruction is fetched immediately after the branch delay slot cycle.

It is, however, possible to fetch a different instruction that would normally be executed prior to the branch instruction.

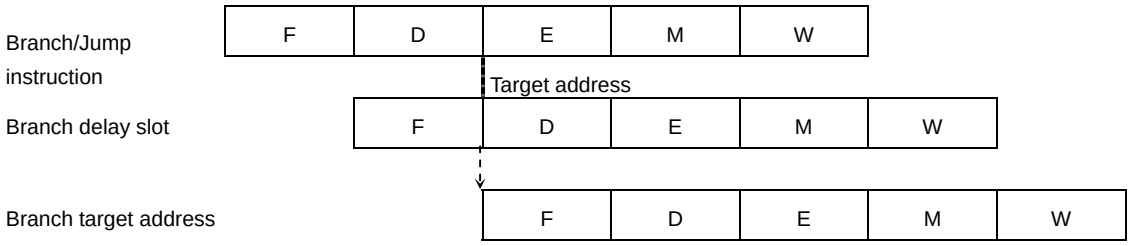


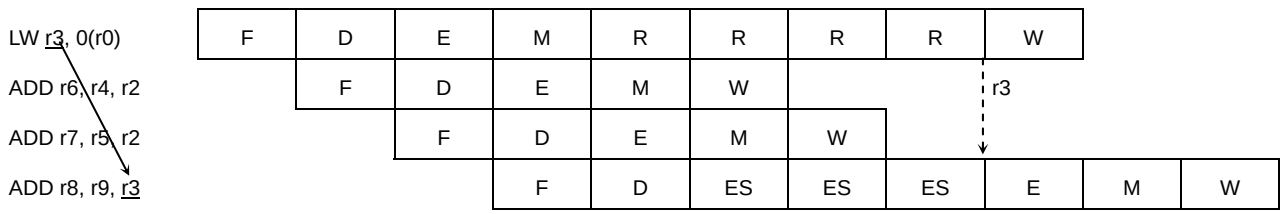
Figure 4-5. Branch instruction delay slot

You can make effective use of the branch delay slot as follows.

- Since the instruction immediately following a branch instruction will be executed just priot to the branch, you can therefore place an instruction (that logically should be executed just before the branch) into the delay slot following the branch instruction.
- The TX39 Processor Core provides Branch Likely instructions in addition to the normal Branch instructions that allow the instruction at the target branch address to be placed in the delay slot. If the branch condition of the Branch Likely instruction is met, the instruction in the delay slot is executed and the branch is taken. If the branch is not taken, the instruction in the delay slot is treated as a NOP. With the R3000A, which dose not support the Branch Likely instruction, the only instructions that can be placed in the delay slot are those unaffected if the branch is not taken.
- If no instruction is placed in the delay slot, a NOP is placed just after the branch instruction.

4.4 Nonblocking Load Function

The nonblocking load function prevents the pipeline from stalling when a cache miss occurs and a refill cycle is required to refill the data cache. Instructions after the load instruction that do not use registers affected by the load will continue to be executed. An example is shown in Figure 4-6. Here a cache miss occurs with the first load instruction. The two instructions following are executed prior to the load. The fourth instruction (ADD), must use a register that will be loaded by the load instruction, therefore the pipeline is stalled until the cache data becomes valid.

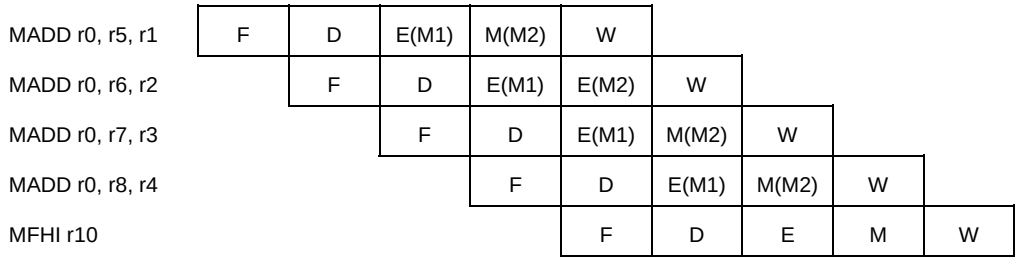


R: Refill cycle, ES: Stall in E stage

Figure 4-6. Nonblocking load function

4.5 Multiply and Multiply/Add Instructions (MULT, MULTU, MADD, MADDU)

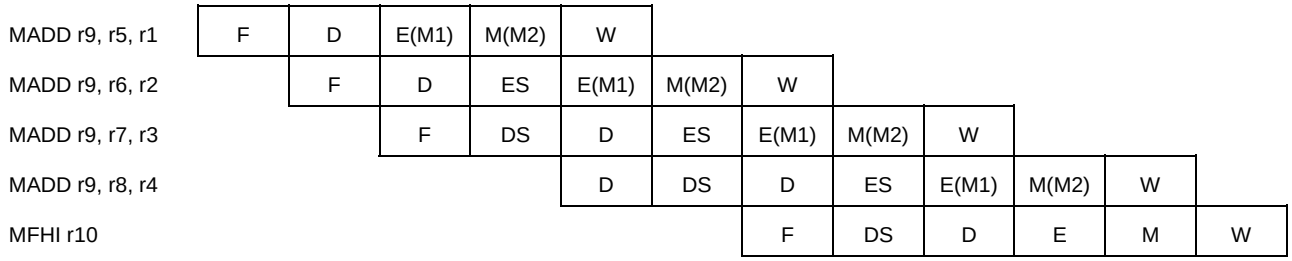
The TX39 Processor Core can execute multiply and multiply/add instructions continuously without pipeline stall, and can use the results in the HI/LO registers in subsequent instructions without pipeline stall, if the results are written to the general-purpose register r0. (Figure 4-7 (a).)



M1: First multiply stage; M2: Second multiply stage

(a) Continued execution of MADD (the pipeline is not stall)

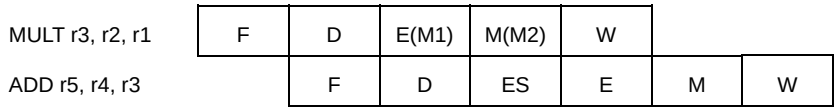
However, if the results are written to a general-purpose register that is not r0, the pipeline is stalled. (Figure 4-7 (b).)



M1: First multiply stage; M2: Second multiply stage

(b) Continued execution of MADD (the pipeline is stalled)

The TX39 Processor Core requires only one clock cycle to use the results of a general-purpose register. (Figure 4-7 (c).)



(c) When there is data dependency in a general-purpose register.

Figure 4-7. Pipeline operation with a multiply instructions

4.6 Divide Instruction (DIV, DIVU)

The TX39 Processor Core performs division instructions in the division unit independently of the pipeline. Division starts from the pipeline E stage and takes 35 cycles. Figure 4-8 shows an example of a divide instruction.

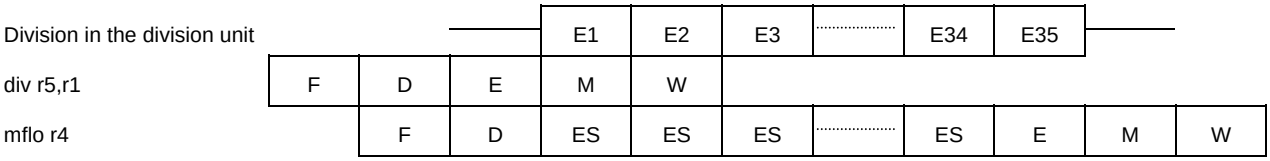


Figure 4-8. Example of DIV instruction

Note: When an MTHI, MTLO, DIV or DIVU instruction comes up for execution when a DIV or DIVU instruction is already being executed in progress, the TX39 will stop the DIV or DIVU in progress and will begin executing the MTHI, MTLO or new DIV or DIVU instruction.

The TX39 Processor Core will not halt execution of a DIV or DIVU instruction when an exception occurs during its execution.

Division stops in Halt and Doze mode. It restarts when the TX39 returns from Halt or Doze mode.

4.7 Streaming

During a cache refill operation, the TX39 Processor Core can resume execution immediately after arrival of necessary data or instruction in cache even though cache refill operation is not completed. This is referred to as "streaming."

4.8 Pipeline Hazards

While the TX39 Processor Core makes every effort to prevent pipeline hazards from occurring through such measures as bypassing and streaming, pipeline hazards can not be completely eliminated and the pipeline sometimes stalls. This section describes the cases in which stalling occurs.

4.8.1 Load instruction

a) Interlock in a load delay slot

As described in Section 4.3.1, Delayed load, an instruction that reads the destination register of the load instruction can be placed immediately after a load instruction. In this case, the pipeline is stalled by a hardware interlock mechanism until the reading of the data from the load instruction memory area (or cache) completes.

Note: In the R3000A, the load delay slot does not include a hardware interlock mechanism. For the case described above, the hazard must be avoided by such software measures as inserting a NOP instruction immediately after the load instruction.

b) Nonblocking load function

When a load instruction results in a cache miss or load from an uncache area, and the subsequent instruction writes the data to the destination register, a hazard may occur. The following is an example of such a case.

A hazard occurs because the ADDU instruction destination register is the same register as that for the LW instruction. In this case, the ADDU instruction stalls at the E stage until the load instruction finishes reading the result from the memory to the r1 register.

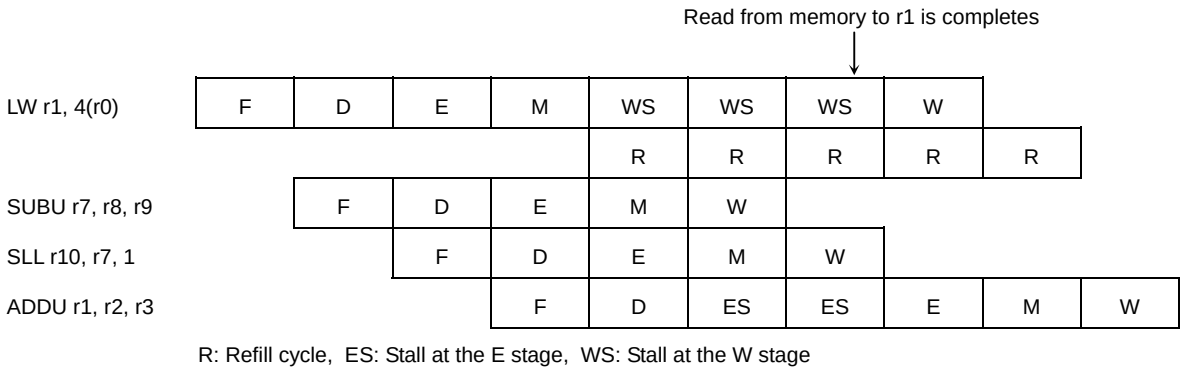


Figure 4-9. Hazard in nonblocking load

c) SYNC instruction

When the SYNC instruction follows a load instruction, the pipeline stalls until the bus cycle of the load instruction completes. The SYNC instruction itself is stalled at the M stage until the load instruction's bus cycle completes.

If the load instruction results in a cache hit, the SYNC instruction does not stall.

If a bus error occurs in a load instruction read bus cycle, the exception program counter (EPC) is set to the address of the subsequent SYNC instruction address.

4.8.2 Store instructions

a) Execution of a load instruction immediately after a store instruction

When a load instruction loads the data stored by an immediately preceding store instruction to an uncache area, the read bus cycle of the load instruction is executed after the write bus cycle of the SW instruction completes, thus guaranteeing the data dependency.

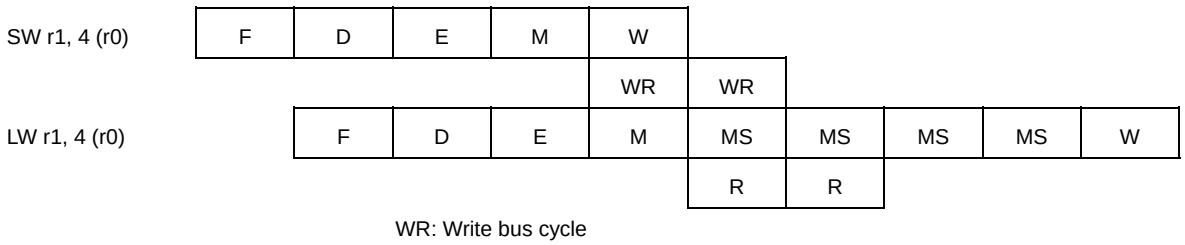


Figure 4-10. Load instruction immediately after store to uncache area

b) SYNC instruction

When the SYNC instruction follows a store instruction, the pipeline stalls until the bus cycle of the store instruction completes. The SYNC instruction itself is stalled at the M stage until the store instruction's bus cycle completes.

4.8.3 Multiply and divide instructions

Multiply and divide instructions (MULT, MULTU, MADD, MADDU, DIV, DIVU, MFHI, MFLO, MTHI, MTLO) use the HI/LO registers in addition to general-purpose registers. Because the execution time for a multiply or divide instruction is longer than that for other instructions, a special hazard occurs.

a) MULT, MULTU, MADD, MADDU instructions

Multiply and multiply/add instructions are executed using the E and M stages. Accordingly, when an instruction immediately after a multiply or multiply/add instruction attempts to reference the general-purpose register used as the destination register for the result of the preceding instruction, the pipeline stalls.

The general-purpose registers are read (or bypassed) at the D stage. When the destination register of a MULT, MULTU, MADD, or MADDU instruction is referenced by the subsequent instruction, the pipeline stalls for one clock cycle.

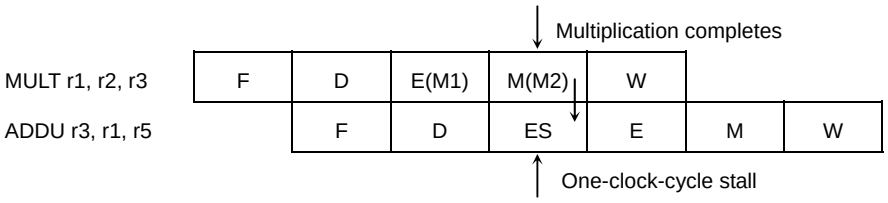


Figure 4-11. Hazard caused by referencing the destination register of a multiply instruction

b) DIV, DIVU instructions

DIV and DIVU instructions require 35 clock cycles to be executed. The DIV and DIVU source rs and rt registers are read at the E stage. The DIV and DIVU instructions are executed by a special divider. Data-independent instructions can be executed during DIV or DIVU instruction execution so that they complete in advance of DIV and DIVU. Data-dependent instructions (MFHI, MFLO, MADD, MADDU) are stalled at the E stage until the DIV or DIVU instruction complete. An interrupt can occur even during execution of a DIV or DIVU instruction.

b-1) Data hazards caused by an MFHI or MFLO instruction

The MFHI or MFLO instruction, which depends on DIV or DIVU instruction data, is stalled at the E stage until the DIV or DIVU division completes.

Note: If an interrupt request is generated during a stall, an interrupt occurs. In this case, At this time, the EPC is set to the stalled MFHI or MFLO instruction address.

b-2) Data hazards caused by a MULT, MULTU, MADD, or MADDU instruction

Executing a MULT, MULTU, MADD, or MADDU instruction immediately after a DIV or DIVU instruction stalls the pipeline until the DIV or DIVU division completes.

Note: If an interrupt request is generated during a stall, an interrupt occurs. At this time, the EPC is set to the stalled MULT, MULTU, MADD, or MADDU instruction address.

c) MFHI or MFLO instruction

The MFHI or MFLO instruction reads the HI or LO register at the M stage. When the subsequent instruction references the destination register, the pipeline stalls.

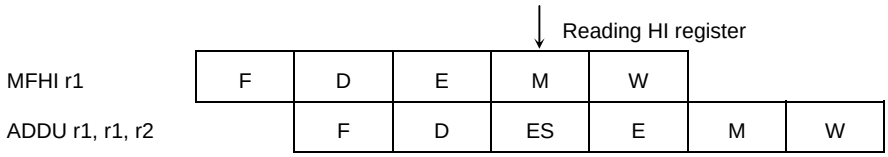


Figure 4-12. Hazard caused by referencing destination register of MFHI or MFLO instruction

The MFHI instruction reads the HI register at the M stage. When the subsequent ADDU instruction references the ADDU instruction's destination register (r1), the pipeline stalls for one clock cycle.

4.8.4 System control coprocessor (CP0) instructions

a) MFC0 instruction

The MFC0 instruction reads the CP0 register at the M stage. When the subsequent instruction reads the general-purpose register used as the MFC0 instruction destination register, the pipeline stalls.

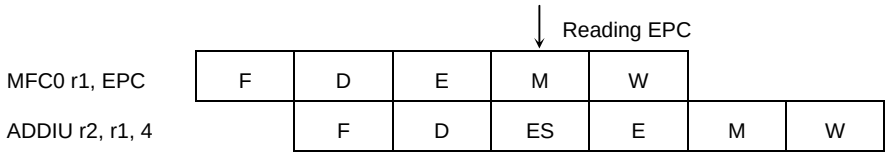


Figure 4-13. Hazard caused by referencing the destination register of the MFC0 instruction

Immediately after the MFC0 instruction reads the EPC, the ADDIU instruction referencing the MFC0 instruction destination register (r1) stalls for one clock cycle.

b) MTC0 instruction

The MTC0 instruction writes the value of a general-purpose register to the CP0 register at the M stage.

b-1) Doze bit (Config register) update

When the MTC0 instruction is used to set the Doze bit, the TX39 Processor Core completes all the bus cycles, then stalls the pipeline and asserts the DOZE signal.

b-2) Halt bit (Config register) update

When the MTC0 instruction is used to set the Halt bit, the TX39 Processor Core completes all the bus cycles, then stalls the pipeline and asserts the HALT signal.

4.8.5 Coprocessor instructions

a) MFCz, CFCz instructions

Instructions with no data dependence on MFCz and CFCz instruction destination general-purpose registers are executed in advance of MFCz and CFCz. Instruction with data dependence on those registers are stalled.

A hazard occurs when the destination registers of the MFCz and CFCz instructions are the same as those of the following instructions. In such a case, the instructions that follow the MFCz and CFCz instructions stall at the E stage.

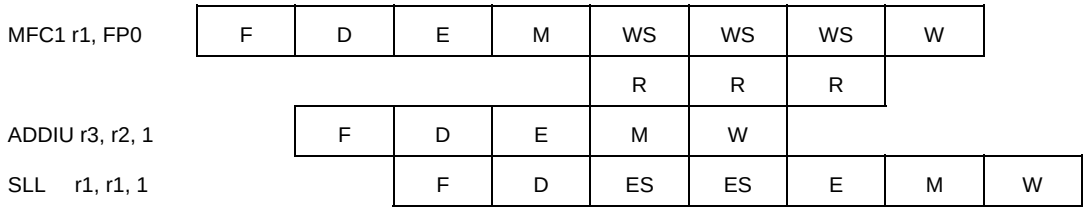


Figure 4-14. Where instructions after MFCz and CFCz use MFCz and CFCz destination registers

b) COPz instructions

To allow correct execution of exceptions the COP1 and COP2 instructions of coprocessors 1 and 2, instructions following COP1 and COP2 stall until the coprocessor read bus cycle completes. However, the COP3 instruction of coprocessor 3 does not stall the instructions following it. If it is necessary to stall instructions following COP3, use the SYNC instruction.

4.9 Restrictions on Instruction Placement

4.9.1 Multiply/Divide instructions

a) Reading the HI or LO register by the next or next-but-one instruction before a DIV or DIVU instruction

The value of the HI and LO registers are undefined for the two instructions immediately preceding a DIV or DIVU instruction. Accordingly, do not place a DIV or DIVU instruction as the next or next-but-one instruction after a MADD, MADDU, MFHI, or MFLO instruction that reads the HI or LO registers. However, note that the MULT or MULTU instructions that do not read the HI or LO registers can be used as the next or next-but-one instruction before a DIV or DIVU instruction.

b) Executing a MTHI, MTLO, DIV, or DIVU is executed instruction during division

When an attempt is made to execute a MTHI, MTLO, DIV or DIVU instruction that updates the HI or LO registers while division is in progress, division processing halts, then the MTHI, MTLO, DIV or DIVU instruction that follows is executed. The LO register value is undefined during execution of an MTHI instruction. The HI register value is undefined during execution of an MTLO instruction.

4.9.2 Jump and branch instructions

a) Jump and branch instructions

A jump/branch instruction must not immediately follow another jump/branch instruction. Operations cannot be guaranteed for consecutive jump/branch instructions. However, an instruction that generates exceptions, such as SYSCALL, BREAK, or SDBBP instruction, may immediately follow a jump/branch instruction.

b) JALR instruction

The source and destination registers used by the JALR instruction must be different registers. Where the same register is specified as the source and destination registers, the result is undefined. An exception is not generated in such a case.

4.9.3 System control coprocessor (CP0) instructions

b-1) CU bit (Status register) update

When the MTC0 instruction is used to update the CU bit, the new value is valid from the third instruction after MTC0. Because it is undefined whether the two instructions immediately after the CU bit update reference the pre- or post-update CU bit value, do not use a coprocessor-related instruction as the next or next-but-one instruction immediately after a CU bit update.

b-2) RE bit (Status register) update

When the MTC0 instruction is used to update the RE bit, the new value is valid from the second instruction after MTC0. Because it is undefined whether the instruction immediately after the RE bit update references the pre- or post-update RE bit value, do not place a load or store instruction immediately after MTC0.

When the RE bit is updated by the MTC0 instruction during a bus operation caused by a load or store instruction before MTC0, the load or store instruction uses the pre-update RE bit value. Therefore, before executing MTC0, it is not necessary to complete the bus cycle by executing an instruction such as SYNC.

b-3) BEV bit (Status register) update

When the MTC0 instruction is used to update the BEV bit, exceptions (address error (instruction fetch) and TLBL (instruction fetch)) generated at the E stage use the vector address specified by the new BEV value from the second instruction after the update. It is undefined whether the instruction immediately after the MTC0 instruction references the pre- or post-update BEV bit value.

Where exception is generated at the M stage, the new BEV value is valid from the instruction immediately after MTC0.

To enable the BEV bit which is referenced by an interrupt exception (Int), update the BEV bit with interrupts disable (IEc=0). Also, to enable the BEV bit which is referenced by a bus error exception (IBE, DBE), execute SYNC immediately before MTC0, then complete the bus cycle caused by an instruction preceding MTC0.

b-4) IntMask and IEc bits (Status register) update

When the MTC0 instruction is used to update the IntMask and IEc bits and also to enable interrupts, interrupts are actually enabled from the second instruction after MTC0. Depending on the pipeline stall status, however, interrupts may be enabled from the instruction immediately after MTC0. When the MTC0 instruction is used to disable interrupts, interrupts are disabled from the instruction immediately after MTC0.

b-5) KUc bit (Status register) update

When the MTC0 instruction is used to update the KUc bit, the new value is valid from the third instruction after MTC0. Because it is undefined whether the two instructions immediately after the KUc bit update reference the pre- or post-update KUc bit value, do not use a coprocessor-related instruction or an instruction used to access the kernel area as the next or the next-but-one instruction immediately after a KUc bit update.

b-6) DALc bit (Cache register) update

When the MTC0 instruction is used to update the DALc bit, the new value is valid from the second instruction (a load or store instruction) after MTC0. Because it is undefined whether the instruction immediately after the DALc bit update references the pre- or post-update DALc bit value, do not place a load or store instruction immediately after MTC0.

b-7) DCBR and DRSize bits (Config register) update

When the MTC0 instruction is used to update the DCBR and DRSize bits, which specify the data cache refill size, the new refill size is valid from the second instruction (a load instruction) after MTC0. It is undefined whether the instruction immediately after MTC0 references the pre- or post-update refill size value.

Note: The data cache refill size can be charged regardless of whether the data cache is enabled or disabled. When MTC0 updates the refill size while the refill cycle is being executed by a load instruction that precedes MTC0, the refill cycle operations use the pre-update refill size. Accordingly, it is not necessary to execute SYNC before MTC0.

b-8) DCE bit (Config register) update

When the MTC0 instruction is used to update the DCE bit, the new value is valid and the data cache enabled or disabled from the second instruction after MTC0. Because it is undefined whether the instruction immediately after MTC0 references the pre- or post-update DCE bit value, do not place an instruction to access the data cache immediately after the MTC0 instruction used to update the DCE bit.

b-9) IRSize bit (Config register) update

When the MTC0 instruction is used to update the IRSize bit, which specifies the instruction cache refill size, the new refill size is valid from the subsequent instruction cache refill cycle. The currently executing instruction cache refill cycle uses the original refill size set prior to the update.

The instruction cache refill size can be changed regardless of whether the instruction cache is enabled or disabled.

b-10) ICE bit (Config register) update

During the instruction cache refill cycle, the MTC0 instruction used to disable the instruction cache (to write 0 to the ICE bit) may be executed by streaming (instruction fetch bypass). In that case, the instruction cache is enabled until the refill cycle completes.

To disable the instruction cache without waiting for completion of the refill cycle, use the following sequence. This will disable the instruction cache from the execution of target instruction.

```
    MTC0  r1,Config; writes 0 to ICE
    BEQ   r0,r0,L1; stops streaming by branching
    nop
L1: LW    r2,0(r0); disables instruction cache starting from this instruction
```

When the instruction cache is disabled, the instruction fetch is performed in units of four words (16 bytes). Where the instruction sequence is separated into blocks of four word each, the ICE bit is referenced at the D stage of the first instruction of each block. Where the ICE bit is set by the MTC0 instruction loaded at byte address 16 n + 0, + 4, or + 8, the instruction cache is enabled from the first instruction (byte address 16 n + 16) of the next four-word block. Where the ICE bit is set by the MTC0 instruction loaded at byte address 16 n + 12, the instruction cache is enabled from the first instruction (byte address 16 n + 32) of the second four-word block.

In addition, the instruction fetch for the instruction loaded at the branch destination address always references the ICE bit. Executing a branch instruction after an MTC0 instruction used to set the ICE bit to 1 always enables the instruction cache from the instruction fetch for the instruction loaded at the branch destination address.

b-11) RFE instruction

The new settings of KUc (kernel/user mode), IEc (interrupt enable), and DALc (data cache autolock) bits are valid from the instruction immediately after the RFE instruction.

The RFE instruction updates the KUp, KUc, IEp, and IEc bits of the Status register in the first half of the E stage. Therefore, where the RFE instruction is used immediately after the MTC0 or MFC0 instruction, which accesses the Status register at the M stage, a hazard occurs. Be sure to place the MTC0/MFC0 and RFE instructions separately by inserting at least one instruction between them.

Where an interrupt occurs as a result of the RFE instruction, the Status register value is undefined. Excecute the RFE instruction with interrupts disabled (IEc = 0).

Chapter 5 Memory Management Unit (MMU)

The TX39 Processor Core doesn't have TLB.

5.1 TX39 Processor Core Operating Modes

The TX39 Processor Core has two operating modes, user mode and kernel mode. Normally it operates in user mode, but when an exception is detected it goes to kernel mode. Once in kernel mode, it remains until an RFE (Restore From Exception) instruction is executed. The available virtual address space differs with the mode, as shown in Figure 5-1.

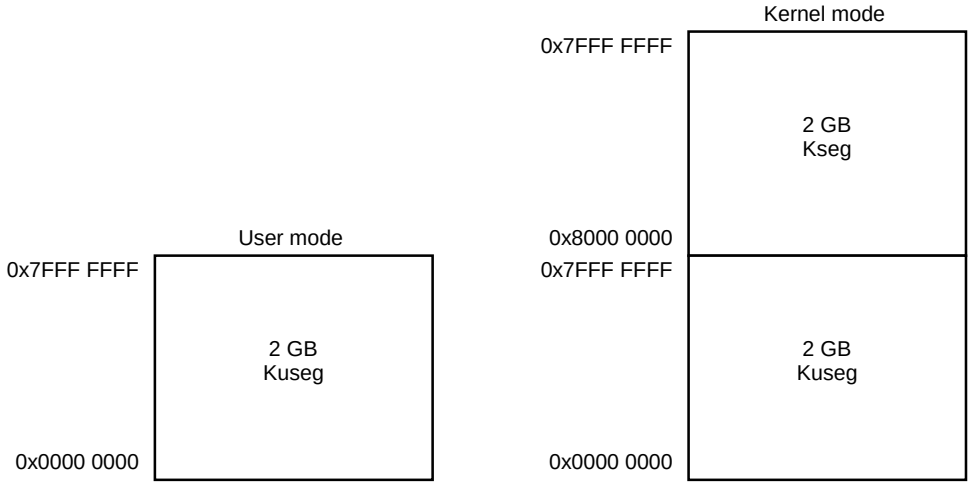


Figure 5-1. Operating modes and virtual address spaces

(1) User mode

User mode makes available only one of the two 2 Gbyte virtual address spaces (kuseg). The most significant bit of each kuseg address is 0. The virtual address range of kuseg is 0x0000 0000 to 0x7FFF FFFF. Attempting to access an address when the MSB is 1 while in user mode returns an Address Error exception.

(2) Kernel mode

Kernel mode makes available a second 2 Gbyte virtual address space (ksef), in addition to the kusef accessible in user mode. The virtual address range of ksef is 0x8000 0000 to 0xFFFF FFFF.

5.2 Direct Segment Mapping

The TX39 Processor Core has a direct segment mapping MMU.

Figure 5-2 shows the virtual address space of the internal MMU.

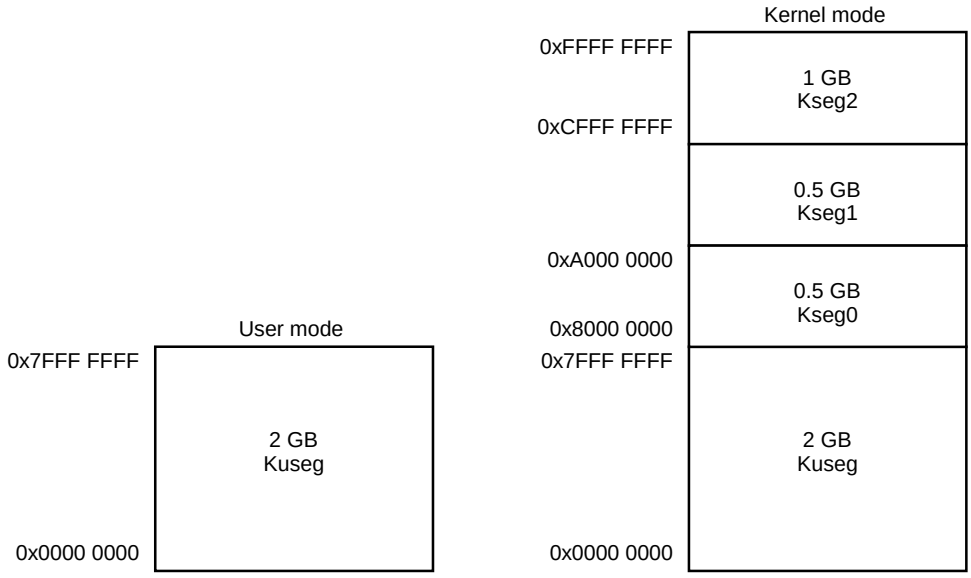


Figure 5-2. Internal MMU virtual address space

(1) User mode

One 2 Gbyte virtual address space (kuseg) is available in user mode. In this mode, the most significant bit of each kuseg address is 0. The virtual address range of kuseg is 0x0000 0000 to 0x7FFF FFFF. Attempting to access an address outside of this range, that is, with the MSB is 1, while in user mode will raise an Address Error exception. Virtual addresses 0x0000 0000 to 0x7FFF FFFF are translated to physical addresses 0x4000 0000 to 0xBFFF FFFF, respectively.

