CorExtend® Instruction Integrator's Guide for M4K®/4KE®/4KS™ and M14K™ Family Cores

Document Number: MD00324
Revision 01.05
June 12, 2011

MIPS Technologies, Inc.
955 East Arques Avenue
Sunnyvale, CA 94085-4521

Contents

Preface

This document describes the integration of the CorExtend® user-defined instruction (UDI) module for:

- MIPS32® Pro Series® processor cores, including the 4KE® Pro core family, the 4KSd™ Pro, and the M4K® Pro cores.

- MIPS32® M14K™ family cores, which support the microMIPS™ Instruction Set Architecture.

The document defines the UDI interface available to a core integrator, and how the UDI module interacts with the rest of the processor pipeline.

The CorExtend capability allows the core's performance to be tailored for specific applications, while still maintaining the benefits of the industry-standard MIPS32® instruction set architecture. By extending the instruction set with custom instructions, UDIs can enable significant performance improvement in critical algorithms beyond what is achievable with standard MIPS32 instructions.

A simple block diagram of a CorExtend UDI block is shown in Figure 1.