The upper 16-Mbyte area of kuseg (0x7F00 0000 to 0x7FFF FFFF) is reserved for on-chip resources and is not cacheable.

(2) Kernel mode

The kernel mode address space is treated as four virtual address segments. One of these, kuseg, is the same as the kuseg space in user mode; the remaining three are kernel segments kseg0, kseg1 and kseg2.

(a) kuseg

This is the same virtual address space available in user mode. Virtual addresses 0x0000 0000 to 0x7FFF FFFF are translated to physical addresses 0x4000 0000 to 0xBFFF FFFF, respectivery.

The upper 16-Mbyte area of kuseg (0x7F00 0000 to 0x7FFF FFFF) is reserved for on-chip resources and is not cacheable.

(b) kseg0

This is a 512 Mbyte segment spanning virtual addresses 0x8000 0000 to 0x9FFF FFFF. Fixed mapping of this segment is made to the 512 Mbyte physical address space from 0x0000 0000 to 1FFF FFFF. This area is cacheable.

(c) kseg1

This is a 512 Mbyte segment from virtual addresses 0xA000 0000 to 0xBFFF FFFF. Fixed mapping of this segment is made to the 512 Mbyte physical address space from 0x0000 0000 to 0x1FFF FFFF. Unlike kseg0, this area is not cacheable.

(d) kseg2

This is a 1 Gbyte linear address space from virtual address 0xC000 0000 to 0xFFFF FFFF. The upper 16-Mbyte area of kseg2 (0xFF00 0000 to 0xFFFF FFFF) is reserved for on-chip resources and is not cacheable. Of this reserved area, the 2 Mbytes from 0xFF20 0000 to 0xFF3F FFFF is intended for use as a debugging monitor area and testing.

Address mapping of the MMU is shown in Figure 5-3. The attributes of each segment are shown in Table 5-1.

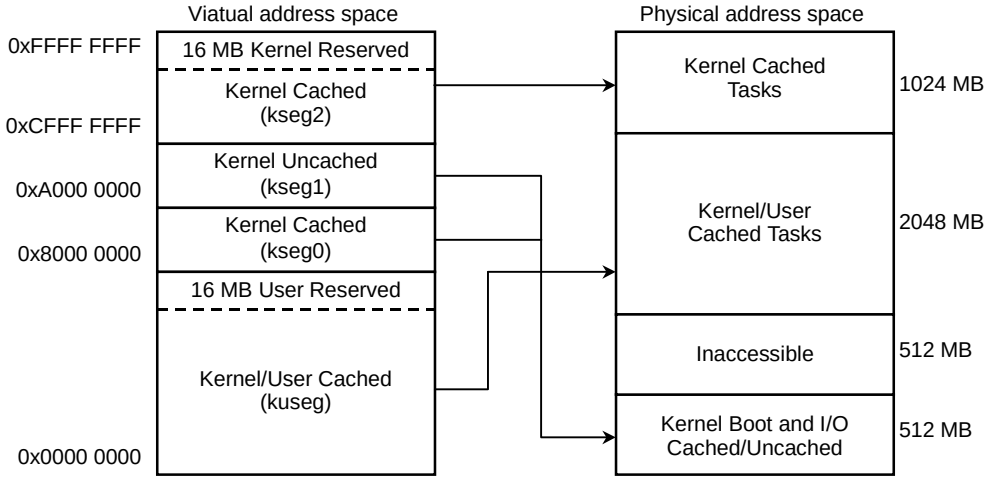


Figure 5-3. Internal MMU address mapping

Table 5-1. Internal MMU address mapping

Segment	Virtual address	Physical address	Cacheable	Mode
kseg2 (reserved)	0xFF00 0000-0xFFFF FFFF	0xFF00 0000-0xFFFF FFFF	Uncacheable	kernel
kseg2	0xC000 0000-0xFEFF FFFF	0xC000 0000-0xFEFF FFFF	Cacheable	kernel
kseg1	0xA000 0000-0xBFFF FFFF	0x0000 0000-0x1FFF FFFF	Uncacheable	kernel
kseg0	0x8000 0000-0x9FFF FFFF	0x0000 0000-0x1FFF FFFF	Cacheable	kernel
kuseg (reserved)	0x7F00 0000-0x7FFF FFFF	0xBF00 0000-0xBFFF FFFF	Uncacheable	kernel/user
kuseg	0x0000 0000-0x7EFF FFFF	0x4000 0000-0xBEFF FFFF	Cacheable	kernel/user

The upper 16 Mbytes of kuseg and kseg2 are reserved for on-chip resources (these areas are not cacheable.)

Of the reserved area in kseg2, the area from 0xFF20 0000 to 0xFF3F FFFF is a 2 Mbyte area reserved by Toshiba (intended for debug monitor and testing, etc.)

Chapter 6 Exception Processing

This chapter explains how exceptions are handled by the TX39 Processor Core, and describes the registers of the system control coprocessor CP0 used during exception handling.

6.1 Load and Store Instructions

When the TX39 Processor Core detects an exception, it suspends normal instruction execution. The processor goes from user mode to kernel mode so it can perform processing to handle the abnormal condition or asynchronous event.

The exception processing system in the TX39 Processor Core is designed for efficient handling of exceptions such as arithmetic overflows, I/O interrupts and system calls. When an exception is detected, all normal instruction execution is suspended. That is, execution of the instruction that caused the exception, as well as execution processing of instructions already in the pipeline is halted. Processing jumps directly to the exception handler designated for the raised exception.

When an exception is raised, the address at which execution should resume is loaded into the EPC (Exception Program Counter) register indicating where processing should resume after the exception has been handled. This will be the address of the instruction that caused the exception; or, if the instruction was supposed to be executed during a branch (delay slot instruction), the resume address will be that of the immediately preceding branch instruction.

Table 6-1. Exceptions defined for the TX39 Processor Core

Exception	Mnemonic	Cause
Reset	Reset †	This exception is raised when the reset signal is de-asserted after having been asserted.
UTLB Refill	UTLB	Reserved for an MMU with TLB.
TLB Refill	TLBL (load) TLBS (store)	Reserved for an MMU with TLB. Used for exception request by a memory access protection circuit. This exception is raised when access is attempted to a protected memory area.
TLB Modified	Mod	Reserved for an MMU with TLB.
Bus Error	IBE (instruction) DBE (data)	An external interrupt raised by a bus interface circuit. A Bus Error exception is raised when an event such as bus time-out, bus parity error, invalid memory address or invalid access type is detected, causing the bus-error pin to be asserted.
Address Error	AdEL (load) AdES (store)	This exception occurs with a misaligned access or an attempt to access a privileged area in user mode. Specific causes are: Load, store or instruction fetch of a word not aligned on a word boundary. Load or store of a halfword not aligned on a halfword boundary. Access attempt to kseg (including kseg0, kseg1, kseg2) in user mode.
Overflow	Ov	This exception is raised for a two's complement overflow occurring with an add or subtract instruction.
System Call	Sys	This exception is raised when a SYSCALL instruction is executed.
Breakpoint	Bp	This exception is raised when a BREAK instruction is executed.
Reserved Instruction	RI	This exception is raised when an undefined or reserved instruction is issued.
Coprocessor Unusable	CpU	This exception is raised when a coprocessor instruction is issued for a coprocessor whose CU bit in the corresponding Status register is not set.
Interrupt	Int	This exception is raised when an interrupt condition occurs.
Non-maskable Interrupt	Nmi †	This exception is raised at the falling edge of the non-maskable interrupt signal.
Debug Exception		Debug Single Step exception and Debug Breakpoint exception. See chapter 8 for detail

† Not an ExcCode mnemonic.

Table 6-2 shows the vector address of each exception and the values in the exception code (ExcCode) field of the Cause register.

Table 6-2. Exception vector addresses and exception codes

Exception	Mnemonic	Vector address †	Exception code
Reset	Reset	0xBFC0 0000 (0xBFC0 0000)	undefined
Non-maskable Interrupt	Nmi		undefined
UTLB Refill	UTLB(load)	0x8000 0000 (0xBFC0 0100)	TLBL(2)
	UTLB(store)		TLBS (3)
TLB Refill	TLBL (load)	0x8000 0080 (0xBFC0 0180)	TLBL (2)
	TLBS (store)		TLBS (3)
TLB Modified	Mod		Mod (1)
Bus Error	IBE (instruction)		IBE (6)
	DBE (data)		DBE (7)
Address Error	AdEL (load)		AdEL (4)
	AdES (store)		AdES (5)
Overflow	Ov		Ov (12)
System Call	Sys		Sys (8)
Breakpoint	Bp		Bp (9)
Reserved Instruction	RI		RI (10)
Coprocessor Unusable	CpU		CpU (11)
Interrupt	Int		Int (0)
Debug		0xBFC0 0200(0xBFC0 0200)	– ††

† The addresses shown here are virtual addresses. The address in parentheses applies when the Status register BEV bit is set to 1.

†† Cause of exception is shown in Debug register. See Chapter 8 for detail.

6.2 Exception Processing Registers

The system control coprocessor (CP0) has seven registers for exception processing, shown in Table 6-1.

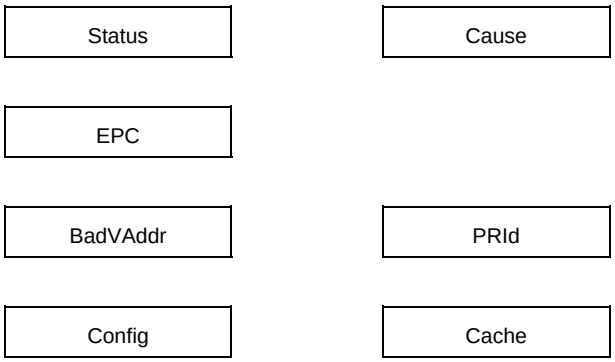


Figure 6-1. Exception processing registers

(a) Cause register
Indicates the nature of the most recent exception.

(b) EPC (Exception Program Counter) register
Holds the program counter at the time the exception occurred, indicating the address where processing is to resume after exception processing is completed.

(c) Status register
Holds the operating mode status (user mode or kernel mode), interrupt mask status, diagnostic status and other such information.

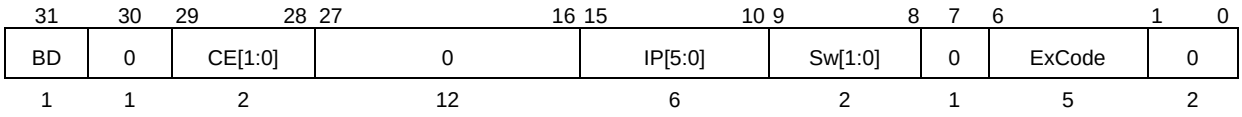
(d) BadVAddr (Bad Virtual Address) register
Holds the most recent virtual address for which a virtual address translation error occurred.

(e) PRId (Processor Revision Identifier) register
Shows the revision number of the TX39 Processor Core.

(f) Cache register
Controls the instruction cache (reserved) and the data cache auto-lock bits.

Note: In addition to the above exception processing registers, the CP0 registers include a Debug and DEPC register for use in debugging. See chapter 8 for detail.

6.2.1 Cause register (register no.13)



Bits	Mnemonic	Field name	Description	Value on Reset	Read/Write
31	BD	Branch Delay	Set to 1 when the most recent exception was caused by an instruction in the branch delay slot (executed during a branch).	Undefined	Read
29-28	CE	Coprocessor Error	Indicates the coprocessor unit number referenced when a Coprocessor Unusable exception is raised. (CE1, CE0) (0, 0) = coprocessor unit no. 0 (0, 1) = coprocessor unit no. 1 (1, 0) = coprocessor unit no. 2 (1, 1) = coprocessor unit no. 3	Undefined	Read
15-10	IP	Interrupt Pending	Indicates a held external interrupt. The status of the external interrupt signal line is shown.	Undefined	Read
9-8	Sw	Software Interrupt	Indicates a held software interrupt. This field can be written in order to set or reset a software interrupt.	Undefined	Read/Write
6-2	ExcCode	Exception Code	Holds an exception code (ExcCode) indicating the cause of an exception. The causes corresponding to each exception code are shown in Table 6-3.	Undefined	Read
30 27-16 7 1-0	0		Ignored on write; zero when read.	0	Read

For active interrupt signals, the corresponding IP bit is set to 1. For inactive interrupt signals, the IP bit is cleared to 0. The IP bit indicates the interrupt signal directly, independent of the Status register IEc bit and IntMask bit.

Figure 6-2. Cause register

Table 6-3. ExcCode field

ExcCode Field of Cause Register

No.	Mnemonic	Cause
0	Int	External interrupt
1	Mod	TLB Modified exception
2	TLBL	TLB Refill exception (load instruction or instruction fetch)
3	TLBS	TLB Refill exception (store instruction)
4	AdEL	Address Error exception (load instruction or instruction fetch)
5	AdES	Address Error exception (store instruction)
6	IBE	Bus Error (instruction fetch) exception
7	DBE	Bus Error (data load instruction or store instruction) exception
8	Sys	System Call exception
9	Bp	Breakpoint exception
10	RI	Reserved Instruction exception
11	CpU	Coprocessor Unusable exception
12	Ov	Arithmetic Overflow exception
13-31	-	reserved

6.2.2 EPC (Exception Program Counter) register (register no.14)

The EPC register is a 32-bit read-only register that stores the address at which processing should resume after an exception ends.

The address placed in this register is the virtual address of the instruction causing the exception. If it is an instruction to be executed during a branch (the instruction in the branch delay slot), the virtual address of the immediately preceding branch instruction is placed in the EPC instead. In this case, the BD bit in the Cause register is set to 1.

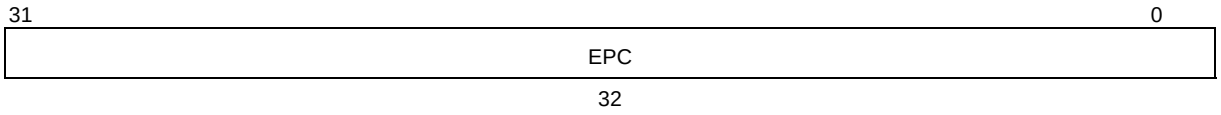


Figure 6-3. EPC register

6.2.3 Status register (register no.12)

This register holds the operating mode status (user mode or kernel mode), interrupt masking status, diagnosis status and similar information.

31 28	25	22	21	20	1916	15 8	7 6	5	4	3	2	1	0
CU[3:0]	0 RE 0	BEV	TS	Nml	0	IntMask Int[5:0] Sw[1:0]	0	KUo	IEo	KUp	IEp	KUc	IEc
4	2 1 2	1	1	1	4	8	2	1	1	1	1	1	1

Bits	Mnemonic	Field name	Description	Value on Reset	Read/Write
31-28	CU	Coprocessor Usability	The usability of the four coprocessors CP0 through CP3 is controlled by bits CU0 to CU3, with 1 = usable and 0 = unusable.	Undefined	Read/Write
25	RE	Reverse Endian	Setting this bit in user mode reverses the initial setting of the endian.	Undefined	Read/Write
22	BEV†	Bootstrap Exception Vector	When this bit is set to 1, if a UTLB Refill exception or general exception occurs, the alternate bootstrap vector (the vector address shown in parentheses in Table 6-2) is used.	1	Read/Write
21	TS	TLB Shutdown	Reserved	0	Read
20	Nml	Non-maskable Interrupt	This bit is set to 1 when a non-maskable interrupt occurs. Writing 1 to this bit clears it to 0.	0	Read/Write
15-8	IntMask	Interrupt Mask	These are mask bits corresponding to hardware interrupts Int5~0 and software interrupts Sw1~0. Here 1 = interrupt enabled and 0 = interrupt masked.	Undefined	Read/Write
5	KUo	Kernel/User Mode old	0 = kernel mode; 1 = user mode.	Undefined	Read/Write
4	IEo	Interrupt Enabled old	1 = interrupt enabled; 0 = interrupt masked.	Undefined	Read/Write
3	KUp	Kernel/User Mode previous	0 = kernel mode; 1 = user mode.	Undefined	Read/Write
2	IEp	Interrupt Enabled previous	1 = interrupt enabled; 0 = interrupt masked.	Undefined	Read/Write
1	KUc	Kernel/User Mode current	0 = kernel mode; 1 = user mode.	0	Read/Write
0	IEc	Interrupt Enabled current	1 = interrupt enabled; 0 = interrupt masked.	0	Read/Write
27-26 24-23 19-16 7-6	0		Ignored on write; 0 when read.	0	Read

† Used mainly for diagnosis and testing.

Figure 6-4. Status register

(1) CU (Coprocessor Usability)

The CU bits CU0 - CU3 control the usability of the four coprocessors CP0 through CP3. Setting a bit to 1 allows the corresponding coprocessor to be used, and clearing the bit to 0 disables that coprocessor. When an instruction for a coprocessor operation is used, the CU bit for that coprocessor must be set; otherwise a Coprocessor Unusable exception will be raised. Note that when the TX39 Processor Core is operating in kernel mode, the system control coprocessor CP0 is always usable regardless of how CU0 is set.

(2) RE (Reverse Endian)

The RE bit determines whether big endian or little endian format is used when the processor is initialized after a Reset exception. This bit is valid only in user mode; setting it to 1 reverses the initial endian setting. In kernel mode the endian is always governed by the endian signal set in a Reset exception. Since the RE bit status is undefined after a Reset exception, it should be initialized by the Reset exception handler in kernel mode.

(3) TS (TLB Shutdown)

The TS bit is always 0.

(4) BEV (Bootstrap Exception Vector)

If the BEV bit is set to 1, then the alternate vector address is used for bootstrap when a UTLB Refill exception or general exception occurs. If BEV is cleared to 0, the normal vector address is used. Immediately after a Reset exception, BEV is set to 1.

The alternate vector address allows an exception to be raised to invoke a diagnostic test prior to testing for normal operation of the cache and main memory systems.

(5) NmI (Non-maskable Interrupt)

This bit is set to 1 when a non-maskable interrupt is raised by the falling edge of the non-maskable interrupt signal. The bit is cleared to 0 by writing a 1 to it or when a Reset exception is raised.

(6) IntMask (Interrupt Mask)

The IntMask bits separately enable or mask each of six hardware and two software interrupts. Clearing a corresponding bit to 0 masks an interrupt, and setting it to 1 enables the interrupt. Note that clearing the IEo/IEp/IEc interrupt enable bits, explained below, has the effect of masking all interrupts.

(7) KUc/KUp/KUo (Kernel/User mode: current/previous/old)

The three bits KUc/KUp/KUo form a three-level stack, indicating the current, previous and old operating modes. For each bit, 0 indicates kernel mode and 1 is user mode. The way these bits are manipulated and used in exception processing is explained in 6.2.5 below. KUc is cleared to 0 when exception raises.

(8) IEc/IEp/IEo (Interrupt Enable: current/previous/old)

The three bits IEc/IEp/IEo form a three-level stack, indicating the current, previous and old interrupt enable status. For each bit, 0 means interrupts are disabled, and 1 means interrupts are enabled. The way these bits are manipulated and used in exception processing is explained in 6.2.5 below. IEc is cleared to 0 when exception raises.

6.2.4 Cache register (register no.7)

This register controls the cache lock function.

31 14	13	12	11	10	9	8	7 6 5 4 3 2 1 0
0	IALo	DALo	IALp	DALp	IALc	DALc	0
18	1	1	1	1	1	1	0

Bits	Mnemonic	Field name	Description	Value on Reset	Read/Write
13	IALo	Instruction Cache Lock(old)	1 = cache lock enable; 0 = cache lock disable	0	Read/Write
12	DALo	Data Cache Lock(old)	1 = cache lock enable; 0 = cache lock disable	0	Read/Write
11	IALp	Instruction Cache Lock(previous)	1 = cache lock enable; 0 = cache lock disable	0	Read/Write
10	DALp	Data Cache Lock(previous)	1 = cache lock enable; 0 = cache lock disable	0	Read/Write
9	IALc	Instruction Cache Lock(current)	1 = cache lock enable; 0 = cache lock disable	0	Read/Write
8	DALc	Data Cache Lock(current)	1 = cache lock enable; 0 = cache lock disable	0	Read/Write
31-14 7-0	0		Ignored on write; 0 when read.	0	Read

Figure 6-5. Cache register

(1) DALc/DALp/DALo (Data Cache Auto-Lock: current/previous/old)

The three bits DALc/DALp/DALo form a three-level stack, indicating the current, previous and old auto-lock status of the data cache. For each bit, 1 means the lock is in effect, and 0 means it is not. A Reset exception clears DALc, DALp and DALo to 0.

When the TX39 Processor Core responds to an exception, it saves the value of the current data cache auto-lock mode (DALc) in the previous mode bit (DALp), and that of the previous mode bit (DALp) in the old mode bit (DALo). The current data cache auto-lock mode (DALc) is cleared to 0, disabling the data cache lock function.

These bits are valid only when a cache with lock function is implemented.

(2) IALc/IALp/IALo (Instruction Cache Auto-Lock: current/previous/old)

The three bits IALc/IALp/IALo form a three-level stack, indicating the current, previous and old auto-lock status of the instruction cache. For each bit, 1 means the lock is in effect, and 0 means it is not. A Reset exception clears IALc, IALp and IALo to 0.

When the TX39 Processor Core responds to an exception, it saves the value of the current instruction cache auto-lock mode (IALc) in the previous mode bit (IALp), and that of the previous mode bit (IALp) in the old mode bit (IALo). The current instruction cache auto-lock mode (IALc) is cleared to 0, disabling the instruction cache lock function.

These bits are valid only when a cache with lock function is implemented.

6.2.5 Status register and Cache register mode bit and exception processing

When the TX39 Processor Core responds to an exception, it saves the values of the current operating mode bit (KUc) and current interrupt enabled mode bit (IEc) in the previous mode bits (KUp and IEp). It saves the values of the previous mode bits (KUp and IEp) in the old mode bits (KUo and IEo). The current mode bits (KUc and IEc) are cleared to 0, with the processor going to kernel mode and interrupts disabled.

Likewise, the TX39 Processor Core saves the values of the current data cache auto-lock mode bit (DALc) and current instruction cache auto-lock mode bit (IALc) in the previous mode bits (DALp and IALp). It saves the values of the previous mode bits (DALp and IALp) in the old mode bits (DALo and IALo). The current mode bits (DALc and IALc) are cleared to 0, disabling the data cache and instruction cache lock functions.

Provision of these three-level mode bits means that, before the software saves the Status register contents, the TX39 Processor Core can respond to two levels of exceptions. Figure 6-6 shows the Status register and Cache register save operations used by the TX39 Processor Core in exception processing.

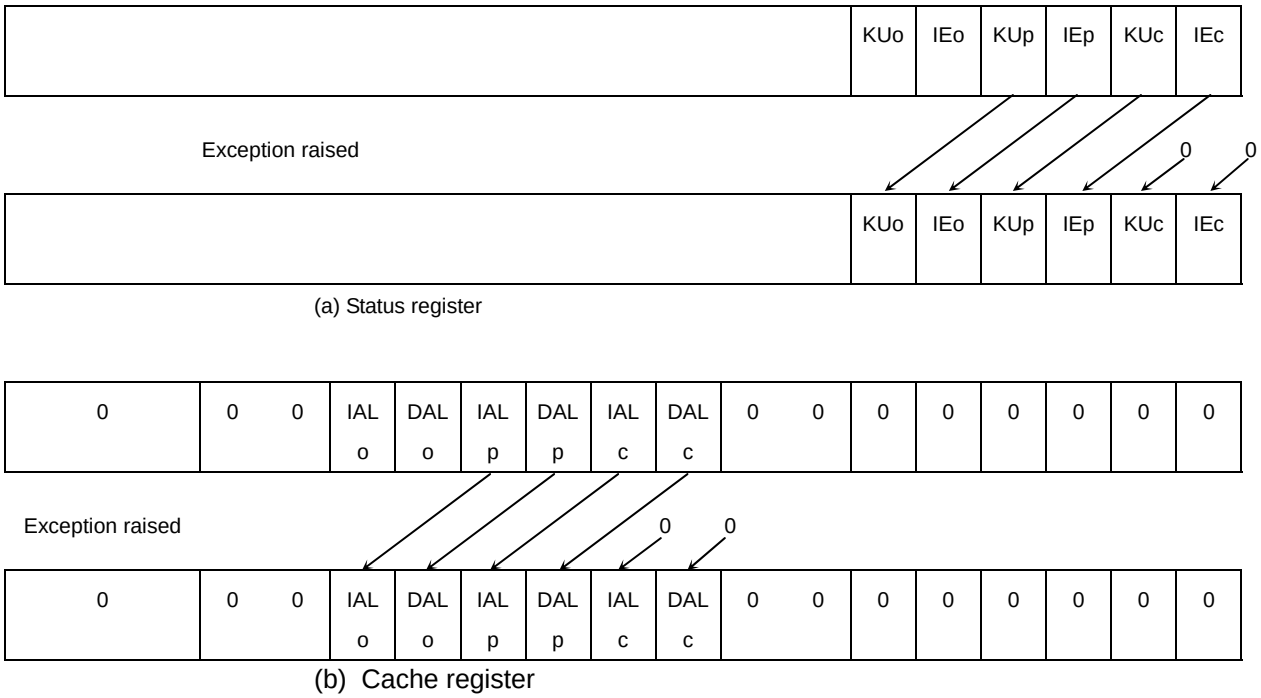


Figure 6-6. Status register and cache register when an exception is raised

After an exception handler has executed to perform exception processing, it must issue an RFE (Restore From Exception) instruction to restore the system to its previous status.

The RFE instruction returns control to processing that was in progress when the exception occurred. When a RFE instruction is executed, the previous interrupt enabled bit (IEp) and previous operating mode bit (KUp) in the Status register are copied to the corresponding current bits (IEc and KUc). The old mode bits (IEo and KUo) are copied to the corresponding previous mode bits (IEp and KUp). The old mode bits (IEo and KUo) retain their current values.

Likewise, the previous data cache auto-lock mode bit (DALp) and previous instruction cache auto-lock mode bit (IALp) in the Cache register are copied to the corresponding current bits (DALc and IALc). The old mode bits (DALo and IALo) are copied to the corresponding previous mode bits (DALp and IALp). The old mode bits (DALo and IALo) retain their current values.

Figure 6-7 shows how the RFE instruction works.

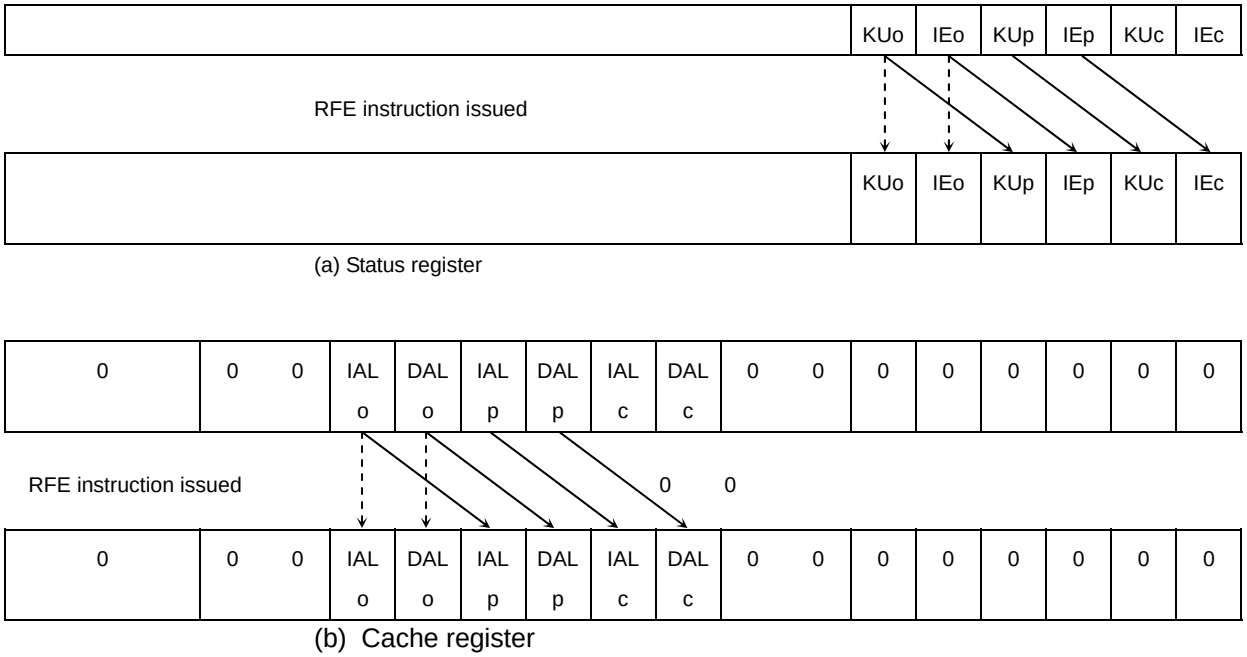


Figure 6-7. Status register and cache register when a RFE instruction is executed

6.2.6 BadVAddr (Bad Virtual Address) register (register no.8)

When an Address Error exception (AdEL or AdES) is raised, the virtual address that caused the error is saved in the BadVAddr register.

When a TLB Refill, TLB Modified or UTLB Refill exception is raised, the virtual address for which address translation failed is saved in BadVaddr.

BadVaddr is a read-only register.

Note: A bus error is not the same as an Address Error and does not cause information to be saved in BadVaddr.

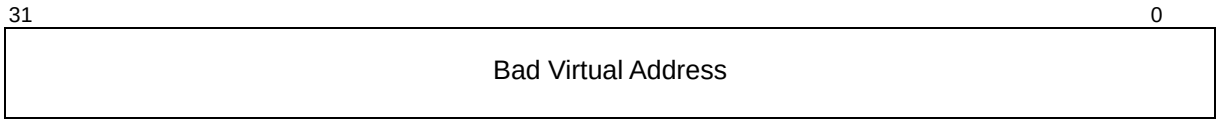
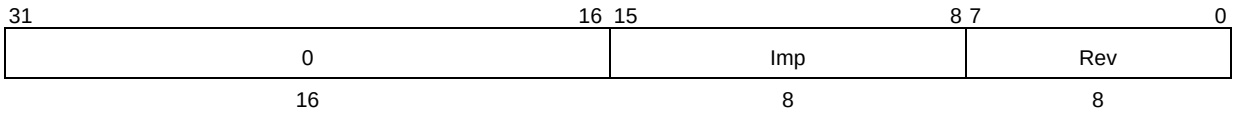


Figure 6-8. BadVaddr register

6.2.7 PRId (Processor Revision Identifier) register (register no.15)

PRId is a 32-bit read-only register, containing information concerning the implementation and revision level of the processor and system control coprocessor (CP0).

The register format is shown in Figure 6-9.



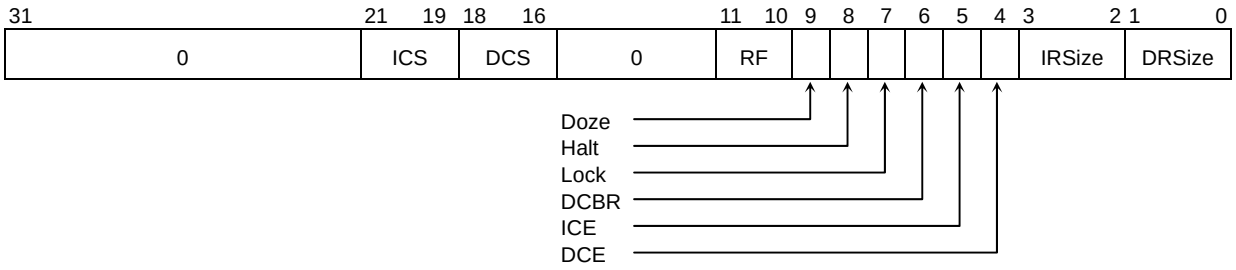
Bits	Mnemonic	Field name	Description	Value on Reset	Read/Write
15-8	Imp	Implementation number	TX39 Processor Core ID	0x22	Read
7-0	Rev	Revision identifier	TX39 Processor Core revision ID†	†	Read
31-16	0		Ignored on write; 0 when read.	0	Read

† Value is shown in product sheet.

Figure 6-9. PRId register

6.2.8 Config (Configuration) register (register no.3)

This register designates the TX39 Coprocessor Core configuration.



Bits	Mnemonic	Field name	Description	Value on Reset	Read/Write
21-19	ICS	Instruction Cache Size	Indicates the instruction cache size. 000: 1 KB; 001: 2 KB; 010: 4 KB; 011: 8 KB; 1xx : (reserved)	†	Read
18-16	DCS	Data Cache Size	Indicates the data cache size. 000: 1 KB; 001: 2 KB; 010: 4 KB; 011: 8 KB; 1xx : (reserved)	†	Read
11-10	RF	Reduced Frequency	Controls clock divider to determine reduced frequency provided externally from TX39 master clock. Please refer product's user manual for detail.	00	Read/Write
9	Doze	Doze††	Setting this bit to 1 puts the TX39 Processor Core in Doze mode and stalls the pipeline. This state is canceled by a Reset exception when a reset signal is received, or when cancelled by a non-maskable interrupt signal or interrupt signal that clears the Doze bit to 0. The Doze bit is cleared even if interrupts are masked. Data cache snoops are possible during Doze mode.	0	Read/Write

† implemented cache size

†† Operation is undefined when both Doze bit and Half bit are set to 1.

Figure 6-10. Config register (1/2)

Bits	Mnemonic	Field name	Description	Value on Reset	Read/Write
8	Halt††	Halt	Setting this bit to 1 puts the TX39 Processor Core in Halt mode. This state is canceled by a Reset exception when a reset signal is received, or when cancelled by a non-maskable interrupt signal or interrupt signal that clears the Halt bit to 0. The Halt bit is cleared even if interrupts are masked. Data cache snoops are not possible in Halt mode. Halt mode reduces power consumption to a greater extent than Doze mode.	0	Read/Write
7	Lock	Lock Config register	Setting this bit to 1 prevents further writes to the Config register. This bit is cleared to 0 by a Reset exception. If a store instruction is used to set other bits at the same time as the Lock bit, the other settings are valid.	0	Reset
6	DCBR	Data Cache Burst Refill	1:Indicates that the value in the DRSize field of the Config register should be used as the data cache refill size. 0:The data cache refill size is 1 word (4 bytes).	0	Read/Write
5	ICE	Instruction Cache Enable	Setting this bit to 1 enables the instruction cache.	1	Read/Write
4	DCE	Data Cache Enable	Setting this bit to 1 enables the data cache.	1	Read/Write
3-2	IRSize	Instruction Burst Refill Size	These bits designate the instruction cache burst refill size as follows. 00: 4 words (16 bytes) 01: 8 words (32 bytes) 10: 16 words (64 bytes) 11: 32 words (128 bytes)	00	Read/Write
1-0	DRSize	Data Burst Refill Size	These bits indicate the data cache burst refill size as follows. (This setting is valid only when the DCBR bit in the Config register is set to 1.) 00: 4 words (16 bytes) 01: 8 words (32 bytes) 10: 16 words (64 bytes) 11: 32 words (128 bytes)	00	Read/Write
31-22, 15-12	0		Ignored on write; 0 when read	0	Read

Note: † After modifications to DCBR, ICE, DCE, IRSize or DRSize, the new cache configuration takes effect after completion of the currently executing bus operation (cache refill).

†† Operation is undefined when both Doze bit and Half bit are set to 1.

Figure 6-10. Config register (2/2)

6.3 Exception Details

6.3.1 Memory location of exception vectors

Exception vector addresses are stored in an area of kseg0 or kseg1.

The vector address of the Reset and NmI exceptions is always in a non-cacheable area of kseg1. Vector addresses of the other exceptions depend on the Status register BEV bit. When BEV is 0 the other exceptions are vectored to a cacheable area of kseg0.

When BEV is 1, all vector addresses are in a non-cacheable area of kseg1.

Exception	Vector address (virtual address)	
	BEV bit = 0	BEV bit = 1
Reset, NmI	0xBFC0 0000	0xBFC0 0000
UTLB Refill	0x8000 0000	0xBFC0 0100
Debug	0xBFC0 0200	0xBFC0 0200
Other	0x8000 0080	0xBFC0 0180

Exception	Vector address (physical address)	
	BEV bit = 0	BEV bit = 1
Reset, NmI	0x1FC0 0000	0x1FC0 0000
UTLB Refill	0x0000 0000	0x1FC0 0100
Debug	0x1FC0 0200	0x1FC0 0200
Other	0x0000 0080	0x1FC0 0180

The virtual address 0xBFC0 0200 is used as the vector address for Debug exceptions. Details are given in Chapter 8.

6.3.2 Address Error exception

- Causes
  - Attempting to load, fetch or store a word not aligned on a word boundary.
  - Attempting to load or store a halfword not aligned on a halfword boundary.
  - Attempting to access kernel mode address space kseg while in user mode.
- Exception mask

  The Address Error exception is not maskable.
- Applicable instructions

  LB, LBU, LH, LHU, LW, LWL, LWR, SB, SH, SW, SWL, SWR.
- Processing
  - The common exception vector (0x8000 0080) is used.
  - ExcCode AdEL(4) or AdES(5) in the Cause register is set depending on whether the memory access attempt was a load or store.
  - When the Address Error exception is raised, the misaligned virtual address causing the exception, or the kernel mode virtual address that was illegally referenced, is placed in the BadVAddr register.
  - The EPC register points to the address of the instruction causing the exception. If, however, the affected instruction was in the branch delay slot (for execution during a branch), the immediately preceding branch instruction address is retained in the EPC register and the Cause register BD bit is set to 1.

6.3.3 Breakpoint exception

- Causes
  - Execution of a BREAK command.
- Exception mask

  The Breakpoint exception is not maskable.
- Applicable instructions

  BREAK
- Processing
  - The common exception vector (0x8000 0080) is used.
  - BP(9) is set for ExcCode in the Cause register.

    The EPC register points to the address of the instruction causing the exception. If, however, the affected instruction was in the branch delay slot (for execution during a branch), the immediately preceding branch instruction address is retained in the EPC register and the Cause register BD bit is set to 1.
- Servicing

  When a Breakpoint exception is raised, control is passed to the designated handling routine. The unused bits of the BREAK instruction (bits 26 to 6) can be used to pass information to the handler. When loading the BREAK instruction contents, the instruction pointed to by the EPC register is loaded. Note that when the Cause register BD bit is set to 1 (when the BREAK instruction is in the branch delay slot), it is necessary to add +4 to the EPC register value.

  In returning from the exception handler, +4 must be added to the address in the EPC register to avoid having the BREAK instruction executed again. If the Cause register BD bit is set to 1 (when the immediately preceding instruction was a branch instruction), the branch instruction must be interpreted and set in the EPC register so that the return from the exception handler will be made to the branch destination of the immediately preceding branch instruction.

6.3.4 Bus Error exception

- Causes
  - This exception is raised when a bus error signal is input to the TX39 Processor Core during a memory bus cycle.
    This occurs during execution of the instruction causing the bus error. The memory bus cycle ends upon notification of a bus error. When a bus error is raised during a burst refill, the following refill is not performed.
    A bus error request made by asserting a bus error signal will be ignored if the TX39 Processor Core is executing a cycle other than a bus cycle. It is therefore not possible to raise a Bus Error exception in a write access using a write buffer. A general interrupt must be used instead.
- Exception mask

  The Bus Error exception is not maskable.
- Applicable instructions

  LB, LBU, LH, LHU, LW, LWL, LWR, SB, SH, SW, SWL, SWR; any fetch instruction.
- Processing
  - The common exception vector (0x8000 0080) is used.
  - IBE(6) or DBE(7) is set for ExcCode in the Cause register.
  - The EPC register will have an undefined value except in the following cases.

    (1) A SYNC instruction follows execution of a load instruction.

    (2) An instruction that follows execution of a load instruction while one-word data cache refill size is in effect, or that follows a load instruction that loads data from an uncached area, needs to use the result of the load.

    In the above case, since the load delay slot instruction will stall until the end of the read operation, the EPC will contain the load delay slot address when a bus error occurs.

    Note: When the destination address of a load instruction is r0 and the following instruction uses r0, the TX39 Processor Core will not stall.
  - The TX39 Processor Core stores the Status register bits KUp, IEp, KUc and IEc in KUo, IEo, KUp and IEp, respectively, and clears the KUc and IEc bits to 0. And, the TX39 Processor Core stores Cache register bits DALp, IALp, DALc and IALc in DALo, IALo, DALp and IALp, respectively, and clears the DALc and IALc bits to 0.
  - The TX39 Processor Core does not store the cache block in cache memory if the block includes a word for which a bus error occurred.
  - When a bus error occurs with a load instruction, the destination register value will be undefined.
  - In the following cases, a Bus Error exception may be raised even though the instruction causing the bus error did not actually execute.

    (1) When a bus error occurs during an instruction cache refill, but the instruction sequence is changed due to a jump/branch instruction in the instruction stream, the instruction at the address where the bus error occurred may not actually execute.

    (2) When a bus error occurs in a data cache block refill, the data at the address where the bus error occurred may not actually have been used.
- Servicing

  The address in the EPC register is undefined. In some cases it is not possible to determine the address where a bus error actually occurred. If this address is required, then external hardware must be used to store addresses. Using such an external circuit will allow you to retain the address where a bus error occurs.

6.3.5 Coprocessor Unusable exception

- Causes
  - Attempting to execute a coprocessor CPz instruction when its corresponding CUz bit in the Status register is cleared to 0 (coprocessor unusable).
  - In user mode, attempting to execute a CP0 instruction when the CU0 bit is cleared to 0. (In kernel mode, an exception is not raised when a CP0 instruction is issued, regardless of the CU0 bit setting.)
- Exception mask

  The Coprocessor Unusable exception is not maskable.
- Applicable instructions

  Coprocessor instructions: LWCz, SWCz, MTCz, MFCz, CTCz, CFCz, COPz, BCzT, BCzF, BCzTL, BCzFL

  Coprocessor 0 instructions: MTC0, MFC0, RFE, COP0
- Processing
  - The common exception vector (0x8000 0080) is used.
  - CpU(11) is set for ExcCode in the Cause register.
  - The coprocessor number referred to at the time of the exception is stored in the Cause register CE (Coprocessor Error) field.
  - The EPC register points to the address of the instruction causing the exception. If, however, that instruction is in the branch delay slot (for execution during a branch), the immediately preceding branch instruction address is retained in the EPC register and the Cause register BD bit is set to 1.

6.3.6 Interrupts

- Causes
  - An Interrupt exception is raised by any of eight interrupts (two software and six hardware). A hardware interrupt is raised when the interrupt signal goes active. A software interrupt is raised by setting the Sw1 or Sw0 bits in the Cause register.
- Exception mask
  - Each of the eight interrupts can be masked individually by clearing its corresponding bit in the IntMask field of the Status register.
  - All interrupts can be masked by clearing the Status register IE bit to 0.
- Processing
  - The common exception vector (0x8000 0080) is used.
  - Int(0) is set for ExcCode in the Cause register.
  - The Cause register IP and Sw fields indicate the status of current interrupt requests. It is possible for more than one of these bits to be set or for none to be set (when an interrupt is asserted and then de-asserted before the register is read).

    Notes: You should disable interrupts when executing the RFE instruction because the Status register contents will be undefined when an interrupt occurs while executing the RFE instruction.
- Servicing

  An interrupt condition set by one of the two software interrupts can be cleared by clearing the corresponding Cause register bit (Sw1 or Sw0) to 0.

  For hardware-generated interrupts, the condition can only be cleared by determining and handling the source of the corresponding active signal.

  The IP field indicates the status of interrupt signals regardless of the Status register IntMask field. The cause of an interrupt should be determined from a logical AND of the IP and IntMask fields.
  - The EPC register points to the address of the instruction causing an exception. If, however, that instruction is in the branch delay slot (for execution during a branch), the immediately preceding branch instruction address is retained in the EPC register and the Cause register BD bit is set to 1.

6.3.7 Overflow exception

- Causes
  - A two's complement overflow results from the execution of an ADD, ADDI or SUB instruction.
- Exception mask

  The Overflow exception is not maskable.
- Applicable instructions

  ADD, ADDI, SUB
- Processing
  - The common exception vector (0x8000 0080) is used.
  - Ov(12) is set for ExcCode in the Cause register.
  - The EPC register points to the address of the instruction causing the exception. If, however, that instruction is in the branch delay slot (for execution during a branch), the immediately preceding branch instruction address is retained in the EPC register and the Cause register BD bit is set to 1.

6.3.8 Reserved Instruction exception

- Causes
  - Attempting to execute an instruction whose major opcode (bits 31~26) is undefined, or a special instruction whose minor opcode (bits 5~0) is undefined.
  - Attempting to execute reserved instruction (LWCz and SWCz).
- Exception mask

  The Reserved Instruction exception is not maskable.
- Processing
  - The common exception vector (0x8000 0080) is used.
  - RI(10) is set for ExcCode in the Cause register.
  - The EPC register points to the address of the instruction causing the exception. If, however, that instruction is in the branch delay slot (for execution during a branch), the immediately preceding branch instruction address is retained in the EPC register and the Cause register BD bit is set to 1.

6.3.9 Reset exception

- Causes
  - The reset signal in the TX39 Processor Core is asserted and then de-asserted.
- Exception mask

  The Reset exception is not maskable.
- Processing
  - A special interrupt vector (0xBFC0 0000) that resides in an uncached area is used. It is therefore not necessary for hardware to initialize cache memory in order to process this exception.
  - The contents of all registers in the TX39 Processor Core become undefined. See the description of each register earlier in this section for details.
  - All data cache and instruction cache valid bits are cleared to 0, as are all data cache lock bits.
  - If a Reset exception is raised during a bus cycle, the bus cycle is immediately ended and the reset is allowed to proceed.

6.3.10 System Call exception

- Causes
  - Execution of an TX39 Processor Core SYSCALL instruction.
- Exception mask

  The System Call exception is not maskable.
- Applicable instructions

  SYSCALL
- Processing
  - The common exception vector (0x8000 0080) is used.
  - Sys(8) is set for ExcCode in the Cause register.
  - The EPC register points to the address of the instruction causing the exception. If, however, that instruction is in the branch delay slot (for execution during a branch), the immediately preceding branch instruction address is retained in the EPC register and the Cause register BD bit is set to 1.

6.3.11 Non-maskable interrupt

- Causes
  - Occurs at the falling edge of the non-maskable interrupt signal.
- Exception mask

  The Non-maskable exception is not maskable. It is raised regardless of the Status register IEc bit setting.
- Processing
  - The same special interrupt vector as for Reset (0xBFC0 0000), residing in an area that is not cached, is used. It is therefore not necessary for hardware to initialize cache memory in order to process this exception.
  - Unlike the Reset exception, here the Status register NmI bit is set.
  - As with other exceptions (except for the Reset exception), the NmI exception occurs at an instruction boundary. If a Non-maskable interrupt occurs during a bus cycle, interrupt processing waits until the bus cycle has ended.
  - All register contents are retained except for the following.
    - The EPC register points to the address of the instruction causing the exception. If, however, that instruction is in the branch delay slot (for execution during a branch), the immediately preceding branch instruction address is retained in the EPC register and the Cause register BD bit is set to 1.
    - The Status register NmI bit is set to 1.
    - The Config register Halt bit and Doze hit are cleared to 0.
    - The Cause register CE bit and ExcCode are undefined.

6.4 Priority of Exceptions

More than one exception may be raised for the same instruction, in which case only the exception with the highest priority is reported. The TX39 Processor Core instruction exception priority is shown in Table 6-4.

See chapter 8 for the priority of debug exceptions.

Table 6-4. Priority of Exceptions

Priority	Exception (Mnemonic)
High	Reset
↑	IBE (instruction fetch)
	DBE (data access)
	NmI
	AdEL (instruction fetch)
	TLBL (instruction fetch)
	CpU
	Ov, Sys, Bp, RI
	AdEL (load instruction)
	AdES (store instruction)
	TLBL (data load)
	TLBS (store instruction)
↓	Mod
Low	Int

6.5 Return from Exception Handler

An example of returning from an exception handler is shown below.

```
MFC0    r27, EPC (store return address in general register)
JR      r27      (jump to return address)
RFE              (execute RFE instruction in branch delay slot)
```

Chapter 7 Caches

The TX39 Processor Core is equipped with separate on-chip caches for data and instructions. These caches can be configured in a variety of sizes as required by the user system.

Note: Currently only the cache configuration described below is supported. It consists of a 4 Kbyte instruction cache and 1 Kbyte data cache.

7.1 Instruction Cache

The instruction cache has the following specifications.

- Cache size: 4 Kbytes (Config register ICS bits = 010)
- Direct mapping
- Block (line) size: 4 words (16 bytes)
- Physical cache
- Burst refill size: Choice of 4/8/16/32 words (set in Config register)
- All valid bits are cleared (made invalid) by a Reset exception

Note: The lock function is not currently supported for the instruction cache. Cache register bits IALc, IALp and IALo do not affect the instruction cache.

Figure 7-1 shows the instruction cache configuration.

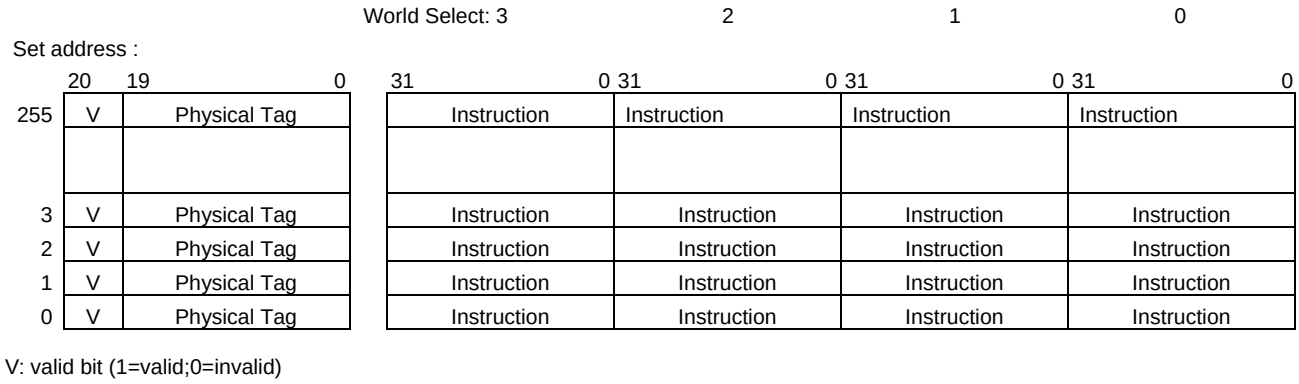


Figure 7-1. Instruction cache configuration

Figure 7-2 shows the instruction cache address field.

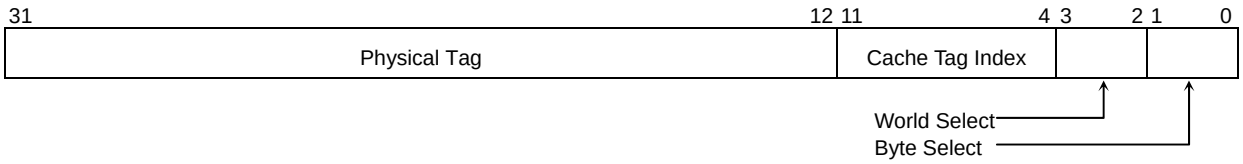


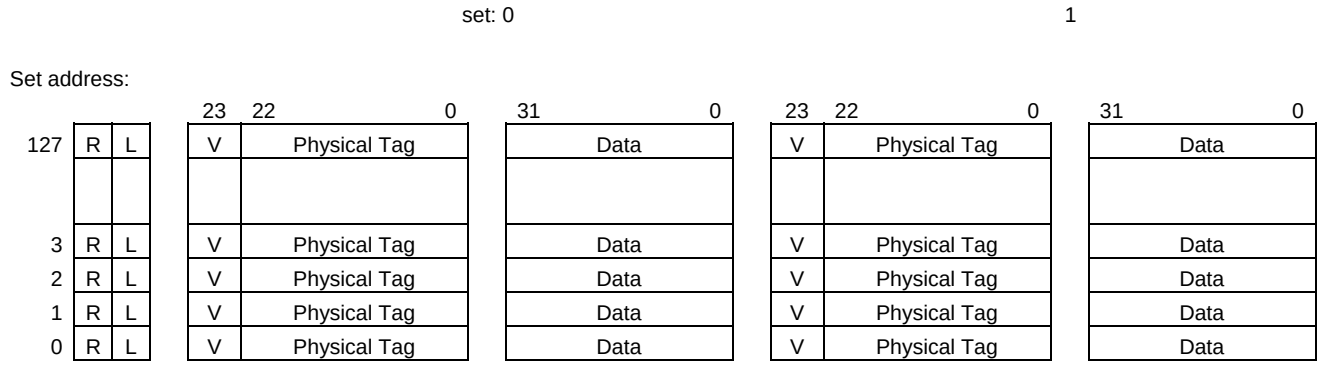
Figure 7-2. Instruction cache address field

7.2 Data Cache

The data cache has the following specifications.

- Cache size: 1 Kbyte (Config register DCS bits = 000)
- Two-way set-associative
- Replace algorithm: LRU (Least Recently Used)
- Block (line) size: 1 word (4 bytes)
- Write-through
- Physical cache
- Refill size: Choice of size 1/4/8/16/32 words (set in Config register)
- Byte-writable
- All valid bits and lock bits cleared by a Reset exception
- Lock function

Figure 7-3 shows the data cache configuration.



R: LRU replace bit(indicates next set to which replacement will be directed; when lock bit is set to 1, indicates this set is not locked)

L: Lock bit(when set to 1, if R bit is 1, set 0 is locked, and if R bits 0, set 1 is locked; when cleared to 0, lock function is disabled)

V: valid bit(1=valid; 0 = invalid)

Figure 7-3. Data cache configuration

Figure 7-4 shows the data cache address field.

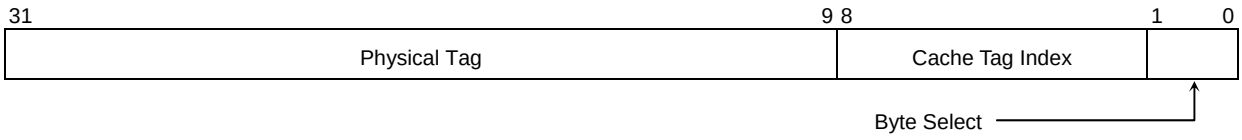


Figure 7-4. Data cache address field

When a data store misses, the data is stored to main memory only, not to the cache (no write allocate).

The data cache can be written in individual bytes. (When a byte or halfword store is used, there is no read-modify-write.)

7.2.1 Lock function

The lock function can be used to route critical data to one data cache set. Data is not replaced when the lock bit is set.

(1) Lock bit setting

Setting the Cache register DALc bit enables the data cache lock function. When data in a line is accessed, the lock bit for that line is set and data in the line can no longer be replaced. If a store miss occurs, the store data is not written to the cache and will therefore not be locked.

Note: When a block refill takes place, the size of data locked in the cache is the same as the block refill size.

The Cache register DALc bit can be set at the head of a subroutine or the like, thereby locking into the cache the data accessed by the subroutine. The lock function can be disabled by clearing the DALc bit. This does not clear the lock bits of individual lines.

(2) Operation during lock

When the lock bit is set for a line, only data in the set indicated by the LRU replace bit (R) can be replaced. A write access to a locked line takes place only to cache memory, without affecting main memory. When a lock has been established by the lock function, store operations can write to memory.

The Cache register lock bits form a three-layer stack consisting of DALc, DALp and DALo. If an exception is raised while the lock function is in effect, the stack is pushed (the DALc and DALp bit values are saved in DALp and DALo, respectively) and DALc is cleared, disabling the lock function. This is to prevent inadvertent locking of data used by the exception handler. After the handler has finished processing, a RFE instruction is executed, popping the stack (the DALo and DALp bit values are restored to DALp and DALc) and refurring the status to that prior to the exception.

(3) Lock bit clearing

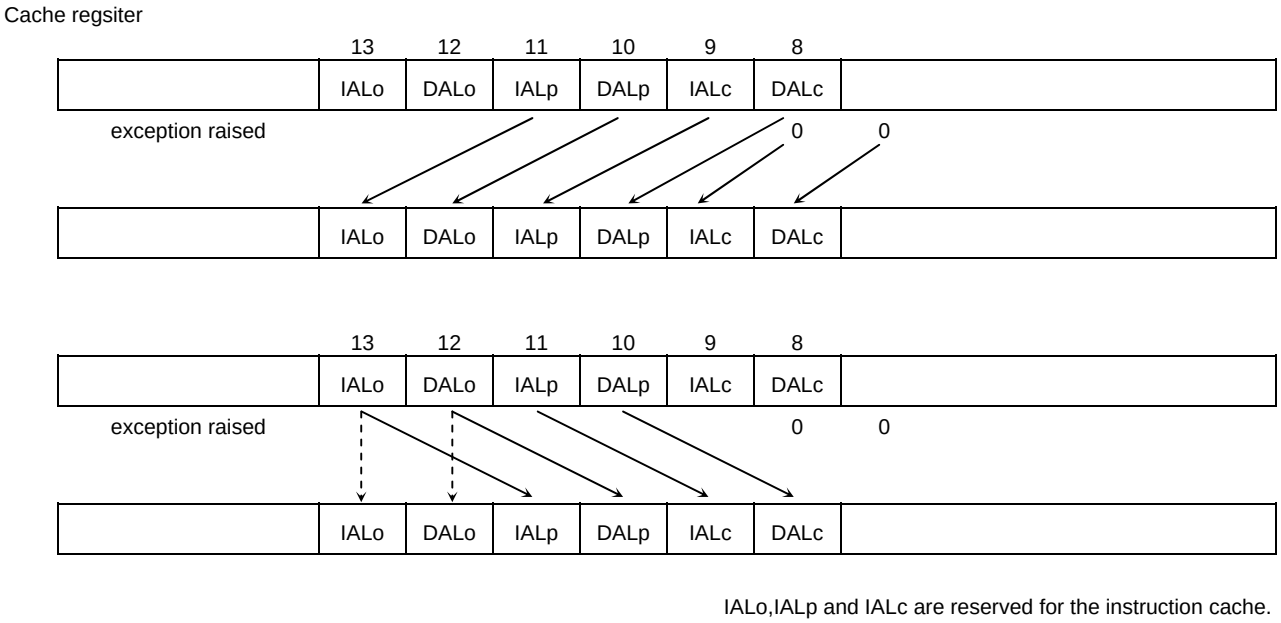


Figure 7-5. Auto-lock bits

The lock bit for an entry is cleared using the CACHE instruction IndexLockBitClear. Clearing the lock bit disables the lock function.

Clear the lock bit as follows when data written to a locked line should be stored in main memory.

1) Read the locked data from cache memory

2) Clear the lock bit

3) Store the data that was read

7.3 Cache Test Function

(1) Cache disabling

The Config register bits ICE (Instruction Cache Enable) and DCE (Data Cache Enable) are used to enable and disable the instruction cache and data cache, respectively.

When a cache is disabled, all cache accesses are misses and there is no refill (nor is there any burst bus cycle; this is the same as accessing a non-cacheable area). The valid bit (V) for each entry cannot be modified.

(2) Cache flushing

Both the instruction cache and data cache are flushed when a Reset exception is raised (all valid bits are cleared to 0).

The instruction cache is flushed by the CACHE instruction IndexInvalidate. The data cache is flushed by the CACHE instruction HitInvalidate.

Note: An instruction cache IndexInvalidate operation is possible only when the instruction cache is disabled (Config register ICE bit = 0).

Additional explanation: As a sure way of disabling the instruction cache, streaming should be stopped by inserting a branch instruction after MTC0, as shown below.

Example:

```
MTC0    Rn, Config      (clear ICE to 0)
J       L1              (branch to L1; stop streaming)
NOP                     (branch delay slot)
L1: CACHE IndexInvalidate, offset (base)
```

(3) Lock bit clearing

The data cache lock bit is cleared by a Reset exception.

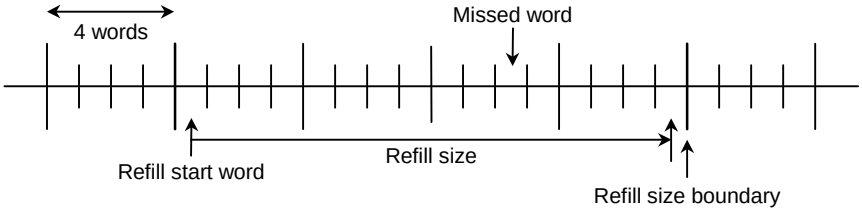
It can also be cleared by the CACHE instruction IndexLockClear. (The IndexLockClear instruction is reserved for clearing instruction cache lock bits.)

7.4 Cache Refill

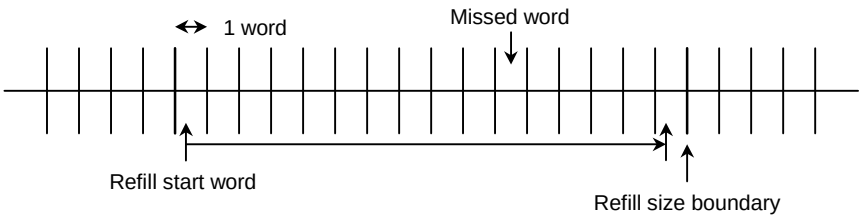
A physical cache line in the TX39 Processor Core comprises 4 words for the instruction cache and 1 word for the data cache. The refill size can be designated independently of the line size. The refill size can be 4/8/16/32 words for the instruction cache, and 1/4/8/16/32 words for the data cache. In a burst read operation, data or instructions of the designated refill size are read. However, when the data cache refill size is set to one word (Config register DCBR = 0), a single read operation is performed.

Both caches are refilled from the head of the refill boundary.

Regardless of the refill size, tags are updated one physical line at a time.



(a) Instruction cache



(b) Data cache

Figure 7-6. Cache refill

Additional explanation: If an instruction changing the cache configuration (MTC0 to modify the Config register, or any CACHE instruction) is executed during a refill cycle, the new configuration takes effect after the refill cycle in progress is completed. Note that instruction cache invalidation is possible only while the instruction cache is disabled.

7.5 Cache Snoop

The TX39 Processor Core has a bus arbitration function that releases bus mastership to an external bus master. Consistency between cache memory and main memory could deteriorate when an external bus master has write access to main memory. The purpose of the cache snoop function is to maintain this data consistency.

When the TX39 Processor Core releases the bus, the bus cycle is snooped by an external bus master. If an address access by the external bus master matches an address stored in the on-chip data cache (cache hit), the valid bit (V) for that cache data is cleared to 0, invalidating it.

Locked data cannot be invalidated, however, even when a hit occurs in a snoop operation.

After a cache block has been invalidated in a snoop, the LRU bit points to the invalidated cache set.

The lock bit is not changed as the result of a snoop.

Note: A snoop is possible even when the data cache is disabled.

Chapter 8 Debugging Functions

The TX39 Processor Core has the following support functions for debugging that have been added to the R3000A instruction base. They are independent of the R3000A architecture, which makes them transparent to user programs.

The real-time debugging system is supported by a third party.

Debug exceptions (Single Step, Break Instruction)

Additional register (DEPC) for holding the PC value when a debug exception occurs

Additional register (Debug) for controlling debug exceptions

Additional instruction (DERET) for return from a debug exception

8.1 System Control Processor (CP0) Registers

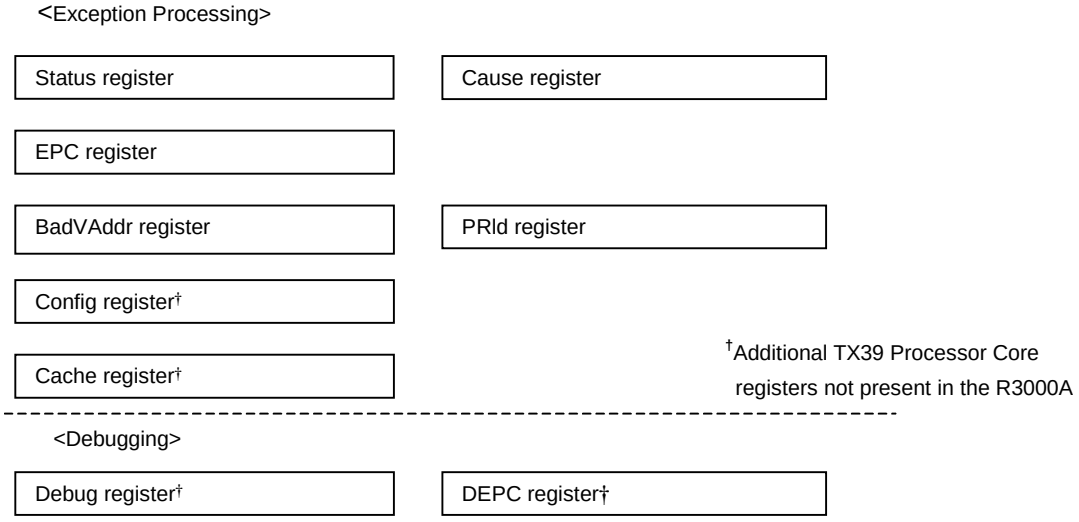


Figure 8-1. CP0 Registers

When a debug exception occurs, only registers Debug and DEPC are updated. The registers accessed by user application programs (general-purpose registers, Status, Cause, EPC, BadVAddr, PRId, Config and Cache) retain their values.

The CP0 registers are listed in Figure 8-1.

Table 8-1. List of system control coprocessor (CP0) registers

No	Mnemonic	Description
0	-	(reserved)
1	-	(reserved)
2	-	(reserved)
3	Config†	Hardware configuration
4	-	(reserved)
5	-	(reserved)
6	-	(reserved)
7	Cache†	Cache lock function
8	BadVAddr	Last virtual address triggering error
9	-	(reserved)
10	-	(reserved)
11	-	(reserved)
12	Status	Information on mode, interrupt enabled, diagnostic status
13	Cause	Indicates nature of last exception
14	EPC	Exception program counter
15	PRId	Processor revision ID
16	Debug††	Debug exception control
17	DEPC††	Program counter for debug exception
18 | 31	-	(reserved)

† Additional TX39 Processor Core register not present in R3000A.

†† Additional TX39 Processor Core register for Debug function not present in R3000A.

(1) DEPC (Debug Exception Program Counter) register (register no.17)

The DEPC register holds the address where processing is to resume after the debug exception has been taken care of.

(Note: DEPC is a read/write register.)

The address that goes in the DEPC register is the virtual address of the instruction that caused the debug exception. If that instruction is in the branch delay slot, the virtual address of the immediately preceding branch or jump instruction goes in this register and Debug register DBD bit is set to 1.

Execution of the DERET instruction causes a jump to the DEPC address.

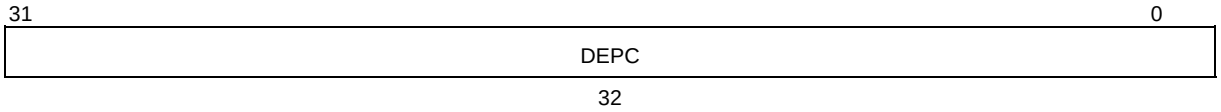


Figure 8-2. DEPC register

(Note: When a debug exception occurs, EPC retains its value.)

(2) Debug register (register no.16)

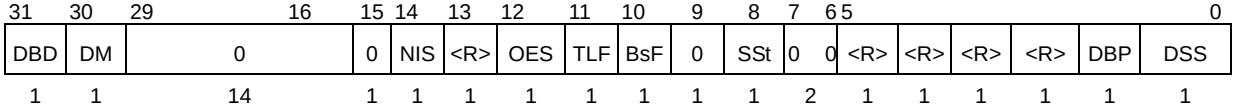


Figure 8-3. Debug register

SSt and BsF are read/write bits; all other bits are read-only, to which writes are ignored.

- DBD (Debug Branch Delay)

  When a debug exception occurs while the instruction in the branch delay slot is executing, this bit is set to 1.

- DM (Debug Mode) (0 at reset)

  This bit indicates whether or not a debug exception handler is running. It is set to 1 when a debug exception is raised, and cleared to 0 upon return from the exception.

  0: Debug handler not running

  1: Debug handler running

- NIS (Non-maskable Interrupt Status)

  This bit is set to 1 when a Non-maskable interrupt occurs at the same time as a debug exception. In this case the Status, Cause, EPC and BadVAddr registers assume their usual status after the occurrence of a Non-maskable interrupt, but the address in DEPC is not the non-maskable interrupt exception vector address (0xBFC0 0000).

  Instead, 0xBFC0 0000 is put in DEPC by the debug exception handler software, after which processing returns directly from the debug exception to the Non-maskable interrupt handler.

- OES (Other Exceptions Status)

  This bit is set to 1 when an exception other than reset, NmI or UTLB Refill occurs at the same time as a debug exception. In this case the Status, Cause, EPC and BadVAddr registers assume their usual status after the occurrence of such an exception, but the address in DEPC will not be the other exception vector address. Instead, 0xBFC0 0180 (if the Status register BEV bit is 1) or 0x8000 0080 (if BEV is 0) is put in DEPC by the debug exception handler software, after which processing returns directly from the debug exception to the other exception handler.

  (Note: Only one of bits NIS, or OES is set, according to the priority of exceptions.)

- TLF (TLB Exception Flag)

  This bit is set to 1 when a TLB-related exception (TLB Refill, UTLB Refill, Mod) occurs for the immediately preceding load or store instruction while a debug exception handler is running (DM bit = 1).

  (Note: A check should be made as to whether a TLB-related exception has occurred or not each time access is made to the user area data.)

- BsF (Bus Error Exception Flag)

  This bit is set to 1 when a bus error exception occurs for a load or store instruction while a debug exception handler is running (DM bit = 1). It is cleared by writing 0 to it.

- SSt (Single Step) (0 at reset)

  This bit indicates whether the single step debug function is enabled (set to 1) or disabled (cleared to 0). The function is disabled when the DM bit is set to 1, i.e., while a debug exception handler is running. This bit is a read/write bit.

- DBp (bit 1)

  Set to 1 to indicate a Debug Breakpoint exception.

- DSS (bit 0)

  Set to 1 to indicate a Single Step exception.

  DBp and DSS bits indicate the most recent debug exception. Each bit represents one of the two debug exceptions and is set to 1 accordingly when that exception occurs.

Note: DSS has a higher priority than DBp, since they occur in the pipeline E stage. For this reason DSS and DBp are not raised at the same time.

- 0

  Ignored when written; returns 0 when read.

- <R>

  Reserved. Undefined value.

8.2 Debug Exceptions

(1) Types of debug exceptions

There are two debug exceptions, as follows.

1) Debug Single Step (DSS)

When the Debug register SS bit is set, this exception is raised each time an instruction is executed.

2) Debug Breakpoint (DBp)

This exception is raised when an SDBBP instruction is executed.

Note: Since the real-time debugging system function has priority, the above two functions are disabled when the real-time debugging system is used.

(2) Debug exception handling

i) Raising a debug exception

- DEPC register updates

  DEPC: The address where the exception was raised is put in this register.

- Debug register updates

  DBD: Set to 1 when the exception was raised for an instruction in the branch delay slot.

  DM: Set to 1.

  DSS, DBp: Set to 1 if the corresponding exception was raised.

  NIS: Set to 1 if a Non-maskable interrupt occurred at the same time as the debug exception.

  OES: Set to 1 if another exception (other than reset, NmI, or UTLB Refill) was raised at the same time as the debug exception.

- Branching to a debug exception handler

  PC: 0xBFC0 0200

  (Note: Registers other than DEPC and Debug retain their values.)

- Masking of exceptions and interrupts in a debug exception handler

  A load or store instruction for which a TLB-related exception (TLB Refill, UTLB Refill, TLB Modified) is raised becomes a NOP; the bus cycle is not executed, and the TLF bit is set.

  When a bus error exception is requested for a load or store instruction, BsF is set. The load/store result in this case is undefined.

  A Non-maskable interrupt request is held internally, and is raised upon return from the debug exception handler.

  Single Step debug exception is disabled.

  Debug interrupts are ignored and not raised.

  (Note: The result of exceptions or interrupts other than those noted above is undefined. Resets are allowed to occur.)

- Cache lock function

This function is disabled regardless of the Cache register value.

ii) Debug exception handler execution

When a debug exception occurs, the user program should determine the nature of the exception from the Debug register bits (DSS, DBp) and invoke the corresponding exception handler.

iii) Return from a debug exception handler

- When a user program exception occurs at the same time as a Debug exception, change the DEPC value so that a return will be made to the exception handler.
  When NIS = 1, change DEPC to 0xBFC0 0000.

  When OES = 1, change DEPC to 0x8000 0080 (if BEV = 0) or 0xBFC0 0180 (if BEV = 0).
- Executing a DERET instruction
  PC: Contains the DEPC value.

  Debug register DM: Cleared to 0.

  Status register KUc, IEc: Set to 1, enabling interrupts.

  The forced disabling of the cache auto-lock function is cleared and becomes governed by the Cache register value.

  Forced prohibition of Single Step exception is cleared, causing these to be governed by the Debug register SSt.

  NmI and debug exception masks are cleared.

(3) Exception priorities

DSS has a higher priority than DBp, since it occurs in the pipeline E stage. For this reason DSS is not raised at the same time as DBp.

It is further possible for debug exceptions and user exceptions to occur simultaneously. In this case processing branches first to the debug exception handler, but the Status, Cause, EPC and BadVAddr registers are updated to the values for the user exception. DEPC is not automatically updated to the user exception vector address, so the return address must be set by user software.

It is possible for DSS to occur at the same time as an instruction fetch Address Error AdEL or instruction fetch TLB Refill exception (TLBL). DSS cannot occur simultaneously with any other exceptions except these two.

The instruction that triggers the instruction fetch Address Error AdEL or instruction fetch TLB Refill exception (TLBL) will not itself be executed, so it is not possible for DBp to occur at the same time as these two exceptions.

8.3 Details of Debug Exceptions

(1) Single Step exception

- Cause
  - When the Debug register SSt bit is set, a Single Step exception is raised each time one instruction is executed.
- Exception masking
  - The Single Step exception can be masked by the Debug register SSt bit. When SSt is cleared to 0, a Single Step exception cannot be raised.

    (Note: In the debug exception handler, a Single Step exception is masked regardless of the SSt bit value.)
- Processing
  - When this exception is raised, processing jumps to a special debug exception handler at 0xBFC0 0200. (In the TX39 Processor Core, the debug exception vector is located in an uncacheable address space.)
  - The DSS bit in the Debug register is set to 1.
  - A Single Step exception is not raised for an instruction in the branch delay slot.
  - The DEPC register points to the instruction for which a Single Step exception was raised (the instruction about to be executed).
  - When DERET is issued, a Single Step exception is not raised for an instruction at the return destination. If the return destination instruction is a branch instruction, a Single Step exception is not raised for that branch instruction or for the instruction in the branch delay slot.

(2) Debug Breakpoint exception

- Cause
  - A Debug Breakpoint exception is raised when an SDBBP instruction is executed.
- Exception masking
  - The Breakpoint exception cannot be masked.

    (Note: Its behavior during another debug exception is undefined.)
- Instruction causing this exception

  SDBBP
- Processing
  - When this exception is raised, processing jumps to a special debug exception handler at 0xBFC0 0200. (In the TX39 Processor Core, the debug exception vector is located in an uncacheable address space.)
  - The DBp bit in the Debug register is set to 1.
  - The DEPC register points to the SDBBP instruction, unless that instruction is in the branch delay slot, in which case the DEPC register points to the branch instruction and the Debug register DBD bit is set to 1.
- Servicing

  The unused bits of the SDBBP instruction (bits 26 to 6) can be used for passing additional information to the exception handler. In order to allow these bits to be looked at, the user program should load the contents of the memory word containing this instruction, using the DEPC register. When Cause register BD bit is set to 1 (the SDBBP instruction is in the branch delay slot), you should add +4 to the value in EPC register.

Appendix A Instruction Set Details

This appendix presents each instruction in alphabetical order, explaining its operation in detail.

Exceptions that might occur during the execution of each instruction are listed at the end of each explanation. The direct causes of exceptions and how they are handled are explained elsewhere in this manual, and are not described in detail in this Appendix.

The figure at the end of this appendix (Figure A-2) gives the bit codes for the constant fields of each instruction. Encoding of bits for some instructions is also indicated in the individual instruction descriptions.

Instruction Classes

The TX39 Processor Core has five classes of CPU instructions, as follows.

- Load/store

  These instructions transfer data between memory and general-purpose registers. "Base register + 16-bit signed immediate offset" is the only supported addressing mode, so the format of all instructions in this class is I-type.

- Computational

  These instructions perform arithmetic logical and shift operations on register values. The format can be R-type (when both operands and the result are register values) or I-type (when one operand is 16-bit immediate data).

- Jump/branch

  These instructions change the program flow. A jump is always made to a paged absolute address, constructed by combining a 26-bit target address with the upper 4 bits of the program counter (J-type format) or to a 32-bit register address (R-type format). In a branch instruction, the target address is the program counter value plus a 16-bit offset. With a Jump And Link instruction, the return address is saved in general register r31.

- Coprocessor

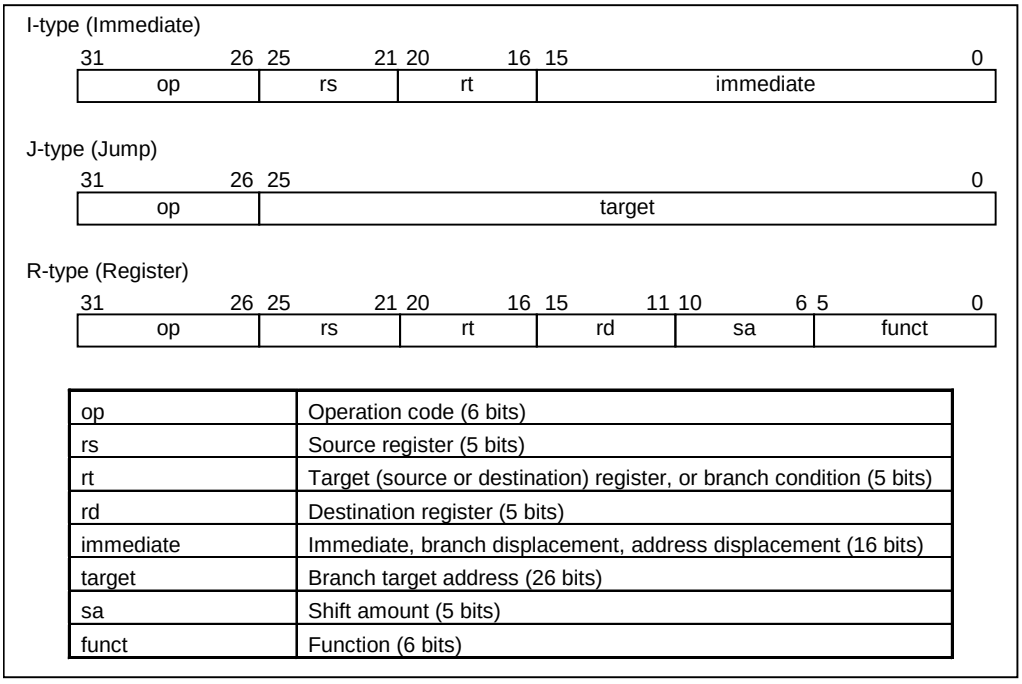
  These instructions execute coprocessor operations. Coprocessor load and store instructions have the I-type format. The format of coprocessor computational instructions differs from one coprocessor to another.

- Special

  These instructions support system calls and breakpoint functions. The format is always R-type.

Instruction Formats

Every instruction consists of a single word (32 bits) aligned on a word boundary. The main instruction formats are shown in Figure A-1.



op	Operation code (6 bits)
rs	Source register (5 bits)
rt	Target (source or destination) register, or branch condition (5 bits)
rd	Destination register (5 bits)
immediate	Immediate, branch displacement, address displacement (16 bits)
target	Branch target address (26 bits)
sa	Shift amount (5 bits)
funct	Function (6 bits)

Figure A-1. CPU Instruction Formats

Instruction Notation Conventions

In this appendix all variable subfields in an instruction format are written in lower-case letters (rs, rt, immediate, etc.).

For some instructions, an alias is used for subfield names, for the sake of clarity. For example, rs in a load/store instruction may be referred to as "base". Such an alias refers to a subfield that can take a variable value and is therefore also written in lower-case letters.

The figure at the end of this appendix (Figure A-2) gives the actual bit codes for all mnemonics. Bit encoding is also indicated in the descriptions of the individual instructions.

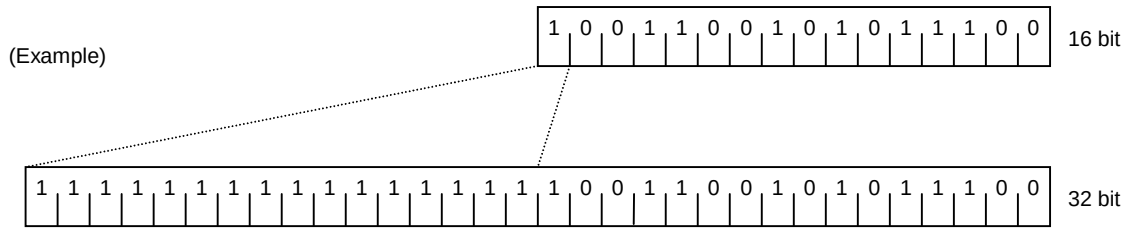
In the explanations that follow, the operation of each instruction is expressed in meta-language. The special symbols used in this instructional notation are shown in Table A-1.

Sign Extension and Zero Extension

With some instructions the bit length may be extended; for example, a 16-bit offset may be extended to 32 bits. This extension can take the form of either a sign extension or zero extension.

- Sign extension

  The extended part is filled with the value of the most significant bit.



- Zero extension

  The extended part is filled with zeros.

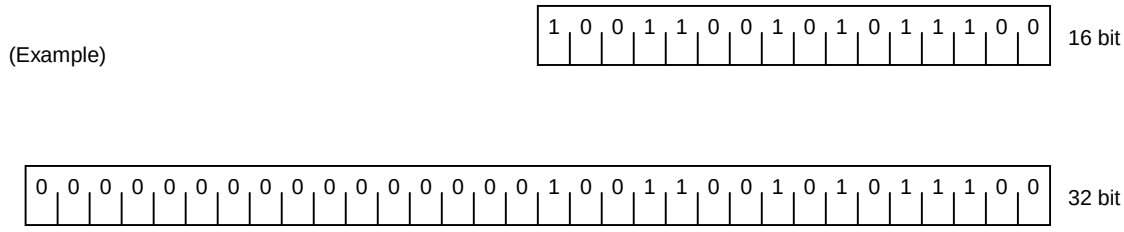


Table A-1. Symbols used in instruction operation notation

Symbol	Meaning
←	Assignment
||	Bit string concatenation
$_x$y	Replication of bit value x into a y-bit string. Note that x is always a single-bit value.
^{x}y~z	Selection of bits y through z of bit string x. Little endian bit notation is always used here. If y is less than z, this expression results in an empty (null length) bit string.
+	Two's complement addition
-	Two's complement subtraction
*	Two's complement multiplication
div	Two's complement division
mod	Two's complement modulo
<	Two's complement "less than" comparison
and	Bitwise logical AND operation
or	Bitwise logical OR operation
xor	Bitwise logical XOR operation
nor	Bitwise logical NOR operation
GPR [x]	General-purpose register x. The content of GPR[0] is always 0, and attempting to change this content has no effect.
CPR [z,x]	General-purpose register x of coprocessor unit z
CCR [z,x]	Control register x of coprocessor unit z
COC [z]	Condition signal of coprocessor unit z
BigEndianMem	Big endian mode as configured at reset (0: little; 1: big). This determines the which endian format is used with the memory interface (see Load Memory and Store Memory) and with kernel mode execution.
Reverse Endian	A signal to reverse the endian format of load and store instructions. This function can be used only in user mode. The endian format is reversed by setting the Status register RE bit. Accordingly, ReverseEndian can be computed as (RE bit AND user mode).
BigEndianCPU	The endian format for load and store instructions (0: little; 1: big). In user mode, the endian format is reversed by setting the RE bit. Accordingly, BigEndianCPU can be computed as BigEndianMem XOR ReverseEndian.
T + i:	This indicates the time steps between operations. Statements within a time step are defined to execute in sequential order, as modified by condition and rule structures. An operation marked by T + i: is executed at instruction cycle i relative to the start of the instruction's execution. For example, an instruction starting at time j executes operations marked T + i: at time i + j. The order is not defined for two instructions or two operations executing at the same time.
vAddress	Virtual address
pAddress	Physical address

Examples of Instruction Notation

Two examples of the notation used in explaining instructions are given below.

Example 1:
GPR[rt] ← immediate || 0^{16}
This means that 16 zero bits are concatenated with an immediate value (normally 16 bits), and the resulting 32-bit string (with the lower 16 bits cleared to 0) is assigned to general-purpose register (GPR) rt.
Example 2:
$(immediate_{15})^{16}$ || immediate $_{15\sim0}$
Bit 15 (the sign bit) of an immediate value is extended to form a 16-bit string, which is linked to bits 15 to 0 of the immediate value, resulting in a 32-bit sign-extended value.

Load and Store Instructions

With the TX39 Processor Core, the instruction immediately following a load instruction can use the loaded value. Hardware is interlocked for this purpose, causing a delay of one instruction cycle. Programming should be carried out with an awareness of the potential effects of the load delay slot.

The descriptions of load/store operations make use of the functions listed in Table A-2 in describing the handling of virtual addresses and physical memory.

Table A-2. Common Load/Store Functions

Function	Meaning
AddressTranslation	A memory management unit (MMU) is used to find the physical address based on a given virtual address.
LoadMemory	The cache and main memory are used to find the contents of the word containing the designated physical address. The low-order two bits of the address and the access type field indicate which of the four bytes in the data word are to be returned. If the cache is enabled for this access, the whole word is returned and loaded into the cache.
StoreMemory	The cache, write buffer and main memory are used to store the word or partial word designated as data in the word containing the designated physical address. The low-order two bits of the address and the access type field indicate which of the four bytes in the data word are to be stored.

The access type field indicates the size of data to be loaded or stored, as given in Table A-3. An address always designates the byte with the smallest byte address in the addressed field, regardless of the access type or the order in which bytes are numbered (endian). This is the left-most byte if big endian is used and the right-most byte if little endian is used.

Table A-3. Load/Store access type designations

Mnemonic	Value	Meaning
WORD	3	Word access (32 bits)
TRIPLEBYTE	2	Triplebyte access (24 bits)
HALFWORD	1	Halfword access (16 bits)
BYTE	0	Byte access (8 bits)

The individual bytes in an addressed word can be determined directly from the access type and the low-order two bits of the address, as shown in Table A-4.

Access Type	Low order address bits		Accessed Bytes							
			Big Endian				Little Endian			
	1	0	31 — 0				31 — 0			
word	0	0	0	1	2	3	3	2	1	0
triple-byte	0	0	0	1	2			2	1	0
	0	1		1	2	3	3	2	1	
halfword	0	0	0	1					1	0
	1	0			2	3	3	2		
byte	0	0	0							0
	0	1		1					1	
	1	0			2			2		
	1	1				3	3			

Table A-4. Load/Store byte access

Jump and Branch Instructions

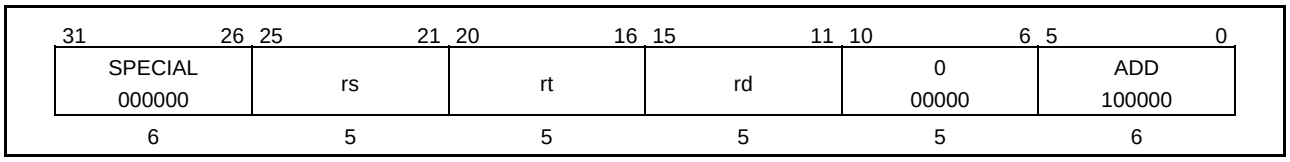
All jump and branch instructions are executed with a delay of one instruction cycle. This means that the immediately following instruction (the instruction in the delay slot) is executed while the branch target instruction is being fetched. A jump or branch instruction should never be put in the delay slot; if this is done, it will not be detected as an error and the result will be undefined.

If an exception or interrupt prevents the delay slot instruction from being completed, the EPC register is set by hardware to point to the preceding jump or branch instruction. Upon returning from the exception or interrupt, both the jump/branch instruction and the instruction in the delay slot are executed.

Jump and branch instructions are sometimes restarted after exceptions or interrupts, so they must be made restartable. When a jump or branch instruction stores a return address value, general-purpose register r31 must not be used as the source register.

Since instructions must be aligned on a word border, the lower two bits of the register value used as an address with a Jump Register instruction or a Jump And Link Register must be 00. If not, an Address Error exception will be raised when the target instruction is fetched.

ADD Add ADD



Format: ADD rd, rs, rt

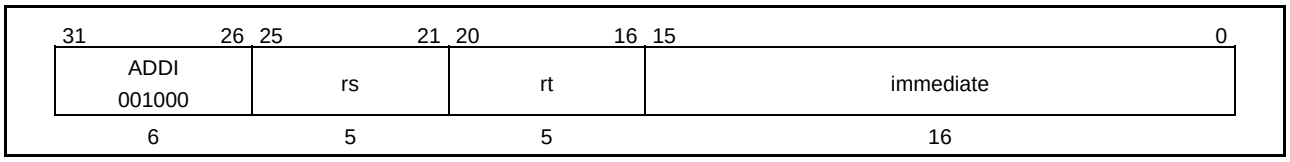
Description: Adds the contents of general-purpose registers rs and rt and puts the result in general-purpose register rd. If carry-out bits 31 and 30 differ, a two's complement overflow exception is raised and destination register rd is not modified.

Operation :

```
T: GPR[rd] ← GPR[rs] + GPR[rt]
```

Exceptions: Overflow

ADDI — Add Immediate — ADDI



Format: ADDI rt, rs, immediate

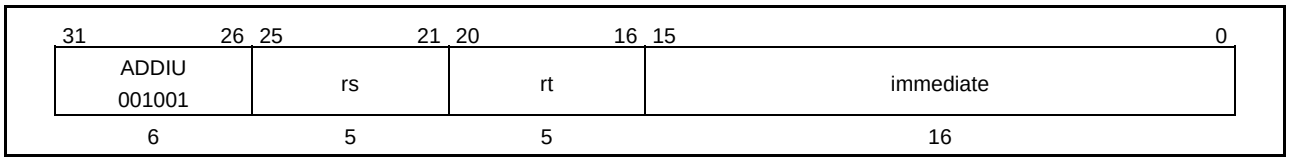
Description: Sign-extends a 16-bit immediate value, adds it to the contents of general-purpose register rs and puts the result in general-purpose register rt. If carry-out bits 31 and 30 differ, a two's complement overflow exception is raised and destination register rt is not modified.

Operation :

```
T: GPR[rt] ← GPR[rs] + (immediate15)16 || immediate15~0
```

Exceptions: Overflow

ADDIU Add Immediate Unsigned ADDIU



Format: ADDIU rt, rs, immediate

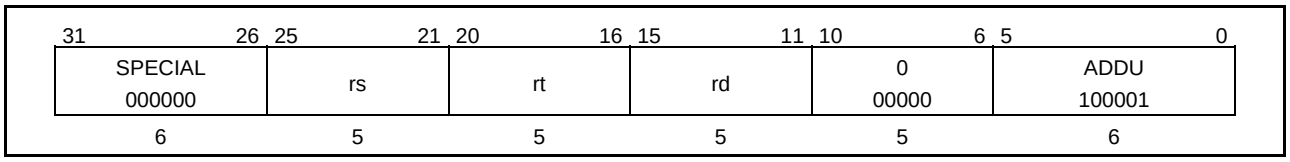
Description: Sign extends a 16-bit immediate value, adds it to the contents of general-purpose register rs and puts the result in general-purpose register rt. The only difference from ADDI is that ADDIU cannot cause an overflow exception.

Operation :

```
T: GPR[rt] ← GPR[rs] + (immediate15 )16 || immediate15~0
```

Exceptions: None

ADDU — Add Unsigned — ADDU



Format: ADDU rd, rs, rt

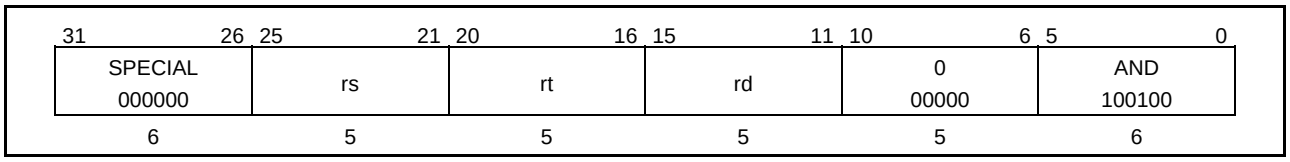
Description: Adds the contents of general-purpose registers rs and rt and puts the result in general-purpose register rd. The only difference from ADD is that ADDU cannot cause an overflow exception.

Operation :

```
T: GPR[rd] ← GPR[rs] + GPR[rt]
```

Exceptions: None

AND And AND



Format: AND rd, rs, rt

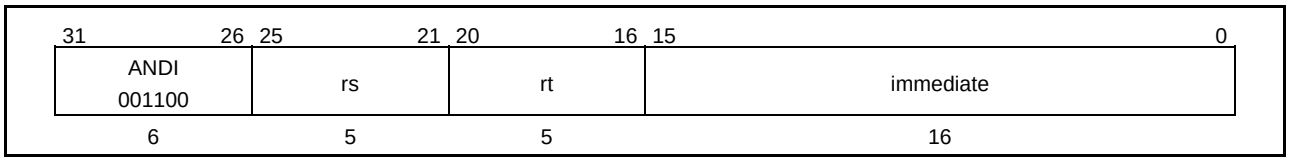
Description: Bitwise ANDs the contents of general-purpose registers rs and rt and puts the result in general-purpose register rd.

Operation :

```
T: GPR[rd] ← GPR[rs) and GPR[rt]
```

Exceptions: None

ANDI Add Immediate ANDI



Format: ANDI rt, rs, immediate

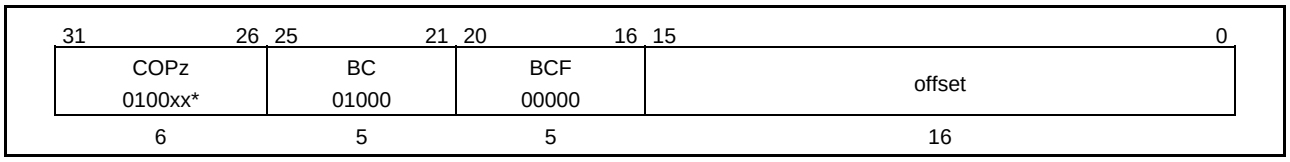
Description: Zero-extends a 16-bit immediate value, bitwise logical ANDs it with the contents of general-purpose register rs and puts the result in general-purpose register rt.

Operation :

```
T: GPR[rt] ← 0^16 || (immediate and GPR[rs]15~0)
```

Exceptions: None

BCzF Branch On Coprocessor z False BCzF



Format: BCzF offset

Description: Generates a branch target address by adding the address of the instruction in the delay slot to the 16-bit offset (that has been left-shifted two bits and sign-extended to 32 bits). If the coprocessor z condition (CPCOND) sampled during execution of the immediately preceding instruction is false, the program branches to the target address after a one-cycle delay.

Operation:

```
T – 1: condition ← not COC[z]
T: target ← (offset15)^14 || offset || 0^2
T + 1: if condition then
            PC ← PC + target
       endif
```

*Refer also to the table on the following page (Operation Code Bit Encoding) or to the section entitled “Bit Encoding of CPU Instruction Opcodes” at the end of this appendix.

BCzF Branch On Coprocessor z False (cont.) BCzF

Exceptions: Coprocessor Unusable exception

Operation Code Bit Encoding:

BCzF

Bit No.	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC0F	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	

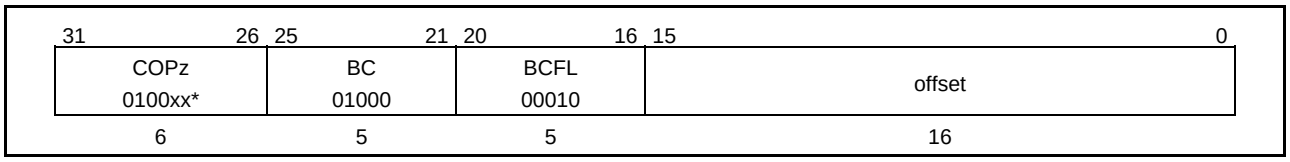
Bit No.	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC1F	0	1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	

Bit No.	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC2F	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	

Bit No.	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC3F	0	1	0	0	1	1	0	1	0	0	0	0	0	0	0	0	

opcode coprocessor unit no. BC sub-opcode branch condition

BCzFL Branch On Coprocessor z False Likely BCzFL



Format: BCzFL offset

Description: Generates a branch target address by adding the address of the instruction in the delay slot to the 16-bit offset (that has been left-shifted two bits and sign-extended to 32 bits). If the coprocessor z condition (CPCOND) sampled during execution of the immediately preceding instruction is false, the program branches to the target address after a one-cycle delay. If the condition is true, the instruction in the delay slot is treated as a NOP.

*Refer also to the table on the following page (Operation Code Bit Encoding) or to the section entitled “Bit Encoding of CPU Instruction Opcodes” at the end of this appendix.

BCzFL Branch On Coprocessor z False Likely (cont.) BCzFL

Operation:

```
T – 1: condition ← not COC[z]
T: target ← (offset15)14 || offset || 02
T + 1: if condition then
            PC ← PC + target
       else
            NullifyCurrentInstruction
       endif
```

Exceptions: Coprocessor Unusable exception

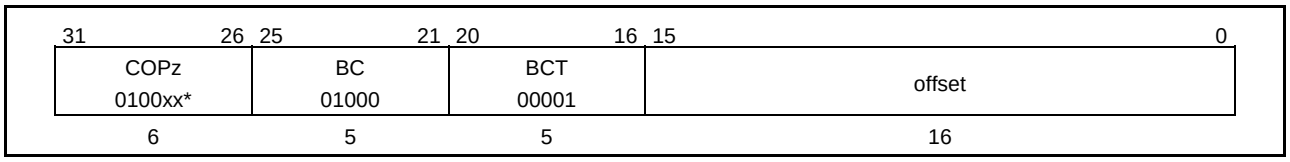
Operation Code Bit Encoding:

BCzF

Bit No.	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC0FL	0	1	0	0	0	0	0	1	0	0	0	0	0	0	1	0	
Bit No.	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC1FL	0	1	0	0	0	1	0	1	0	0	0	0	0	0	1	0	
Bit No.	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC2FL	0	1	0	0	1	0	0	1	0	0	0	0	0	0	1	0	
Bit No.	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC3FL	0	1	0	0	1	1	0	1	0	0	0	0	0	0	1	0	

opcode | coprocessor unit no. | BC sub-opcode | branch condition

BCzT Branch On Coprocessor z True BCzT



Format: BCzT offset

Description: Generates a branch target address by adding the address of the instruction in the delay slot to the 16-bit offset (that has been left-shifted two bits and sign-extended to 32 bits). If the coprocessor z condition (CPCOND) sampled during execution of the immediately preceding instruction is true, the program branches to the target address after a one-cycle delay.

Operation:

```
T – 1: condition ← COC[z]
T: target ← (offset15)^14 || offset || 0^2
T + 1: if condition then
            PC ← PC + target
       endif
```

*Refer also to the table on the following page (Operation Code Bit Encoding) or to the section entitled “Bit Encoding of CPU Instruction Opcodes” at the end of this appendix.

BCzT Branch On Coprocessor z True (cont.) BCzT

Exceptions: Coprocessor Unusable exception

Operation Code Bit Encoding:

BCzT

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
Bit No.																	
BC0T	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	1	

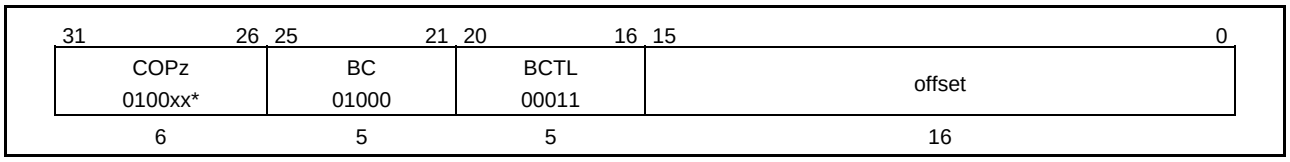
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
Bit No.																	
BC1T	0	1	0	0	0	1	0	1	0	0	0	0	0	0	0	1	

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
Bit No.																	
BC2T	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	1	

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
Bit No.																	
BC3T	0	1	0	0	1	1	0	1	0	0	0	0	0	0	0	1	

opcode coprocessor unit no. BC sub-opcode branch condition

BCzTL Branch On Coprocessor z True Likely BCzTL



Format: BCzTL offset

Description: Generates a branch target address by adding the address of the instruction in the delay slot to the 16-bit offset (that has been left-shifted two bits and sign-extended to 32 bits). If the coprocessor z condition (CPCOND) sampled during execution of the immediately preceding instruction is true, the program branches to the target address after a one-cycle delay. If the condition is false, the instruction in the delay slot is treated as a NOP.

Operation:

```
T – 1: condition ← COC[z]
T: target ← (offset15)^14 || offset || 0^2
T + 1: if condition then
            PC ← PC + target
       else
            NullifyCurrentInstruction
       endif
```

*Refer also to the table on the following page (Operation Code Bit Encoding) or to the section entitled “Bit Encoding of CPU Instruction Opcodes” at the end of this appendix.

BCzTL Branch On Coprocessor z True Likely (cont.) BCzTL

Exceptions: Coprocessor Unusable exception

Operation Code Bit Encoding:

BCzTL

Bit No.	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC0TL	0	1	0	0	0	0	0	1	0	0	0	0	0	0	1	1	

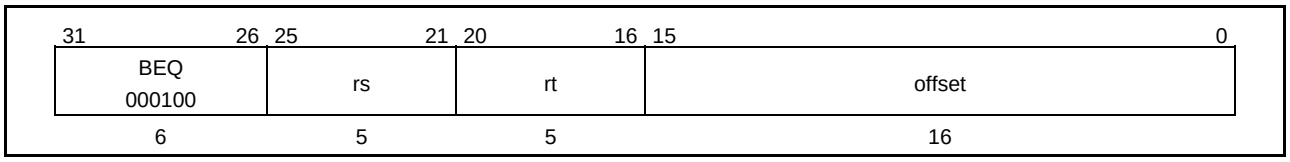
Bit No.	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC1TL	0	1	0	0	0	1	0	1	0	0	0	0	0	0	1	1	

Bit No.	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC2TL	0	1	0	0	1	0	0	1	0	0	0	0	0	0	1	1	

Bit No.	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC3TL	0	1	0	0	1	1	0	1	0	0	0	0	0	0	1	1	

opcode coprocessor unit no. BC sub-opcode branch condition

BEQ Branch On Equal BEQ



Format: BEQ rs, rt, offset

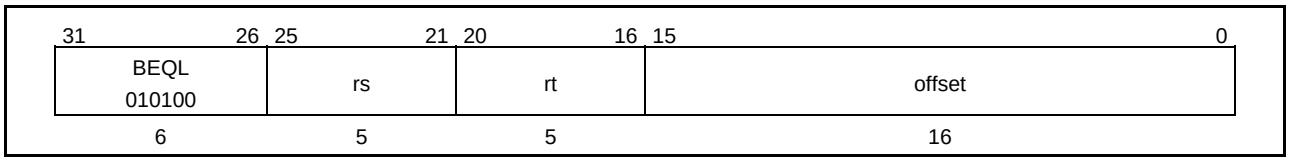
Description: Generates a branch target address by adding the address of the instruction in the delay slot to the 16-bit offset (that has been left-shifted two bits and sign-extended to 32 bits). The contents of general registers rs and rt are compared and, if equal, the program branches to the target address after a one-cycle delay.

Operation:

```
T: target ← (offset15)^14 || offset || 0^2
   condition ← (GPR[rs] = GPR[rt])
T + 1: if condition then
          PC ← PC + target
       endif
```

Exceptions: None

BEQL Branch On Equal Likely BEQL



Format: BEQL rs, rt, offset

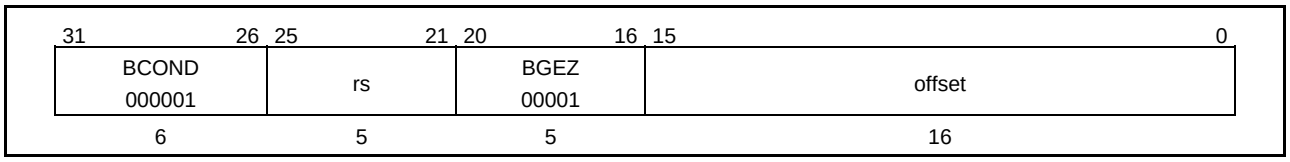
Description: Generates the branch target address by adding the address of the instruction in the delay slot to the 16-bit offset (that has been left-shifted two bits and sign-extended to 32 bits). It compares the contents of general registers rs and rt and, if equal, the program branches to the target address after a one-cycle delay. If the branch is not taken, the instruction in the delay slot is treated as a NOP.

Operation:

```
T: target ← (offset15)^14 || offset || 0^2
   condition ← (GPR[rs] = GPR[rt])
T + 1: if condition then
            PC ← PC + target
       else
            NullifyCurrentInstruction
       endif
```

Exceptions: None

BGEZ Branch On Greater Than Or Equal To Zero BGEZ



Format: BGEZ rs, offset

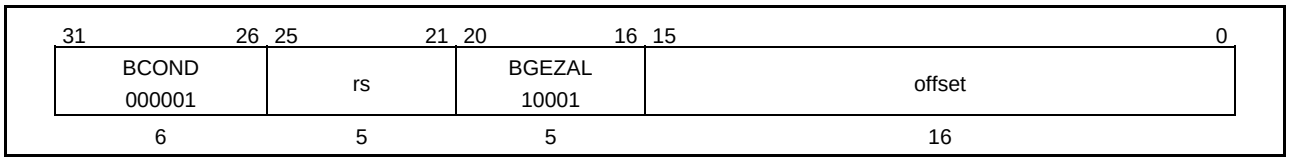
Description: Generates a branch target address by adding the address of the instruction in the delay slot to the 16-bit offset (that has been left-shifted two bits and sign-extended to 32 bits). If the sign bit of the value in general-purpose register rs is 0 (i.e., the value is positive or 0), the program branches to the target address after a one-cycle delay.

Operation:

```
T: target ← (offset15)^14 || offset || 0^2
   condition ← (GPR[rs]31 = 0)
T + 1: if condition then
            PC ← PC + target
       endif
```

Exceptions: None

BGEZAL Branch On Greater Than Or Equal To Zero And Link BGEZAL



Format: BGEZAL rs, offset

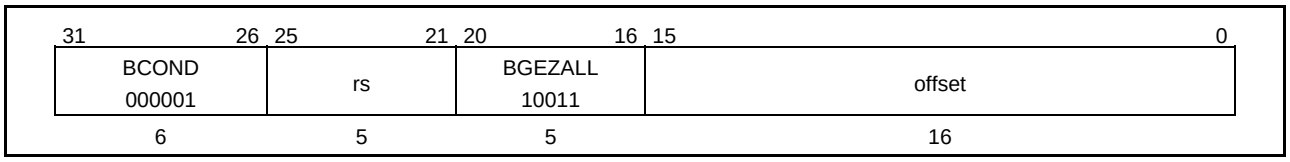
Description: Generates a branch target address by adding the address of the instruction in the delay slot to the 16-bit offset (that has been left-shifted two bits and sign-extended to 32 bits). The address of the instruction following the instruction in the delay slot is unconditionally placed in link register r31 as the return address from the branch. If the sign bit of the value in general-purpose register rs is 0 (i.e., the value is positive or 0), the program branches to the target address after a one-cycle delay. Register r31 should not be used for rs, as this would prevent the instruction from restarting. However, if this is done it is not trapped as an error.

Operation:

```
T: target ← (offset15)^14 || offset || 0^2
   condition ← (GPR[rs]31 = 0)
   GPR[31] ← PC + 8
T + 1: if condition then
            PC ← PC + target
       endif
```

Exceptions: None

BGEZALL Branch On Greater Than Or Equal To Zero And Link Likely BGEZALL



Format: BGEZALL rs, offset

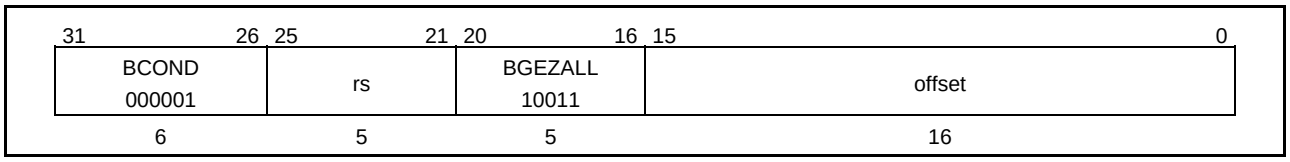
Description: Generates a branch target address by adding the address of the instruction in the delay slot to the 16-bit offset (that has been left-shifted two bits and sign-extended to 32 bits). The address of the instruction following the instruction in the delay slot is unconditionally placed in link register r31 as the return address from the branch. If the sign bit of the value in general-purpose register rs is 0 (i.e., the value is positive or 0), the program branches to the target address after a one-cycle delay. Register r31 should not be used for rs, as this would prevent the instruction from restarting. However, if this is done it is not trapped as an error. If the branch is not taken, the instruction in the delay slot is treated as a NOP.

Operation:

```
T: target ← (offset15)^14 || offset || 0^2
   condition ← (GPR[rs]31 = 0)
   GPR[31] ← PC + 8
T + 1: if condition then
            PC ← PC + target
       else
            NullifyCurrentInstruction
       endif
```

Exceptions: None

BGEZL Branch On Greater Than Or Equal To Zero Likely BGEZL



Format: BGEZL rs, offset

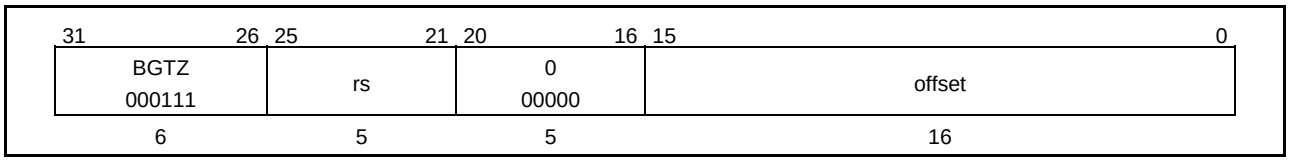
Description: Generates a branch target address by adding the address of the instruction in the delay slot to the 16-bit offset (that has been left-shifted two bits and sign-extended to 32 bits). If the sign bit of the value in general-purpose register rs is 0 (i.e., the value is positive or 0), the program branches to the target address after a one-cycle delay. If the branch is not taken, the instruction in the delay slot is treated as a NOP.

Operation:

```
T: target ← (offset15)^14 || offset || 0^2
   condition ← (GPR[rs]31 = 0)
T + 1: if condition then
            PC ← PC + target
       else
            NullifyCurrentInstruction
       endif
```

Exceptions: None

BGTZ Branch On Greater Than Zero BGTZ



Format: BGTZ rs, offset

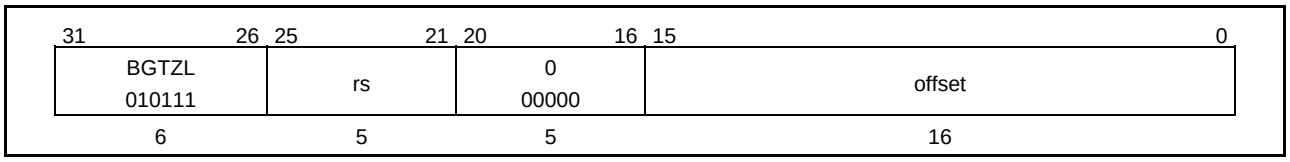
Description: Generates a branch target address by adding the address of the instruction in the delay slot to the 16-bit offset (that has been left-shifted two bits and sign-extended to 32 bits). If the value in general-purpose register rs is positive (i.e., the sign bit of rs is 0 and the rs value is not 0), the program branches to the target address after a one-cycle delay.

Operation:

```
T: target ← (offset15)^14 || offset || 0^2
   condition ← (GPR[rs]31 = 0) and (GPR[rs] ≠ 0^32)
T + 1: if condition then
          PC ← PC + target
       endif
```

Exceptions: None

BGTZL Branch On Greater Than Zero Likely BGTZL



Format: BGTZL rs, offset

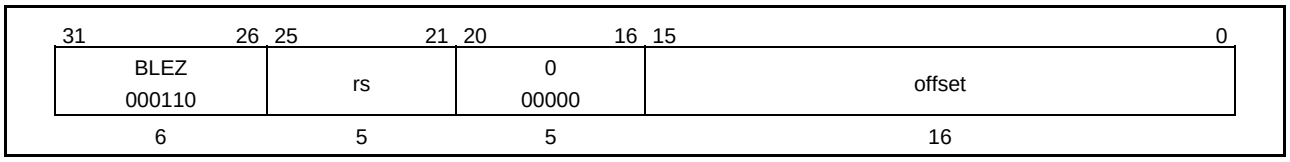
Description: Generates a branch target address by adding the address of the instruction in the delay slot to the 16-bit offset (that has been left-shifted two bits and sign-extended to 32 bits). If the value in general-purpose register rs is positive (i.e., the sign bit of rs is 0 and the rs value is not 0), the program branches to the target address after a one-cycle delay. If the branch is not taken, the instruction in the delay slot is treated as a NOP.

Operation:

```
T: target ← (offset15)^14 || offset || 0^2
   condition ← (GPR[rs]31 = 0) and (GPR[rs] ≠ 0^32)
T + 1: if condition then
            PC ← PC + target
       else
            NullifyCurrentInstruction
       endif
```

Exceptions: None

BLEZ Branch On Less Than Or Equal To Zero BLEZ



Format: BLEZ rs, offset

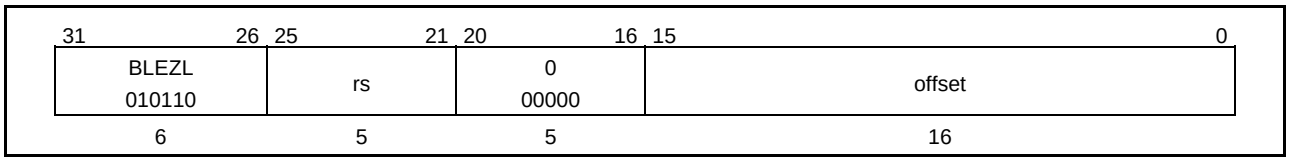
Description: Generates a branch target address by adding the address of the instruction in the delay slot to the 16-bit offset (that has been left-shifted two bits and sign-extended to 32 bits). If the value in general-purpose register rs is negative or 0 (i.e., the sign bit of rs is 1 or the rs value is 0), the program branches to the target address after a one-cycle delay.

Operation:

```
T: target ← (offset15)^14 || offset || 0^2
   condition ← (GPR[rs]31 = 0) or (GPR[rs] = 0^32)
T + 1: if condition then
          PC ← PC + target
       endif
```

Exceptions: None

BLEZL Branch On Less Than Or Equal To Zero Likely BLEZL



Format: BLEZL rs, offset

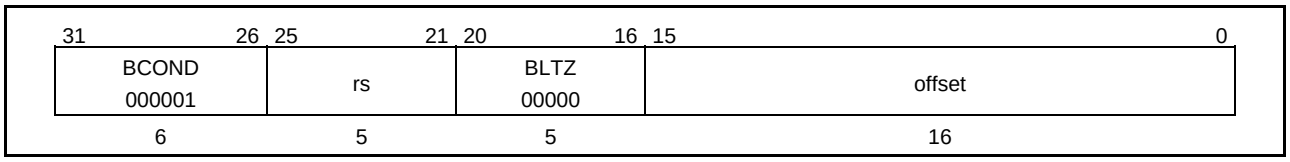
Description: Generates a branch target address by adding the address of the instruction in the delay slot to the 16-bit offset (that has been left-shifted two bits and sign-extended to 32 bits). If the value in general-purpose register rs is negative or 0 (i.e., the sign bit of rs is 1 or the rs value is 0), the program branches to the target address after a one-cycle delay. If the branch is not taken, the instruction in the delay slot is treated as a NOP.

Operation:

```
T: target ← (offset15)^14 || offset || 0^2
   condition ← (GPR[rs]31 = 1) or (GPR[rs] = 0^32)
T + 1: if condition then
            PC ← PC + target
       else
            NullifyCurrentInstruction
       endif
```

Exceptions: None

BLTZ Branch On Less Than Zero BLTZ



Format: BLTZ rs, offset

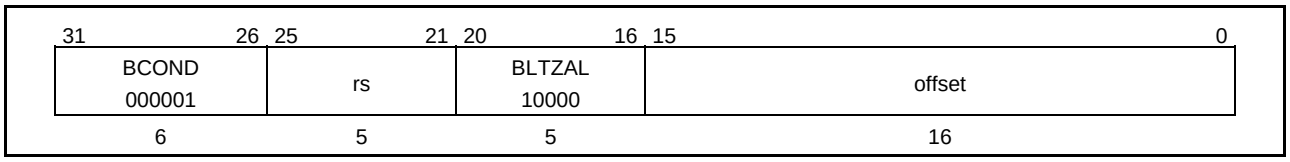
Description: Generates a branch target address by adding the address of the instruction in the delay slot to the 16-bit offset (that has been left-shifted two bits and sign-extended to 32 bits). If the value in general-purpose register rs is negative (i.e., the sign bit of rs is 1), the program branches to the target address after a one-cycle delay.

Operation:

```
T: target ← (offset15)^14 || offset || 0^2
   condition ← (GPR[rs]31 = 1)
T + 1: if condition then
          PC ← PC + target
       endif
```

Exceptions: None

BLTZAL Branch On Less Than Zero And Link BLTZAL



Format: BLTZAL rs, offset

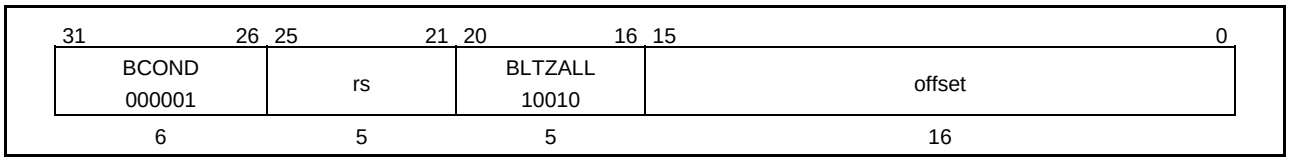
Description: Generates a branch target address by adding the address of the instruction in the delay slot to the 16-bit offset (that has been left-shifted two bits and sign-extended to 32 bits). The address of the instruction following the instruction in the delay slot is unconditionally placed in link register r31 as the return address from the branch. If the value in general-purpose register rs is negative (i.e., the sign bit of rs is 1), the program branches to the target address after a one-cycle delay.
Register r31 should not be used for rs, as this would prevent the instruction from restarting. However, if this is done it is not trapped as an error.

Operation:

```
T: target ← (offset15)^14 || offset || 0^2
   condition ← (GPR[rs]31 = 1)
   GPR[31] ← PC + 8
T + 1: if condition then
            PC ← PC + target
       endif
```

Exceptions: None

BLTZALL Branch On Less Than Zero And Link Likely BLTZALL



Format: BLTZALL rs, offset

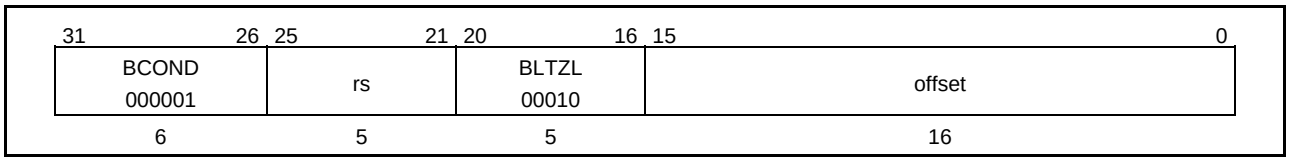
Description: Generates a branch target address by adding the address of the instruction in the delay slot to the 16-bit offset (that has been left-shifted two bits and sign-extended to 32 bits). The address of the instruction following the instruction in the delay slot is unconditionally placed in link register r31 as the return address from the branch. If the value in general-purpose register rs is negative (i.e., the sign bit of rs is 1), the program branches to the target address after a one-cycle delay.
Register r31 should not be used for rs, as this would prevent the instruction from restarting. However, if this is done it is not trapped as an error.
If the branch is not taken, the instruction in the delay slot is treated as a NOP.

Operation:

```
T: target ← (offset15)^14 || offset || 0^2
   condition ← (GPR[rs]31 = 1)
   GPR[31] ← PC + 8
T + 1: if condition then
            PC ← PC + target
       else
            NullifyCurrentInstruction
       endif
```

Exceptions: None

BLTZL Branch On Less Than Zero Likely BLTZL



Format: BLTZL rs, offset

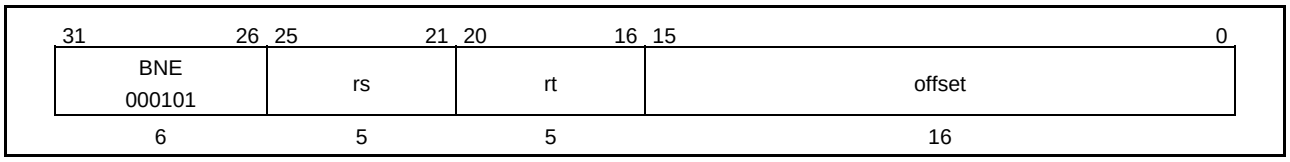
Description: Generates a branch target address by adding the address of the instruction in the delay slot to the 16-bit offset (that has been left-shifted two bits and sign-extended to 32 bits). If the value in general-purpose register rs is negative (i.e., the sign bit of rs is 1), the program branches to the target address after a one-cycle delay. If the branch is not taken, the instruction in the delay slot is treated as a NOP.

Operation:

```
T: target ← (offset15)^14 || offset || 0^2
   condition ← (GPR[rs]31 = 1)
T + 1: if condition then
            PC ← PC + target
       else
            NullifyCurrentInstruction
       endif
```

Exceptions: None

BNE Branch On Not Equal BNE



Format: BNE rs, rt, offset

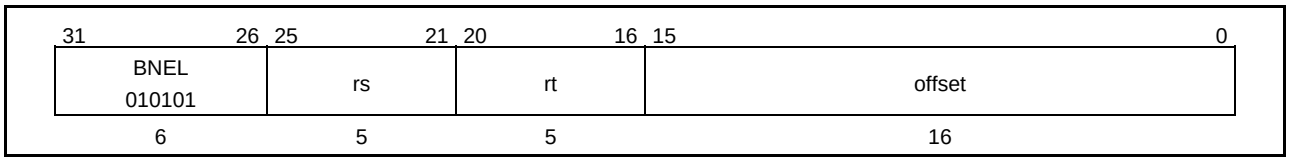
Description: Generates a branch target address by adding the address of the instruction in the delay slot to the 16-bit offset (that has been left-shifted two bits and sign-extended to 32 bits). The contents of general registers rs and rt are compared and, if not equal, the program branches to the target address after a one-cycle delay.

Operation:

```
T: target ← (offset15)^14 || offset || 0^2
   condition ← (GPR[rs] ≠ GPR[rt])
T + 1: if condition then
            PC ← PC + target
       endif
```

Exceptions: None

BNEL Branch On Not Equal Likely BNEL



Format: BNEL rs, rt, offset

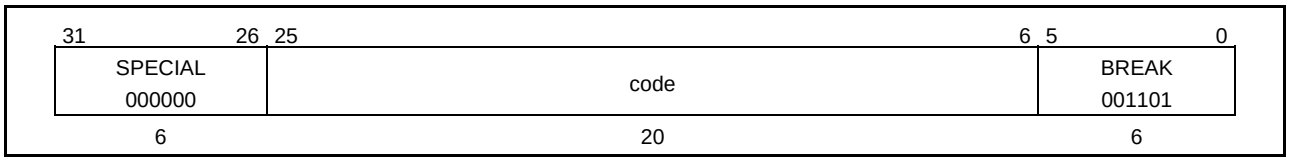
Description: Generates a branch target address by adding the address of the instruction in the delay slot to the 16-bit offset (that has been left-shifted two bits and sign-extended to 32 bits). The contents of general registers rs and rt are compared and, if not equal, the program branches to the target address after a one-cycle delay. If the branch is not taken, the instruction in the delay slot is treated as a NOP.

Operation:

```
T: target ← (offset15)^14 || offset || 0^2
   condition ← (GPR[rs] ≠ GPR[rt])
T + 1: if condition then
            PC ← PC + target
       else
            NullifyCurrentInstruction
       endif
```

Exceptions: None

BREAK Breakpoint BREAK



Format: BREAK code

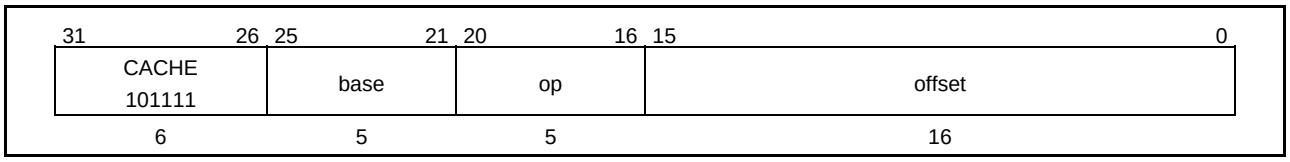
Description: Raises a Breakpoint exception, then immediately passes control to an exception handler. The code field can be used to pass software parameters, but the only way to have the code field retrieved by the exception handler is use the DEPC register to load the contents of the memory word containing this instruction.

Operation:

```
T: BreakpointException
```

Exceptions: Breakpoint exception

CACHE Cache CACHE



Format: CACHE op, offset(base)

Description: Generates a virtual address by sign-extending the 16-bit offset and adding the result to the contents of register base. The virtual address is translated to a physical address, and a 5-bit sub-opcode designates the cache operation to be performed at that address.
If CP0 is unusable (in user mode), the Status register CP0 enable bit is cleared and a Coprocessor Unusable exception is raised. The behavior of this instruction for operation and cache combinations other than those listed in the table below, and when used with an uncached address, is undefined.
Cache index operations (shown for bits 20 through 18 below) designate a cache block using part of the virtual address.
For a directly mapped cache of $2^{CACHESIZE}$ bytes with $2^{BLOCKSIZE}$ bytes per tag, a block is designated as $vAddr_{CACHESIZE-1 \sim BLOCKSIZE}$. In the case of a $2^{WAYSIZE}$-way set-associative cache of $2^{CACHESIZE}$ bytes with $2^{BLOCKSIZE}$ bytes per tag, a set is designated as $vAddr_{CACHESIZE-WAYSIZE-1 \sim BLOCKSIZE}$.
A Cache hit operation (shown for bits 20 through 18 below) accesses the designated cache as an ordinary data reference. If a cache block contains valid data for the generated physical address, it is a hit and the designated operation is performed. In case of a miss, that is, if the cache block is invalid or contains a different address, no operation is performed.

Bits 17~16 of the Cache instruction select the target cache as follows.

Bit#		Cache ID	Cache Name
17	16		
0	0	I	Instruction
0	1	D	Data
1	0	–	(reserved)
1	1	–	(reserved)

CACHE Cache (cont.) CACHE

Bits 20~18 of the Cache instruction select the operation to be performed as follows.

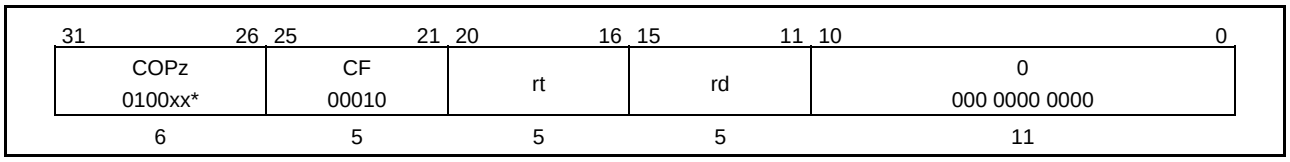
Bit#			Cache ID	Operation Name	Description
20	19	18			
0	0	0	I	IndexInvalidate	Sets the cache state of the cache block to Invalid. This instruction is valid only when the instruction cache is invalid (Config register ICE bit is 0).
0	0	1	D	IndexLRUBitClear	Clears the LRU bit of the cache at the designated index.
0	1	0	D	IndexLockBitClear	Clears the Lock bit of the cache at the designated index.
1	0	0	D	HitInvalidate	If a cache block contains the designated address, sets that cache block to Invalid.

Operation:

```
T: vAddr ← ((offset15)^16 || offset15-0) + GPR[base]
   (pAddr, uncached ← AddressTranslation (vAddr, DATA)
```

Exceptions: Coprocessor Unusable exception

CFCz Move Control From Coprocessor CFCz



Format: CFCz rt, rd

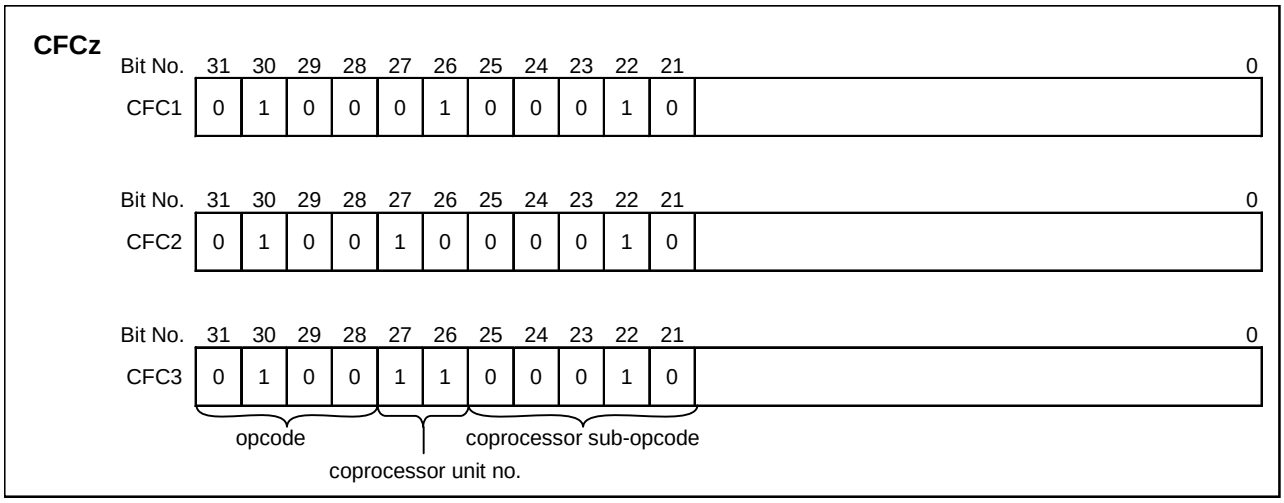
Description: Loads the contents of coprocessor z's control register rd into general-purpose register rt. This instruction is not valid when issued for CP0.

Operation:

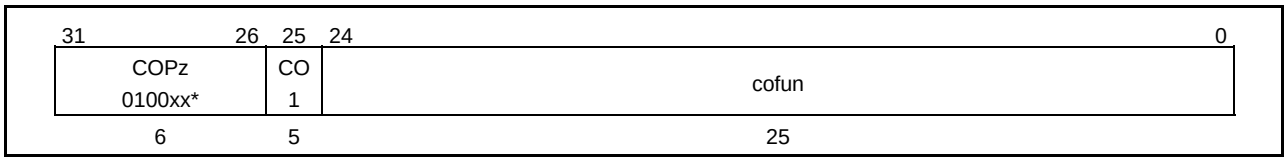
```
T: GPR[rt] ← CCR[z, rd]
```

Exceptions: Coprocessor Unusable exception

* Operation Code Bit Encoding :



COPz Coprocessor Operation COPz



Format: COPz cofun

Description: Performs the operation designated by cofun in coprocessor z. This operation may involve selecting or accessing internal coprocessor registers or changing the status of the coprocessor condition signal (CPCOND), but will not modify internal states of the processor or cache/memory system.

Operation:

```
T: CoprocessorOperation (z, cofun)
```

Exceptions: Coprocessor Unusable exception

* Operation Code Bit Encoding :

COPz

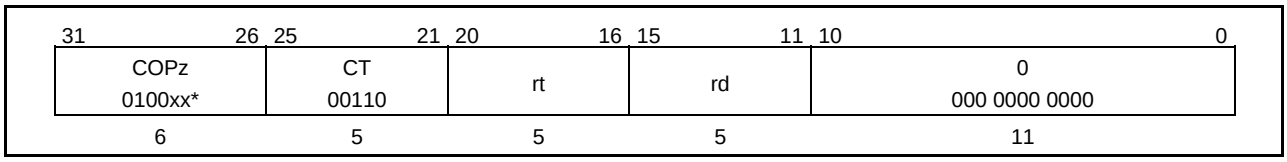
Bit No.	31	30	29	28	27	26	25	0
COP0	0	1	0	0	0	0	1	
COP1	0	1	0	0	0	1	1	
COP2	0	1	0	0	1	0	1	
COP3	0	1	0	0	1	1	1	

opcode

coprocessor unit no.

coprocessor sub-opcode (see to Figure A-2 at end of appendix)

CTCz Move Control To Coprocessor CTCz



Format: CTCz rt, rd

Description: Loads the contents of general register rt into control register rd of coprocessor z. This instruction is not valid when issued for CP0.

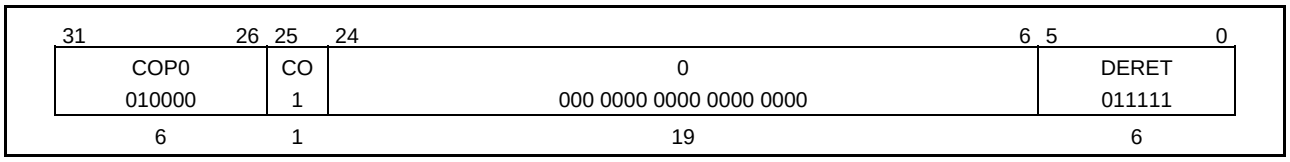
Operation:

```
T: CCR[z, rd] ← GPR[rt]
```

Exceptions: Coprocessor Unusable exception

*Refer to the section entitled “Bit Encoding of CPU Instruction Opcodes” at the end of this appendix.

DERET Debug Exception Return DERET



Format: DERET

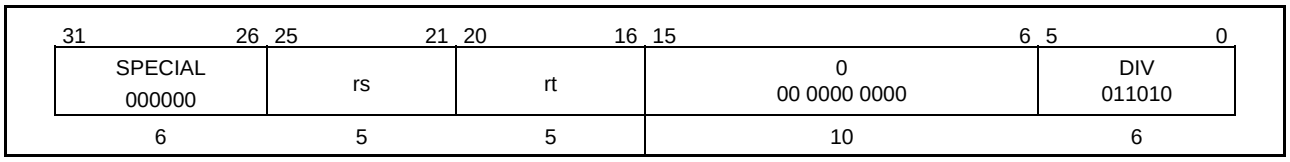
Description: Executes a return from a self-debug interrupt or exception. This instruction requires a branch delay slot like that of the branch or jump instructions, and executes with a delay of one instruction cycle. The DERET instruction itself cannot be put in the delay slot.
The return address stored in the DEPC register is copied to the PC, and processing returns to the original program.
Note: If a MTC0 instruction was used to set the return address in the DEPC register, a minimum of two instructions must be executed before executing DERET.

Operation:

```
T: temp ← DEPC
T + 1: PC ← temp
       Debug30 ← 0
```

Exceptions: Coprocessor Unusable exception

DIV — Divide — DIV



Format: DIV rs, rt

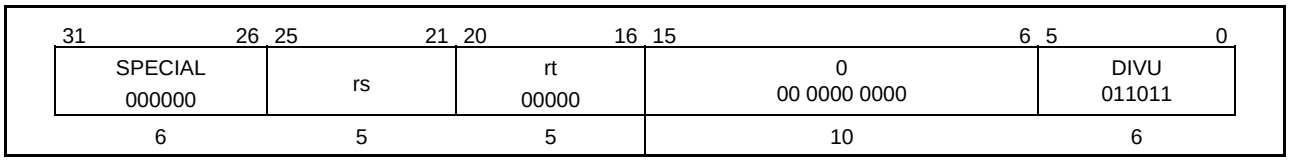
Description: Divides the contents of general register rs by the contents of general register rt, treating both operands as two's complement integers. An overflow exception is never raised. If the divisor is zero, the result is undefined.
Ordinarily, instructions are placed after this instruction to check for zero division and overflow.
The quotient word is loaded into special register LO, and the remainder word into special register HI.
When an attempt is made to read the division result using MFHI, MFLO, MADD or MADDU before the divide operation is completed, the read operation is delayed by an interlock.
Divide operations are executed in an independent ALU and can be carried out in parallel with the execution of other instructions. For this reason, the ALU can continue executing instructions even during a cache miss or other delay cycle in which ordinary instructions cannot be processed.
If either of the two preceding instructions is MFHI, MFLO, MADD or MADDU, the results of those instructions are undefined. For the DIV operation to be carried out correctly, reads of HI or LO must be separated from writes by two or more instructions.

Operation:

```
T – 2: LO ← undefined
       HI ← undefined
T – 1: LO ← undefined
       HI ← undefined
T: LO ← GPR[rs] div GPR[rt]
   HI ← GPR[rs] mod GPR[rt]
```

Exceptions: None

DIVU Divide Unsigned DIVU



Format: DIVU rs, rt

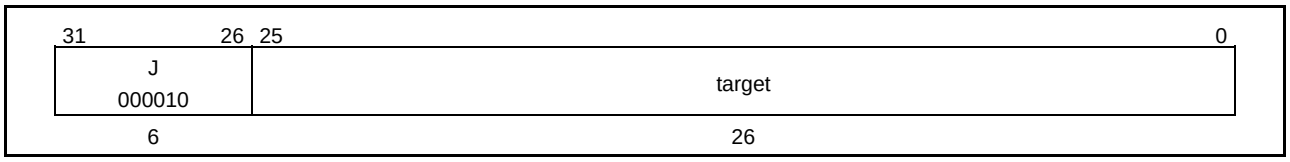
Description: This instruction divides the contents of general register rs by the contents of general register rt, treating both operands as two's complement integers. An integer overflow exception is never raised. If the divisor is zero, the result is undefined.
Ordinarily, an instruction is placed after this instruction to check for zero division.
When an attempt is made to read the division result using MFHI, MFLO, MADD or MADDU before the divide operation is completed, the read operation is delayed by an interlock.
Divide operations are executed in an independent ALU and can be carried out in parallel with the execution of other instructions. For this reason, the ALU can continue executing instructions even during a cache miss or other delay cycle in which ordinary instructions cannot be processed.
Upon completion of the operation, the quotient word is loaded into special register LO, and the remainder word into special register HI.
If either of the two preceding instructions is MFHI, MFLO, MADD or MADDU, the results of those instructions are undefined. For the DIVU operation to be carried out correctly, reads of HI or LO must be separated from writes by two or more instructions.

Operation:

```
T – 2: LO ← undefined
       HI ← undefined
T – 1: LO ← undefined
       HI ← undefined
T: LO ← (0 || GPR[rs]) div (0 || GPR[rt])
   HI ← (0 || GPR[rs]) mod (0 || GPR[rt])
```

Exceptions: None

J Jump J



Format: J target

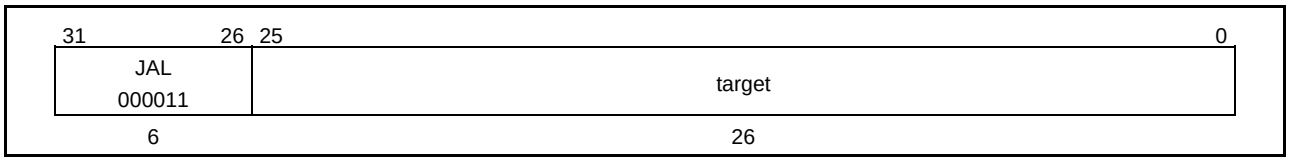
Description: Generates a jump target address by left-shifting the 26-bit target by two bits and combining the result with the high-order 4 bits of the address of the instruction in the delay slot. The program jumps unconditionally to this address after a delay of one instruction cycle.

Operation:

```
T: temp ← target
T + 1: PC ← PC31~28 || temp ||0^2
```

Exceptions: None

JAL — Jump And Link — JAL



Format: JAL target

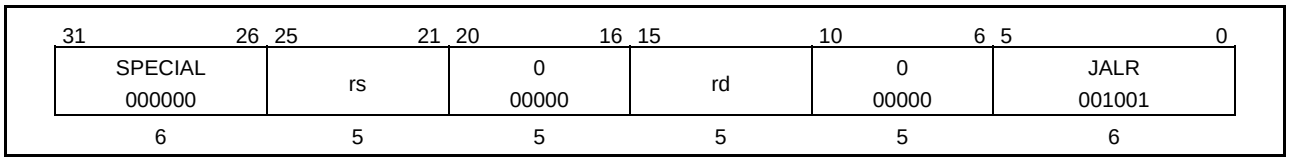
Description: Generates a jump target address by left-shifting the 26-bit target by 2 bits and combining the result with the high-order 4 bits of the address of the instruction in the delay slot. The program jumps unconditionally to this address after a delay of one instruction cycle. The address of the instruction after the delay slot is placed in link register r31 as the return address from the jump.

Operation:

```
T: temp ← target
   GPR[31] ← PC + 8
T + 1: PC ← PC31–28 || temp ||0^2
```

Exceptions: None

JALR Jump And Link Register JALR



Format: JALR rs

JALR rd, rs

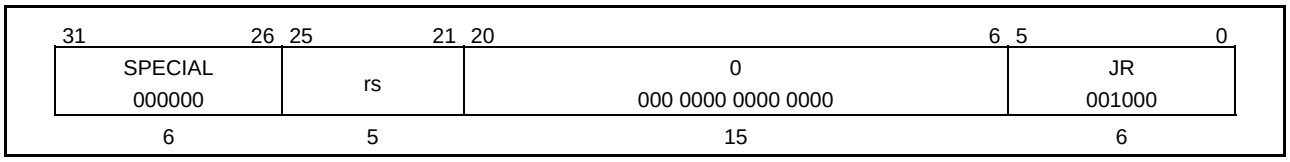
Description: Causes the program to jump unconditionally to the address in general register rs after a delay of one instruction cycle. The address of the instruction following the delay slot is put in general register rd as the return address from the jump. If rd is omitted from the assembly language instruction, r31 is used as the default value.
Register specifiers rs and rd must not be equal, since such an instruction would not have the same result if re-executed. This error is not trapped, however, the result is undefined.
Since instructions must be aligned on a word boundary, the two low-order bits of the value in target register rs must be 00. If not, an Address Error exception will be raised when the target instruction is fetched.

Operation:

```
T: temp ← GPR[rs]
   GPR[rd] ← PC + 8
T + 1: PC ← temp
```

Exceptions: None

JR Jump Register JR



Format: JR rs

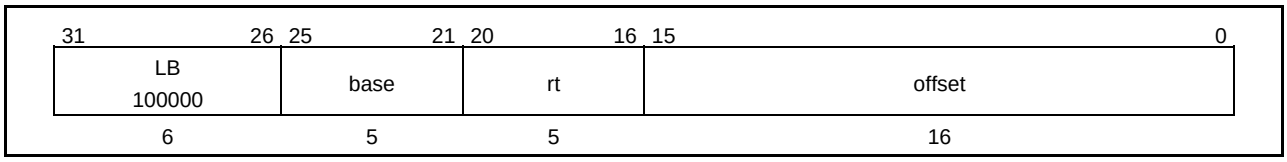
Description: Causes the program to jump unconditionally to the address in general register rs after a delay of one instruction cycle.
Since instructions must be aligned on a word boundary, the two low-order bits of target register rs must be 00. If not, an Address Error exception will be raised when the target instruction is fetched.

Operation:

```
T: temp ← GPR[rs]
T + 1: PC ← temp
```

Exceptions: None

LB Load Byte LB



Format: LB rt, offset(base)

Description: Generates a 32-bit effective address by sign-extending the 16-bit offset and adding it to the contents of general-purpose register base. It then sign-extends the byte at the memory location pointed to by the effective address and loads the result into general-purpose register rt.

Operation:

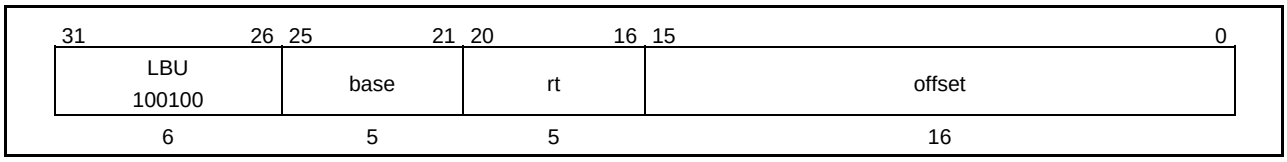
```
T: vAddr ← ((offset15)^16 || offset15~0) + GPR[base]
   (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
   pAddr ← pAddr31~2 || (pAddr1~0 xor ReverseEndian^2)
   mem ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
   byte ← vAddr1~0 xor BigEndianCPU^2
   GPR[rt] ←(mem7+8*byte)^24 || mem7+8byte~8*byte
```

Exceptions: UTLB Refill exception (reserved)

TLB Refill exception (reserved)

Address Error exception

LBU Load Byte Unsigned LBU



Format: LBU rt, offset(base)

Description: Generates a 32-bit effective address by sign-extending the 16-bit offset and adding it to the contents of general-purpose register base. It then zero-extends the byte at the memory location pointed to by the effective address and loads the result into general-purpose register rt.

Operation:

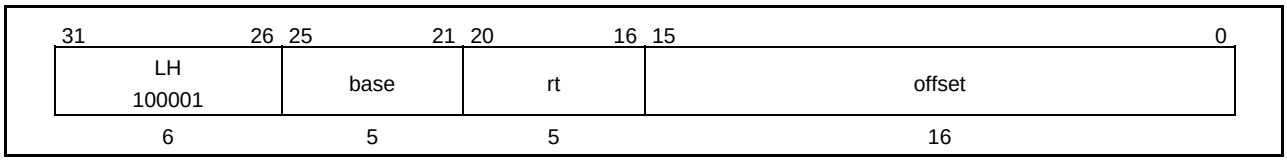
```
T: vAddr ← ((offset15)^16 || offset15~0) + GPR[base]
   (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
   pAddr ← pAddr31~2 || (pAddr1~0 xor ReverseEndian^2)
   mem ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
   byte ← vAddr1~0 xor BigEndianCPU^2
   GPR[rt] ← 0^24 || mem7 + 8*byte ~ 8*byte
```

Exceptions: UTLB Refill exception (reserved)

TLB Refill exception (reserved)

Address Error exception

LH Load Halfword LH



Format: LH rt, offset(base)

Description: Generates a 32-bit effective address by sign-extending the 16-bit offset and adding it to the contents of general-purpose register base. It then sign-extends the halfword at the memory location pointed to by the effective address and loads the result into general-purpose register rt.
If the effective address is not aligned on a halfword boundary, i.e., if the least significant bit of the effective address is not 0, an Address Error exception is raised.

Operation:

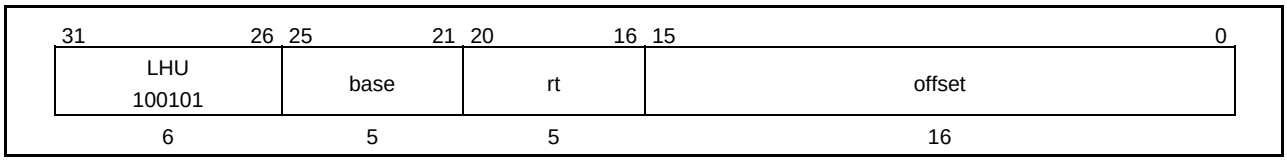
```
T: vAddr ← ((offset15)^16 || offset15~0) + GPR[base]
   (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
   pAddr ← pAddr31~2 || (pAddr1~0 xor (ReverseEndian || 0))
   mem ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
   byte ← vAddr1~0 xor (BigEndianCPU || 0)
   GPR[rt] ← (mem15 + 8*byte)^16 || mem15 + 8*byte ~ 8*byte
```

Exceptions: UTLB Refill exception (reserved)

TLB Refill exception (reserved)

Address Error exception

LHU Load Halfword Unsigned LHU



Format: LHU rt, offset(base)

Description: Generates a 32-bit effective address by sign-extending the 16-bit offset and adding it to the contents of general-purpose register base. It then zero-extends the halfword at the memory location pointed to by the effective address and loads the result into general-purpose register rt.
If the effective address is not aligned on a halfword boundary, i.e., if the least significant bit of the effective address is not 0, an Address Error exception is raised.

Operation:

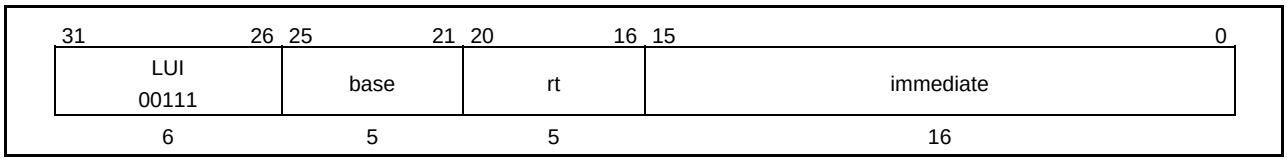
```
T: vAddr ← ((offset15)^16 || offset15~0) + GPR[base]
   (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
   pAddr ← pAddr31~2 || (pAddr1~0 xor (ReverseEndian^2)
   mem ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
   byte ← vAddr1~0 xor (BigEndianCPU || 0)
   GPR[rt] ← 0^16 || mem15 + 8*byte ~ 8*byte
```

Exceptions: UTLB Refill exception (reserved)

TLB Refill exception (reserved)

Address Error exception

LUI Load Upper Immediate LUI



Format: LUI rt, immediate

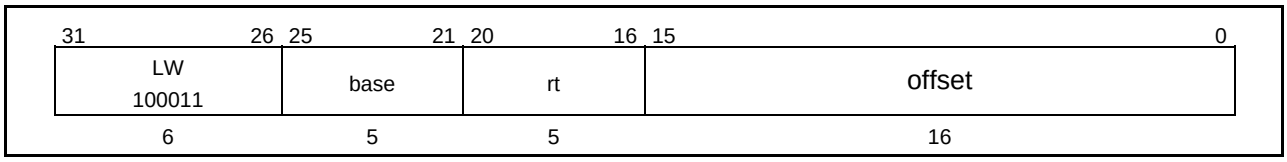
Description: Left-shifts 16-bit immediate by the 16 bits, zero-fills the low-order 16 bits of the word, and puts the result in general register rt.

Operation:

```
T: GPR[rt] ← immediate || 0^16
```

Exceptions: None

LW Load Word LW



Format: LW rt, offset(base)

Description: Generates a 32-bit effective address by sign-extending the 16-bit offset and adding it to the contents of general-purpose register base. It then loads the word at the memory location pointed to by the effective address into general-purpose register rt.
If the effective address is not aligned on a word boundary, i.e., if the low-order 2 bits of the effective address are not 00, an Address Error exception is raised.

Operation:

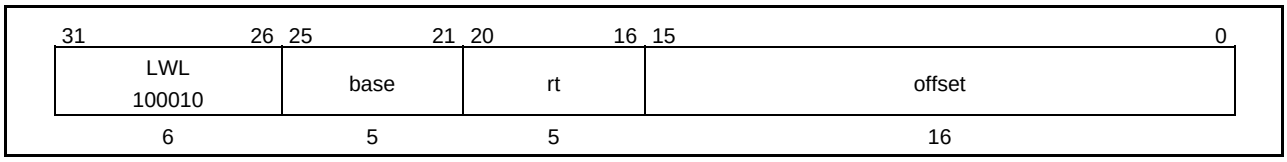
```
T: vAddr ← ((offset15)^16 || offset15~0) + GPR[base]
   (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
   mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
   GPR[rt] ← mem
```

Exceptions: UTLB Refill exception (reserved)

TLB Refill exception (reserved)

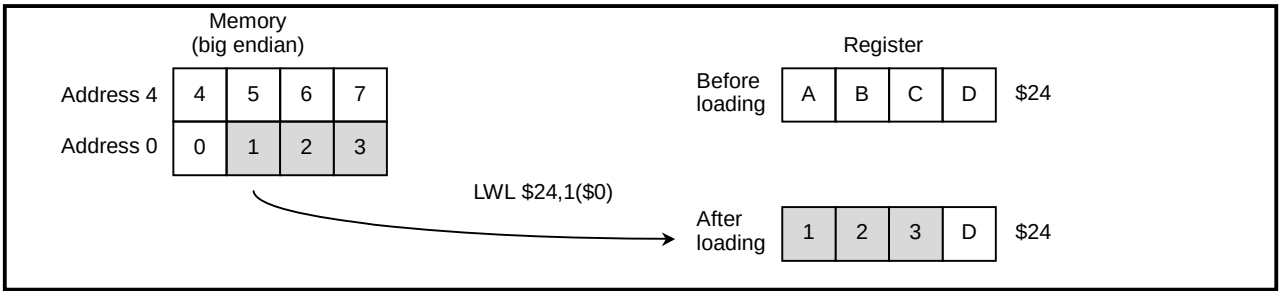
Address Error exception

LWL Load Word Left LWL



Format: LWL rt, offset(base)

Description: Used together with LWR to load four consecutive bytes to a register when the bytes cross a word boundary. LWL loads the left part of the register from the appropriate part of the high-order word; LWR loads the right part of the register from the appropriate part of the low-order word.
This instruction generates a 32-bit effective address that can point to any byte, by sign-extending the 16-bit offset and adding it to the contents of general-purpose register base. Only bytes from the word in memory containing the designated starting byte are read. Depending on the starting byte, from one to four bytes are loaded.
The concept is illustrated below. This instruction (LWL) first loads the designated memory byte into the high-order (left-most) byte of the register; it then continues loading bytes from memory into the register, proceeding toward the low-order byte of the memory word and the low-order byte of the register, until it reaches the low-order byte of the memory word. The least-significant (right-most) byte of the register is not changed.



LWL Load Word Left (cont.) LWL

It is alright to put a load instruction that uses the same rt as the LWL instruction immediately before LWL (or LWR). The contents of general-purpose register rt are bypassed internally in the processor, eliminating the need for a NOP between the two instructions. No Address Error instruction is raised due to misalignment.

Operation:

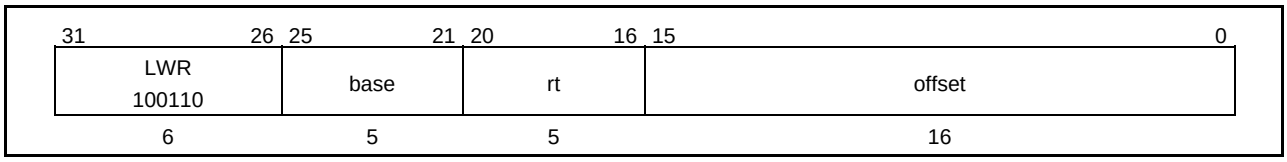
```
T: vAddr ← ((offset15)^16 || offset15~0) + GPR[base]
   (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
   pAddr ← pAddr31~2 || (pAddr1~0 xor ReverseEndian^2)
   if BigEndianMem = 0 then
       pAddr ← pAddrPSIZE-31~2 || 0^2
   endif
   byte ← vAddr1~0 xor BigEndianCPU^2
   mem ← LoadMemory (uncached, byte, pAddr, vAddr, DATA)
   GPR[rt] ← mem7+8*byte~0 || GPR[rt]23-8*byte~0
```

Exceptions: UTLB Refill exception (reserved)

TLB Refill exception (reserved)

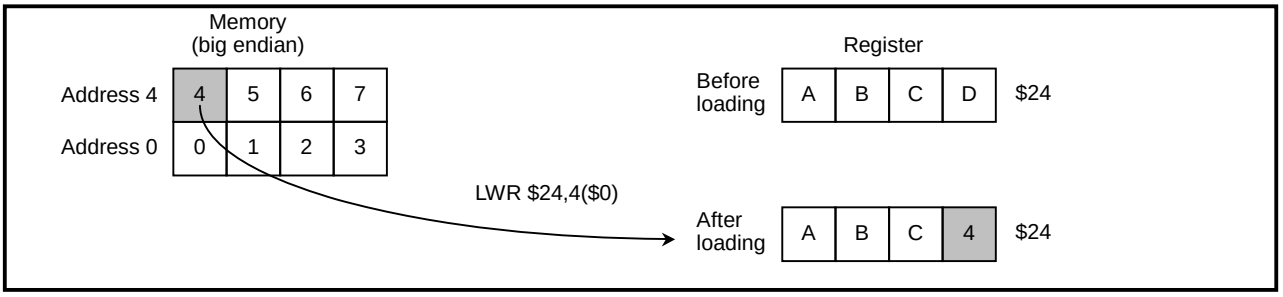
Address Error exception

LWR Load Word Right LWR



Format: LWR rt, offset(base)

Description: Used together with LWL to load four consecutive bytes to a register when the bytes cross a word boundary. LWR loads the right part of the register from the appropriate part of the low-order word; LWL loads the left part of the register from the appropriate part of the high-order word.
This instruction generates a 32-bit effective address that can point to any byte, by sign-extending the 16-bit offset and adding it to the contents of general-purpose register base. Only bytes from the word in memory containing the designated starting byte are read. Depending on the starting byte, from one to four bytes are loaded.
The concept is illustrated below. This instruction (LWR) first loads the designated memory byte into the low-order (right-most) byte of the register; it then continues loading bytes from memory into the register, proceeding toward the high-order byte of the memory word and the high-order byte of the register, until it reaches the high-order byte of the memory word. The most-significant (left-most) byte of the register is not changed.



LWR Load Word Right (cont.) LWR

It is alright to put a load instruction that uses the same rt as the LWR instruction immediately before LWR. The contents of general-purpose register rt are bypassed internally in the processor, eliminating the need for a NOP between the two instructions. No Address Error instruction is raised due to misalignment.

Operation:

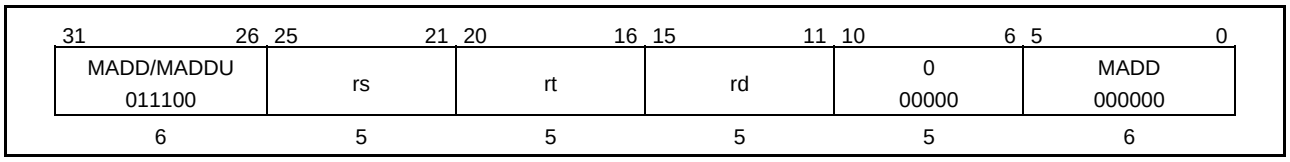
```
T: vAddr ← ((offset15)^16 || offset15~0) + GPR[base]
   (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
   pAddr ← pAddr31~2 || (pAddr1~0 xor ReverseEndian^2)
   if BigEndianMem = 1 then
        pAddr ← pAddrPSIZE-31~2 || 0^2
   endif
   byte ← vAddr1~0 xor BigEndianCPU^2
   mem ← LoadMemory (uncached, WORD-byte, pAddr, vAddr, DATA)
   GPR[rt] ← GPR[rt]31~32-8*byte || mem31-8*byte~0
```

Exceptions: UTLB Refill exception (reserved)

TLB Refill exception (reserved)

Address Error exception

MADD Multiply/Add MADD



Format: MADD rs, rt

MADD rd, rs, rt

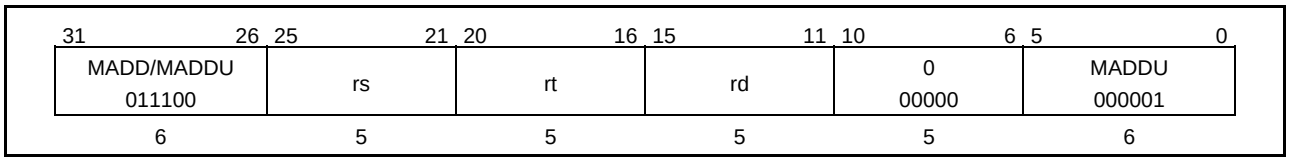
Description: Multiplies the contents of general registers rs and rt, treating both values as two's complement, and puts the double-word result in special registers HI and LO. An overflow exception is never raised.
The low-order word of the multiplication result is put in general register rd and in special register LO, whereas the high-order word of the result is put in special register HI.
If rd is omitted in assembly language, 0 is used as the default value. To guarantee correct operation even if an interrupt occurs, neither of the two instructions following MADD should be DIV or DIVU instructions which modify the HI and LO register contents.

Operation:

```
T: t ← (HI || LO) + GPR[rs]*GPR[rt]
   LO ← t31~0
   HI ← t63~32
   GPR[rd] ← t31~0
```

Exceptions: None

MADDU Multiply/Add Unsigned MADDU



Format: MADD rs, rt

MADDU rd, rs, rt

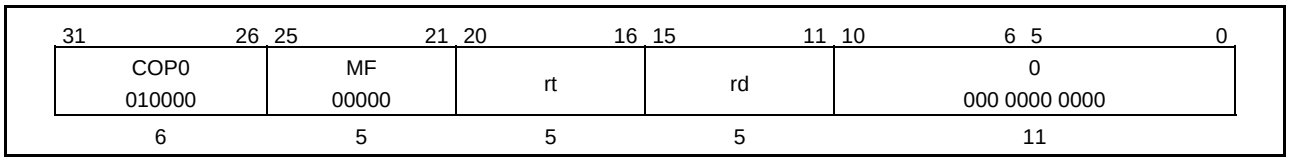
Description: Multiplies the contents of general registers rs and rt, treating both values as unsigned , and puts the double-word result in special registers HI and LO. An overflow exception is never raised.
The low-order word of the multiplication result is put in general register rd and in special register LO, whereas the high-order word of the result is put in special register HI.
If rd is omitted in assembly language, 0 is used as the default value. To guarantee correct operation even if an interrupt occurs, neither of the two instructions following MADDU should be DIV or DIVU instructions which the HI and LO register contents.

Operation:

```
T: t ← (HI || LO) + (0 || GPR[rs])*( 0 || GPR[rt])
   LO ← t31~2
   HI ← t63~32
   GPR[rd] ← t31~0
```

Exceptions: None

MFC0 Move From System Control Coprocessor MFC0



Format: MFC0 rt, rd

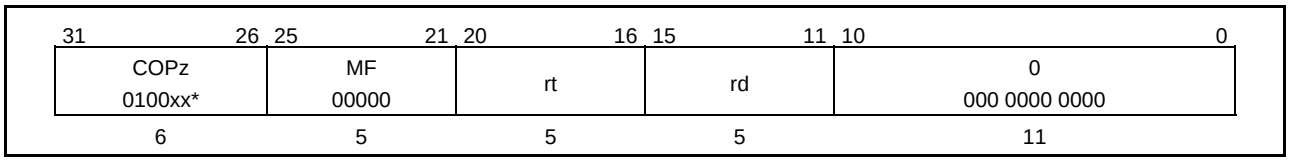
Description: Loads the contents of coprocessor CP0 register rd into general-purpose register rt.

Operation:

```
T: GPR[rt] ← CPR[0, rd]
```

Exceptions: Coprocessor Unusable exception

MFCz — Move From Coprocessor — MFCz



Format: MFCz rt, rd

Description: Loads the contents of coprocessor z register rd into general-purpose register rt.

Operation:

```
T: GPR[rt] ← CPR[z, rd]
```

Exceptions: Coprocessor Unusable exception

*Refer also to the table on the following page (Operation Code Bit Encoding) or to the section entitled “Bit Encoding of CPU Instruction Opcodes” at the end of this appendix.

MFCz Move From Coprocessor (cont.) MFCz

*Operation Code Bit Encoding :

MFCz

Bit No.	31	30	29	28	27	26	25	24	23	22	21	0
MFC0	0	1	0	0	0	0	0	0	0	0	0	

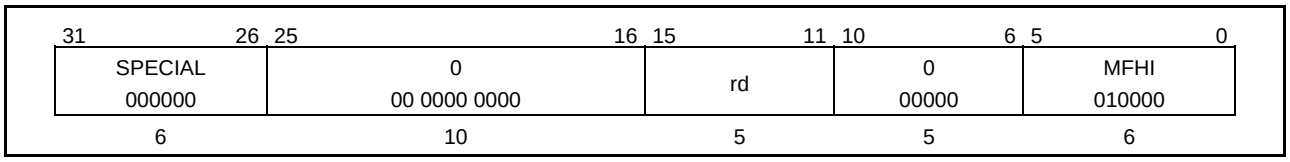
Bit No.	31	30	29	28	27	26	25	24	23	22	21	0
MFC1	0	1	0	0	0	1	0	0	0	0	0	

Bit No.	31	30	29	28	27	26	25	24	23	22	21	0
MFC2	0	1	0	0	1	0	0	0	0	0	0	

Bit No.	31	30	29	28	27	26	25	24	23	22	21	0
MFC3	0	1	0	0	1	1	0	0	0	0	0	

opcode
coprocessor unit no.
coprocessor sub-opcode

MFHI Move From HI MFHI



Format: MFHI rd

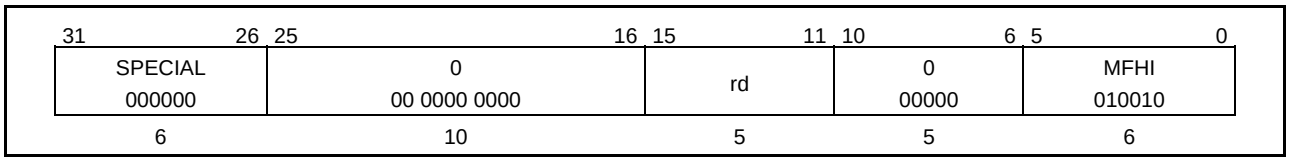
Description: Loads the contents of special register HI into general-purpose register rd.
To guarantee correct operation even if an interrupt occurs, neither of the two instructions following MFHI should be DIV or DIVU instructions which modify the HI register contents.

Operation:

```
T: GPR[rd] ← HI
```

Exceptions: None

MFLO Move From LO MFLO



Format: MFLO rd

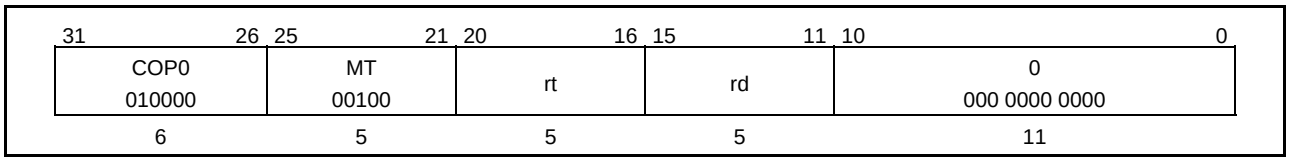
Description: Loads the contents of special register LO into general-purpose register rd.
To guarantee correct operation even if an interrupt occurs, neither of the two instructions following MFLO should be DIV or DIVU instructions which the LO register contents.

Operation:

```
T: GPR[rd] ← LO
```

Exceptions: None

MTC0 Move To System Control Coprocessor MTC0



Format: MTC0 rt, rd

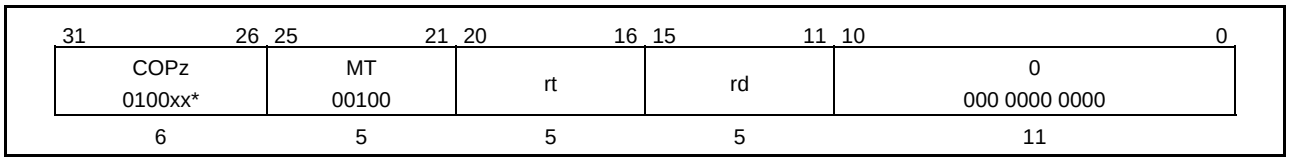
Description: Loads the contents of general-purpose register rt into CP0 coprocessor register rd. Executing this instruction may in some cases modify the state of the virtual address translation system, therefore the behavior of a load instruction, store instruction or TLB operation placed immediately before or after the MTC0 instruction cannot be defined.

Operation:

```
T: CPR[0, rd] ← GPR[rt]
```

Exceptions: Coprocessor Unusable exception

MTCz Move To Coprocessor MTCz



Format: MTCz rt, rd

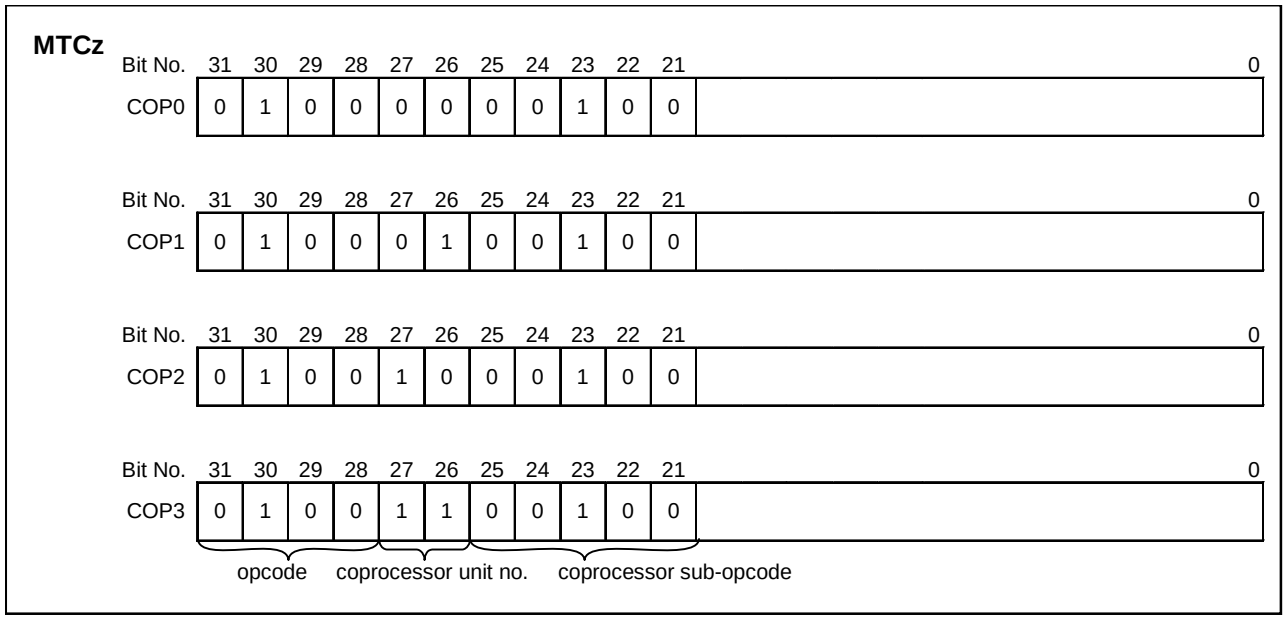
Description: Loads the contents of general-purpose register rt into coprocessor z register rd.

Operation:

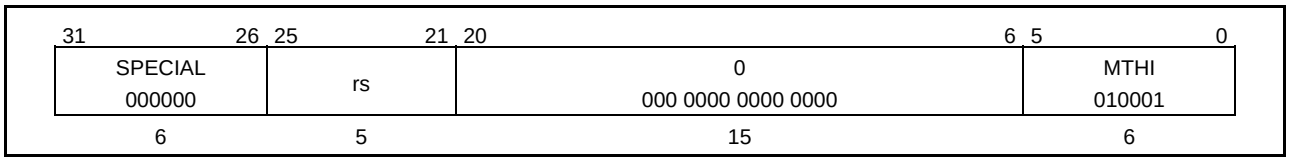
```
T: CPR[z, rd] ← GPR[rt]
```

Exceptions: Coprocessor Unusable exception

*Operation Code Bit Encoding :



MTHI Move To HI MTHI



Format: MTHI rs

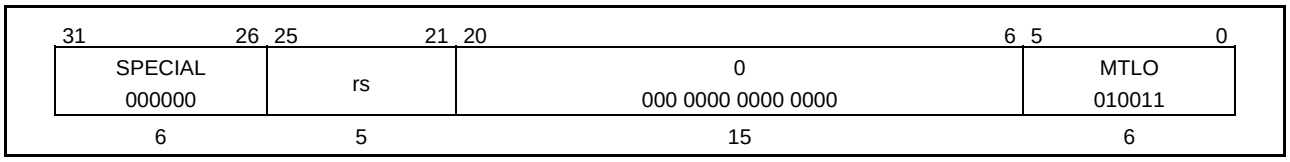
Description: Loads the contents of general-purpose register rs into special register HI.
If executed after a DIV or DIVU instruction or before a MFLO, MFHI, MTLO or MTHI instruction, the contents of special register LO will be undefined.

Operation:

```
T: HI ← GPR[rs]
```

Exceptions: None

MTLO Move To LO MTLO



Format: MTLO rs

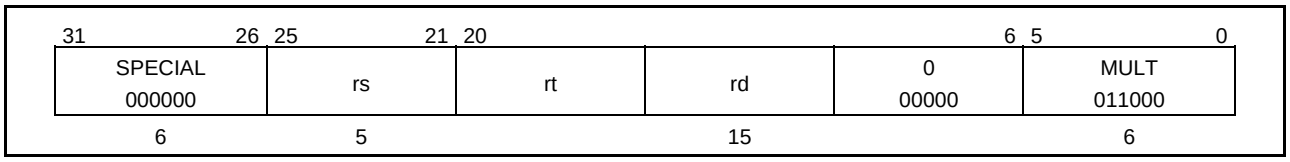
Description: Loads the contents of general-purpose register rs into special register LO.
If executed after a DIV or DIVU instruction or before a MFLO, MFHI, MTLO or MTHI instruction, the contents of special register HI will be undefined.

Operation:

```
T: LO ← GPR[rs]
```

Exceptions: None

MULT Multiply MULT



Format: MULT rs, rt

MULT rd, rs, rt

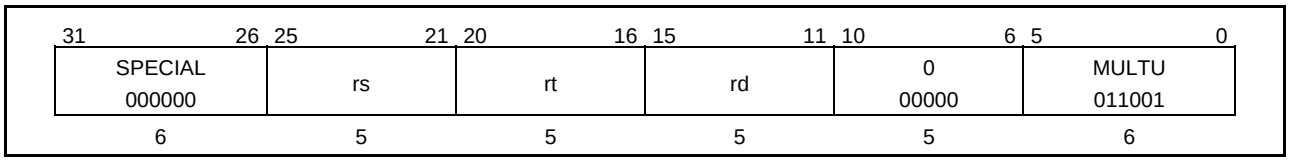
Description: Multiplies the contents of general-purpose register rs by the contents of general register rt, treating both register values as 32-bit two's complement values. This instruction cannot raise an integer overflow exception.
The low-order word of the multiplication result is put in general register rd and in special register LO, whereas the high-order word of the result is put in special register HI.
If rd is omitted in assembly language, 0 is used as the default value.

Operation:

```
T: t ← GPR[rs]*GPR[rt]
   LO ← t31~0
   HI ← t63~32
   GPR[rd] ← t31~0
```

Exceptions: None

MULTU Multiply Unsigned MULTU



Format: MULTU rs, rt

MULTU rd, rs, rt

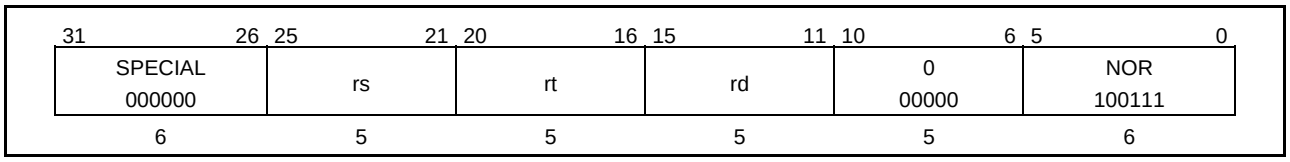
Description: Multiplies the contents of general-purpose register rs by the contents of general register rt, treating both register values as 32-bit unsigned values. This instruction cannot raise an integer overflow exception.
The low-order word of the multiplication result is put in general register rd and in special register LO, whereas the high-order word of the result is put in special register HI.
If rd is omitted in assembly language, 0 is used as the default value.

Operation:

```
T: t ← (0||GPR[rs])*(0||GPR[rt])
   LO ← t31~0
   HI ← t63~32
   GPR[rd] ← t31~0
```

Exceptions: None

NOR Nor NOR



Format: NOR rd, rs, rt

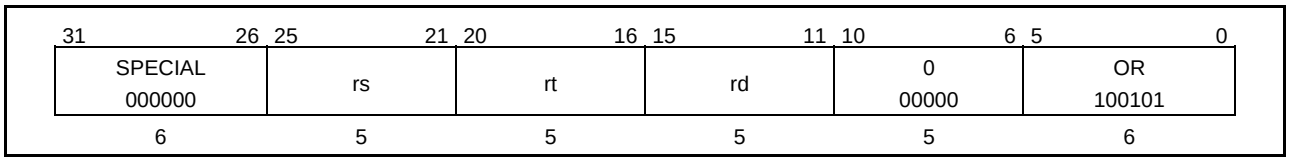
Description: Bitwise NORs the contents of general register rs with the contents of general register rt, and loads the result in general register rd.

Operation:

```
T: GPR[rd] ← GPR[rs] nor GPR[rt]
```

Exceptions: None

OR Or OR



Format: OR rd, rs, rt

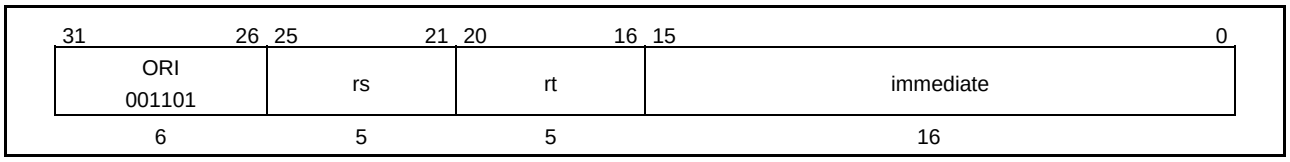
Description: Bitwise ORs the contents of general-purpose register rs with the contents of general-purpose register rt, and loads the result in general-purpose register rd.

Operation:

```
T: GPR[rd] ← GPR[rs] or GPR[rt]
```

Exceptions: None

ORI Or Immediate ORI



Format: ORI rt, rs, immediate

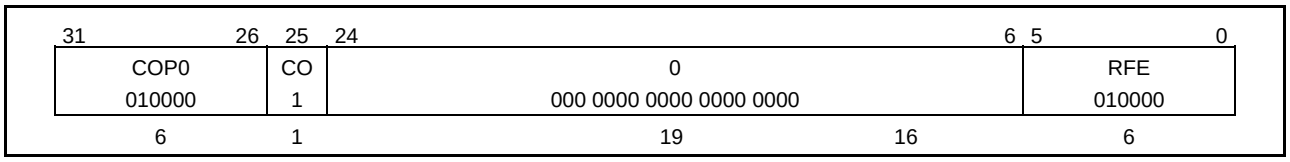
Description: Zero-extends the 16-bit immediate value, bitwise ORs the result with the contents of general-purpose register rs, and loads the result in general-purpose register rt.

Operation:

```
T: GPR[rt] ← GPR[rs]31~16 || (immediate or GPR[rs]15~0)
```

Exceptions: None

RFE Restore From Exception RFE



Format: RFE

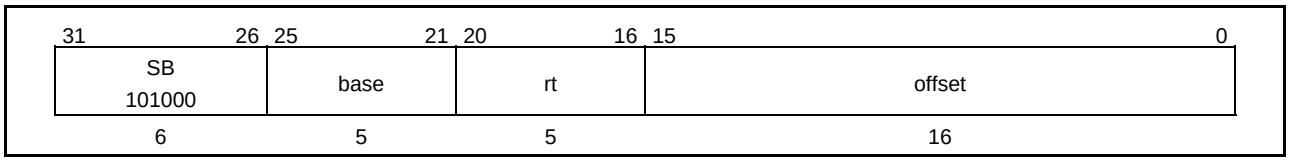
Description: Copies the Status register bits for previous interrupt mask mode and previous kernel/user mode (IEp and KUp) to the current mode bits (IEc and KUc), and copies the old mode bits (IEo and KUo) to the previous mode bits (IEp and KUp). The old mode bits remain unchanged.
Similarly, it copies the Cache register bits for previous data cache auto-lock mode and previous instruction cache auto-lock mode (DALp and IALp) to the current mode bits (DALc and IALc), and copies the old mode bits (DALo and IALo) to the previous mode bits (DALp and IALp). The old mode bits remain unchanged.
Normally an RFE instruction is placed in the delay slot after a JR instruction in order to restore the PC.

Operation:

```
T: Status ← Status31~4 || Status5~2
   Cache ← 0^8 || Cache13~12 || Cache13~0 || 0^8
```

Exceptions: Coprocessor Unusable exception

SB Store Byte SB



Format: SB rt, offset(base)

Description: Generates a 32-bit effective address by sign-extending the 16-bit offset and adding it to the contents of general-purpose register base. It then stores the least significant byte of register rt at the resulting effective address.

Operation:

```
T: vAddr ← ((offset15)^16 || offset15~0) + GPR[base]
   (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
   pAddr ← pAddr31~2 || (pAddr1~0 xor ReverseEndian^2)
   byte ← vAddr1~0 xor BigEndianCPU^2
   data ←GPR[rt]31-8*byte~0 || 0^8*byte
   StoreMemory (uncached, BYTE, data, pAddr, vAddr, DATA)
```

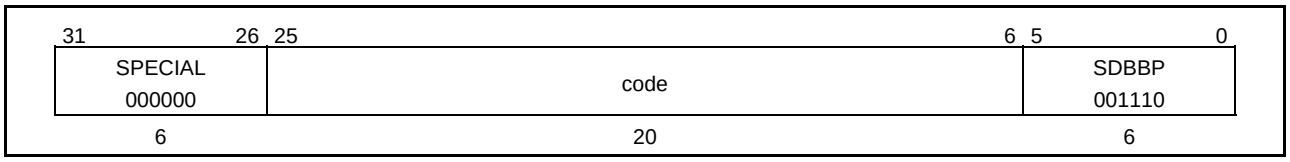
Exceptions: UTLB Refill exception (reserved)

TLB Refill exception (reserved)

TLB Modified exception (reserved)

Address Error exception

SDBBP Software Debug Breakpoint SDBBP



Format: SDBBP code

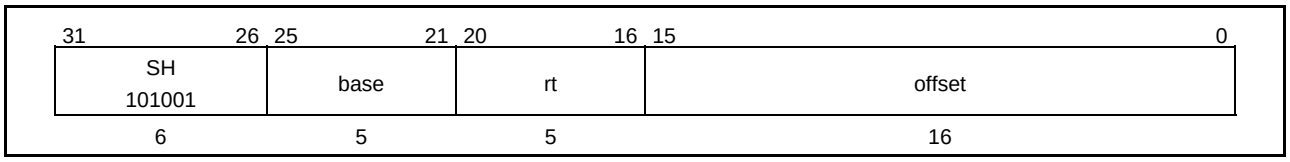
Description: Raises a Debug Breakpoint exception, passing control to an exception handler.
The code field can be used for passing information to the exception handler, but the only way to have the code field retrieved by the exception handler is to load the contents of the memory word containing this instruction using the DEPC register.

Operation:

```
T: Software DebugBreakpointException
```

Exceptions: Debug Breakpoint exception

SH Store Halfword SH



Format: SH rt, offset(base)

Description: Generates an unsigned 32-bit effective address by sign-extending the 16-bit offset and adding it to the contents of general-purpose register base. It then stores the least significant halfword of register rt at the resulting effective address. If the effective address is not aligned on a halfword boundary, that is if the least significant bit of the effective address is not 0, an Address Error exception is raised.

Operation:

```
T: vAddr ← ((offset15)^16 || offset15~0) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
pAddr ← pAddr31~2 ||(pAddr1~0 xor (ReverseEndian || 0))
byte ← vAddr1~0 xor (BigEndianCPU || 0)
data ←GPR[rt]31-8*byte~0 || 0^8*byte
StoreMemory (uncached, HALFWORD, data, pAddr, vAddr, DATA)
```

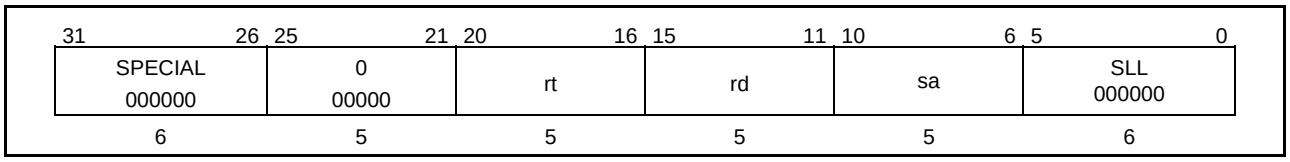
Exceptions: UTLB Refill exception (reserved)

TLB Refill exception (reserved)

TLB Modified exception (reserved)

Address Error exception

SLL Shift Left Logical SLL



Format: SLL rd, rt, sa

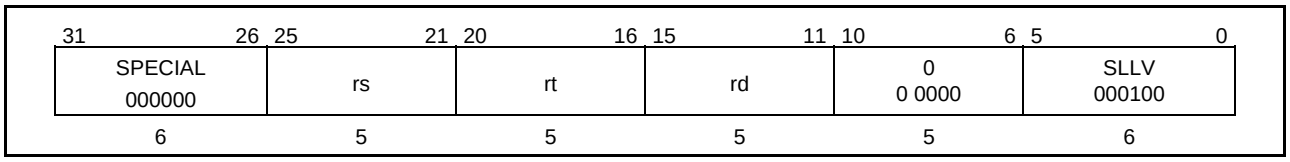
Description: Left-shifts the contents of general-purpose register rt by sa bits, zero-fills the low-order bits, and puts the result in register rd.

Operation:

```
T: GPR[rd] ← GPR[rt]31-sa~0 || 0 sa
```

Exceptions: None

SLLV Shift Left Logical Variable SLLV



Format: SLLV rd, rt, rs

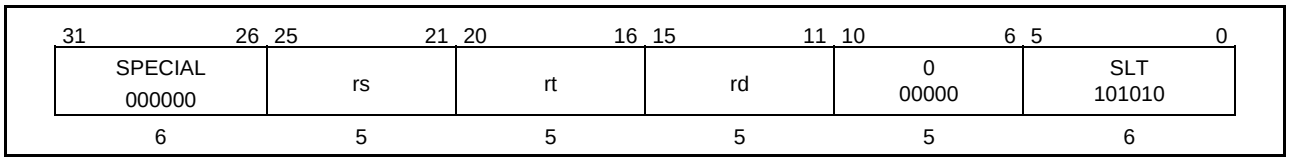
Description: Left-shifts the contents of general-purpose register rt (by the number of bits designated in the low-order five bits of general-purpose register rs), zero-fills the low-order bits and puts the 32-bit result in register rd.

Operation:

```
T: s ← GPR[rs]4~0
   GPR[rd] ← GPR[rt](31-s)~0 || 0^s
```

Exceptions: None

SLT Set On Less Than SLT



Format: SLT rd, rs, rt

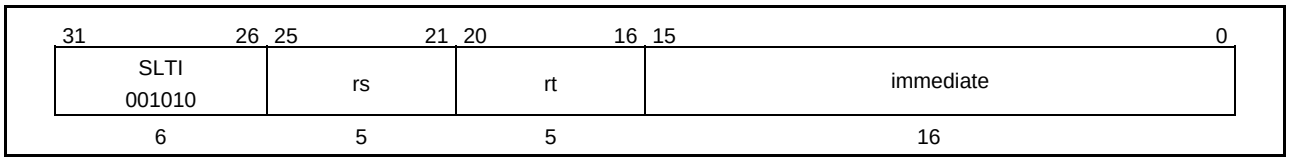
Description: Compares the contents of general-purpose registers rt and rs as 32-bit signed integers. A 1, if rs is less than rt, or a 0, otherwise, is placed in general-purpose register rd as the result of the comparison.
No overflow exception is raised. The comparison is valid even if the subtraction used in making the comparison overflows.

Operation:

```
T: if GPR[rs]< GPR[rt] then
       GPR[rd] ← 0^31 || 1
   else
       GPR[rd] ← 0^32
   endif
```

Exceptions: None

SLTI Set On Less Than Immediate SLTI



Format: SLTI rt, rs, immediate

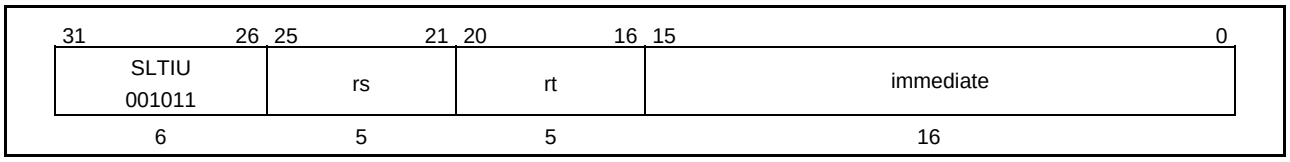
Description: Sign-extends the 16-bit immediate value and compares the result with the contents of general-purpose register rs, treating both values as 32-bit signed integers. A 1, if rs is less than the sigh-extended immediate value, or a 0, otherwise, is placed in general-purpose register rt as the result of the comparison.
No overflow exception is raised. The comparison is valid even if the subtraction used in making the comparison overflows.

Operation:

```
T: if GPR[rs]< (immediate15)16 || immediate15~0 then
       GPR[rd] ← 031 || 1
   else
       GPR[rd] ← 032
   endif
```

Exceptions: None

SLTIU Set On Less Than Immediate Unsigned SLTIU



Format: SLTU rt, rs, immediate

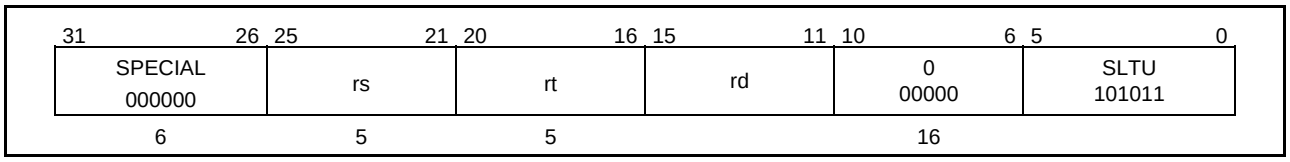
Description: Sign-extends the 16-bit immediate value and compares the result with the contents of general-purpose register rs, treating both values as 32-bit unsigned integers. A 1, if rs is less than the sigh-extended immediate value, or a 0, otherwise, is placed in general-purpose register rt as result of the comparison.
No overflow exception is raised. The comparison is valid even if the subtraction used in making the comparison overflows.

Operation:

```
T: if (0 || GPR[rs]< (0 || immediate15)^16 || immediate15~0) then
       GPR[rd] ← 0^31 || 1
   else
       GPR[rd] ← 0^32
   endif
```

Exceptions: None

SLTU Set On Less Than Unsigned SLTU



Format: SLTU rd, rs, rt

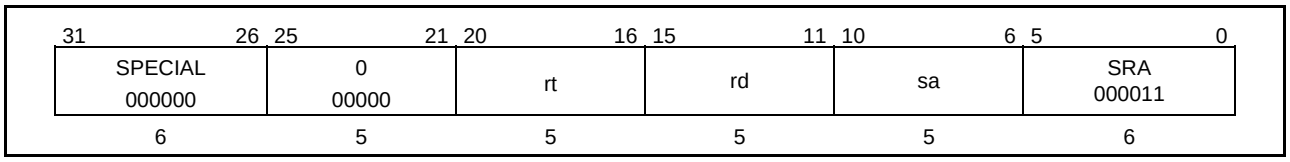
Description: Compares the contents of general registers rt and rs as 32-bit unsigned integers. A 1, if rs is less than rt, or a 0, otherwise, is placed in general-purpose register rd as the result of the comparison.
No overflow exception is raised. The comparison is valid even if the subtraction used in making the comparison overflows.

Operation:

```
T: if (0 || GPR[rs]) < (0 || GPR[rt]) then
       GPR[rd] ← 0^31 || 1
   else
       GPR[rd] ← 0^32
   endif
```

Exceptions: None

SRA Shift Right Arithmetic SRA



Format: SRA rd, rt, sa

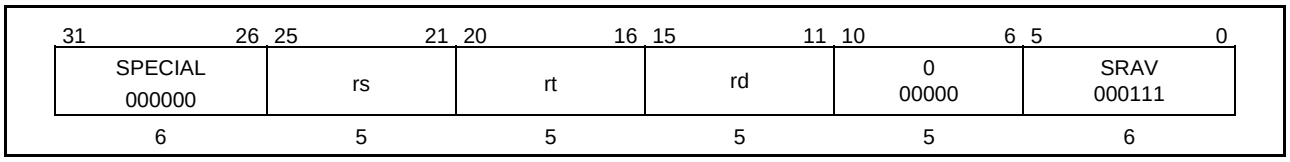
Description: Right-shifts the contents of general-purpose register rt by sa bits, sign-extends the high-order bits, and puts the result in register rd.

Operation:

```
T: GPR[rd] ← (GPR[rt]31)^sa || GPR[rt]31~sa
```

Exceptions: None

SRAV Shift Right Arithmetic Variable SRAV



Format: SRAV rd, rt, rs

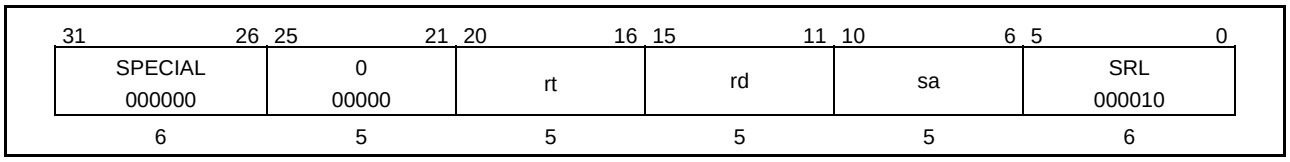
Description: Right-shifts the contents of general-purpose register rt (by the number of bits designated in the low-order five bits of general-purpose register rs), sign-extends the high-order bits, and puts the result in register rd.

Operation:

```
T: s ← GPR[rs]4~0
   GPR[rd] ← (GPR[rt]31)^s || GPR[rt]31~s
```

Exceptions: None

SRL Shift Right Logical SRL



Format: SRL rd, rt, sa

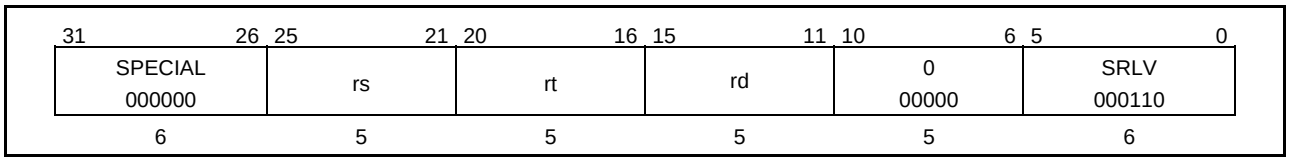
Description: Right-shifts the contents of general-purpose register rt by sa bits, zero-fills the high-order bits, and puts the result in register rd.

Operation:

```
T: GPR[rd] ← 0^sa || GPR[rt]31~sa
```

Exceptions: None

SRLV Shift Right Logical Variable SRLV



Format: SRLV rd, rt, rs

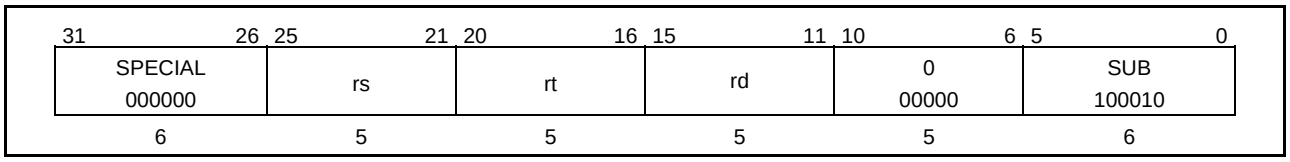
Description: Right-shifts the contents of general register rt (by the number of bits designated in the low-order five bits of general register rs), zero-fills the high-order bits, and puts the result in register rd.

Operation:

```
T: s ← GPR[rs]4~0
   GPR[rd] ← 0^s || GPR[rt]31~s
```

Exceptions: None

SUB Subtract SUB



Format: SUB rd, rs, rt

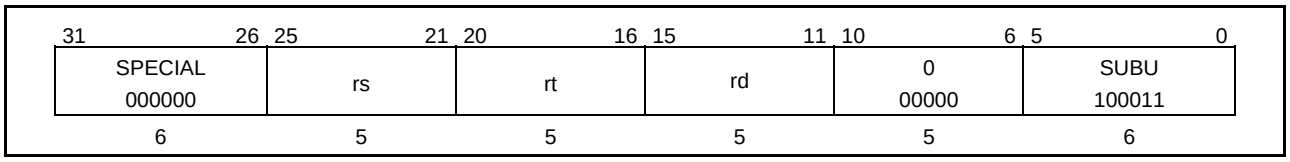
Description: Subtracts the contents of general-purpose register rt from general-purpose register rs and puts the result in general-purpose register rd. If carry-out bits 31 and 30 differ, a two's complement overflow exception is raised and destination register rd is not modified.

Operation:

```
T: GPR[rd] ← GPR[rs] - GPR[rt]
```

Exceptions: Overflow exception

SUBU Subtract Unsigned SUBU



Format: SUBU rd, rs, rt

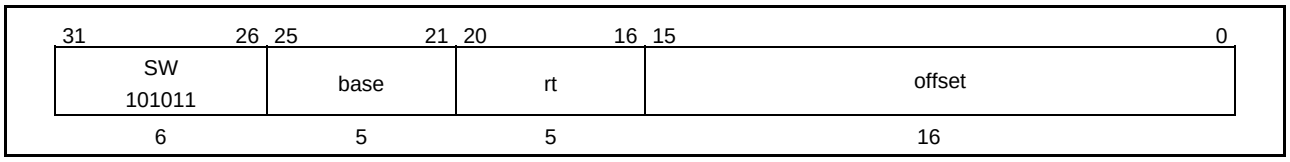
Description: Subtracts the contents of general-purpose register rt from general-purpose register rs and puts the result in general-purpose register rd. The only difference from SUB is that SUBU cannot cause an overflow exception.

Operation:

```
T: GPR[rd] ← GPR[rs] - GPR[rt]
```

Exceptions: None

SW Store Word SW



Format: SW rt, offset(base)

Description: Generates a 32-bit effective address by sign-extending the 16-bit offset value and adding it to the contents of general-purpose register base. It then stores the contents of register rt at the resulting effective address.
If the effective address is not aligned on a word boundary, that is, if the low-order two bits of the effective address are not 00, an Address Error exception is raised.

Operation:

```
T: vAddr ← ((offset15)^16 || offset15~0) + GPR[base]
   (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
   data ←GPR[rt]
   StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
```

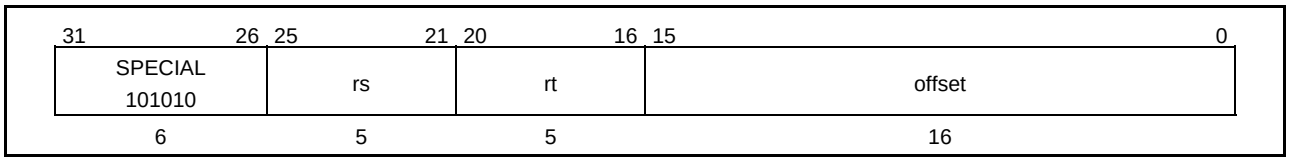
Exceptions: UTLB Refill exception (reserved)

TLB Refill exception (reserved)

TLB Modified exception (reserved)

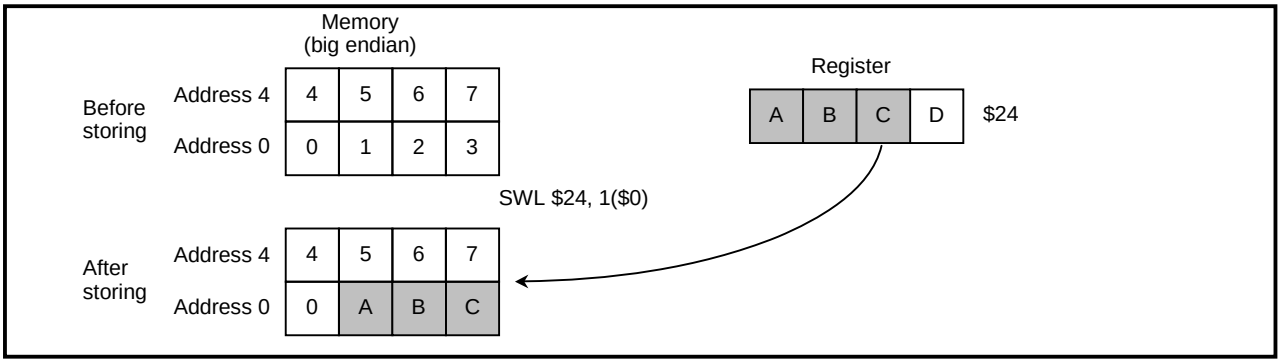
Address Error exception

SWL Store Word Left SWL



Format: SWL rt, offset(base)

Description: Used together with SWR to store the contents of a register into four consecutive bytes of memory when the bytes cross a word boundary. SWL stores the left part of the register into the appropriate part of the high-order word in memory; SWR stores the right part of the register into the appropriate part of the low-order word in memory.
This instruction generates a 32-bit effective address that can point to any byte by sign-extending the 16-bit offset and adding it to the contents of general-purpose register base. Only the one word in memory containing the designated starting byte is modified. Depending on the starting byte, from one to four bytes are stored.
The concept is illustrated below. This instruction (SWL) starts from the high-order (left-most) byte of the register and stores it into the designated memory byte; it then continues storing bytes from register to memory, proceeding toward the low-order byte of the register and the low-order byte of the memory word, until it reaches the low-order byte of the memory word.
No Address Error instruction is raised due to misalignment.



SWL Store Word Left (cont.) SWL

Operation:

```
T: vAddr ← ((offset15)16 || offset15~0) + GPR[base]
   (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
   pAddr ← pAddr31~2 || (pAddr1~0 xor ReverseEndian2)
   If BigEndianMem = 0 then
        pAddr ← pAddr31~2 || 02
   endif
   byte ← vAddr1~0 xor BigEndianCPU2
   data ← 0 24 - 8*byte || GPR[rt]31~24-8*byte
   StoreMemory (uncached, byte, data, pAddr, vAddr, DATA)
```

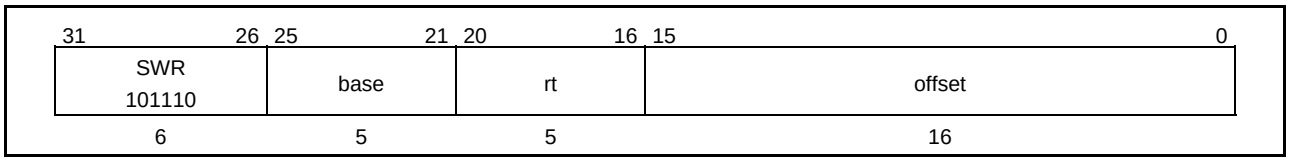
Exceptions: UTLB Refill exception (reserved)

TLB Refill exception (reserved)

TLB Modified exception (reserved)

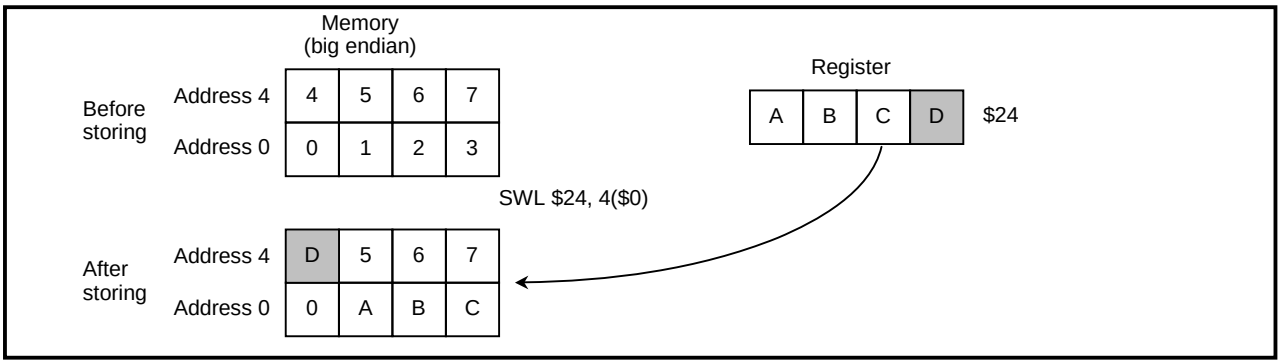
Address Error exception

SWR Store Word Right SWR



Format: SWR rt, offset(base)

Description: Used together with SWL to store the contents of a register into four consecutive bytes of memory when the bytes cross a word boundary. SWR stores the right part of the register into the appropriate part of the low-order word in memory; SWL stores the left part of the register into the appropriate part of the high-order word in memory.
This instruction generates a 32-bit effective address that can point to any byte by sign-extending the 16-bit offset and adding it to the contents of general-purpose register base.
Only the one word in memory containing the designated starting byte is modified.
Depending on the starting byte, from one to four bytes are stored.
The concept is illustrated below. This instruction (SWR) starts from the low-order (right-most) byte of the register and stores it into the designated memory byte; it then continues storing bytes from register to memory, proceeding toward the high-order byte of the register and the high-order byte of the memory word, until it reaches the high-order byte of the memory word.
No Address Error instruction is raised due to misalignment.



SWR Store Word Right (cont.) SWR

Operation:

```
T: vAddr ← ((offset15)16 || offset15~0) + GPR[base]
   (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
   pAddr ← pAddr31~2 || (pAddr1~0 xor ReverseEndian2)
   If BigEndianMem = 0 then
        pAddr ← pAddr31~2 || 02
   endif
   byte ← vAddr1~0 xor BigEndianCPU2
   data ← GPR[rt]31-8*byte || 08*byte
   StoreMemory (uncached, WORD-byte, data, pAddr, vAddr, DATA)
```

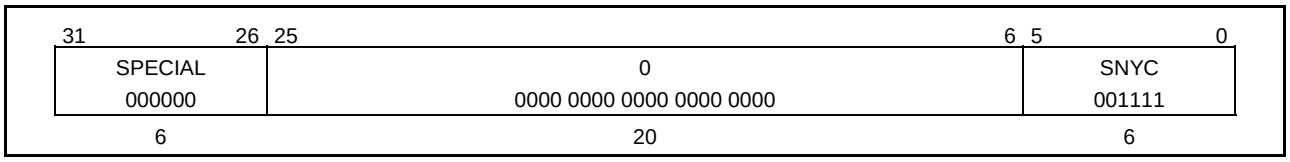
Exceptions: UTLB Refill exception (reserved)

TLB Refill exception (reserved)

TLB Modified exception (reserved)

Address Error exception

SYNC Synchronize SYNC



Format: SYNC

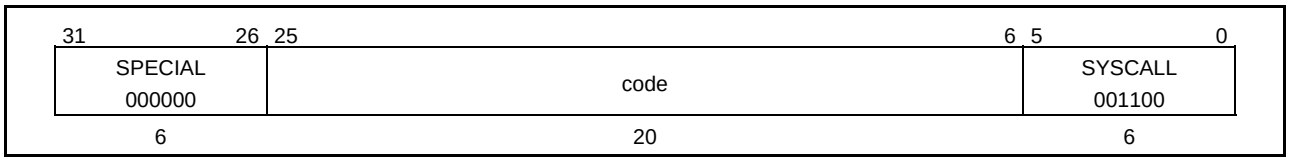
Description: Interlocks the pipeline until the load, store or data cache refill operation of the previous instruction is completed.
The TX39 Processor Core can continue processing instructions following a load instruction even if a cache refill is caused by the load instruction or a load is made from a noncacheable area.
Executing a SYNC instruction interlocks subsequent instructions until the SYNC instruction execution is completed. This ensures that the instructions following a load instruction are executed in the proper sequence.
This instruction is valid in user mode.

Operation:

```
T: SyncOperation()
```

Exceptions: None

SYSCALL System Call SYSCALL



Format: SYSCALL code

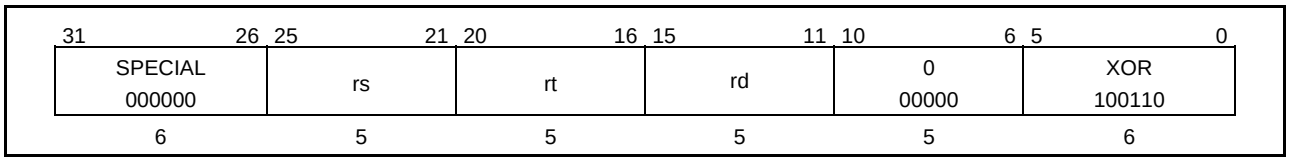
Description: Raises a System Call exception, then immediately passes control to an exception handler. The code field can be used to pass information to an exception handler, but the only way to have the code field retrieved by the exception handler is to use the EPC register to load the contents of the memory word containing this instruction.

Operation:

```
T: SystemCallException
```

Exceptions: System Call exception

XOR Exclusive Or XOR



Format: XOR rd, rs, rt

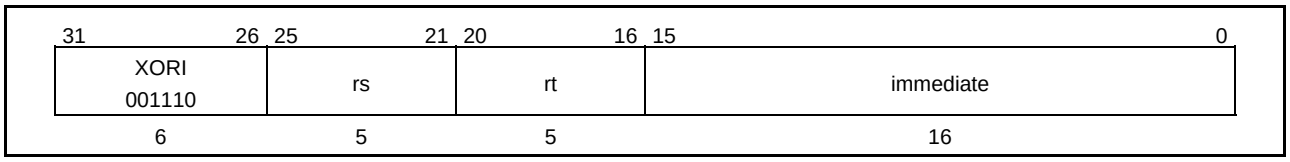
Description: Bitwise exclusive-ORs the contents of general-purpose register rs with the contents of general-purpose register rt and loads the result in general-purpose register rd.

Operation:

```
T: GPR[rd] ← GPR[rs] xor GPR[rt]
```

Exceptions: None

XORI Exclusive Or Immediate XORI



Format: XORI rt, rs, immediate

Description: Zero-extends the 16-bit immediate value, bitwise exclusive-ORs it with the contents of general-purpose register rs, then loads the result in general-purpose register rt.

Operation:

```
T: GPR[rt] ← GPR[rs] xor (0^16 || immediate)
```

Exceptions: None

Bit Encoding of CPU Instruction Opcodes

Figure A-2 shows the bit codes for all CPU instructions (ISA and extended ISA).

OPcode

31~29 \ 28~26	0	1	2	3	4	5	6	7
0	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	COP0	COP1	COP2	COP3	BEQLδ	BNELδ	BLEZLδ	BGTZLδ
3	*	*	*	*	MADD/ MADDUδ	*	*	*
4	LB	LH	LWL	LW	LBU	LHU	LWR	*
5	SB	SH	SWL	SW	*	*	SWR	CACHEδ
6	*	ξ	ξ	ξ	*	*	*	*
7	*	ξ	ξ	ξ	*	*	*	*

SPECIAL function

5~3 \ 2~0	0	1	2	3	4	5	6	7
0	SLL	*	SRL	SRA	SLLV	*	SRLV	SRAV
1	JR	JALR	*	*	SYSCALL	BREAK	SDBBPδ	SYNCδ
2	MFHI	MTHI	MFLO	MTLO	*	*	*	*
3	MULT	MULTU	DIV	DIVU	*	*	*	*
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	*	*	SLT	SLTU	*	*	*	*
6	*	*	*	*	*	*	*	*
7	*	*	*	*	*	*	*	*

BCOND

20~19 \ 18~16	0	1	2	3	4	5	6	7
0	BLTZ	BGEZ	BLTZLχ	BGEZLχ	γ	γ	γ	γ
1	γ	γ	γ	γ	γ	γ	γ	γ
2	BLTZAL	BGEZAL	BLTZALLχ	BGEZALLχ	γ	γ	γ	γ
3	γ	γ	γ	γ	γ	γ	γ	γ

COPz rs

25,24 \ 23~21	0	1	2	3	4	5	6	7
0	MF	γ	CF	γ	MT	γ	CT	γ
1	BC	γ	γ	γ	γ	γ	γ	γ
2	CO							
3								

COPz rt

20~19 \ 18~16	0	1	2	3	4	5	6	7
0	BCF	BCT	BCFLχ	BCTLχ	γ	γ	γ	γ
1	γ	γ	γ	γ	γ	γ	γ	γ
2	γ	γ	γ	γ	γ	γ	γ	γ
3	γ	γ	γ	γ	γ	γ	γ	γ

Figure A-2. Operation Code Bit Encoding

CP0 Function

5~3 \ 2~0	0	1	2	3	4	5	6	7
0	ϕ	(TLBR) ϕ	(TLBWI) ϕ	ϕ	ϕ	ϕ	(TLBWR) ϕ	ϕ
1	(TLBP) ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
2	RFE	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
3	*	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	DERETχ
4	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
5	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
6	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
7	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ

MADD/MADDU

5~3 \ 2~0	0	1	2	3	4	5	6	7
0	MADD	MADDU	γ	γ	γ	γ	γ	
1	γ	γ	γ	γ	γ	γ	γ	γ
2	γ	γ	γ	γ	γ	γ	γ	γ
3	γ	γ	γ	γ	γ	γ	γ	γ
4	γ	γ	γ	γ	γ	γ	γ	γ
5	γ	γ	γ	γ	γ	γ	γ	γ
6	γ	γ	γ	γ	γ	γ	γ	γ
7	γ	γ	γ	γ	γ	γ	γ	γ

Figure A-2. Operation Code Bit Encoding (cont)

Notation :

- * Reserved for future architecture implementations; use of this instruction with existing versions raises a Reserved Instruction exception.
- γ Invalid instruction, but dose not raise Reserved Instruction exception in the case of the TX39 Processor Core.
- δ Valid on the TX39 Processor Core but raises a Reserved Instruction exception on the R3000A.
- ϕ Reserved for memory management unit (MMU). Dose not raise a Reserved Instruction exception in the case of the TX39 Processor Core.
- ξ Raises a Reserved Instruction exception. Valid on the R3000A.
- χ Valid on the TX39 Processor Core but invalid on the R3000A.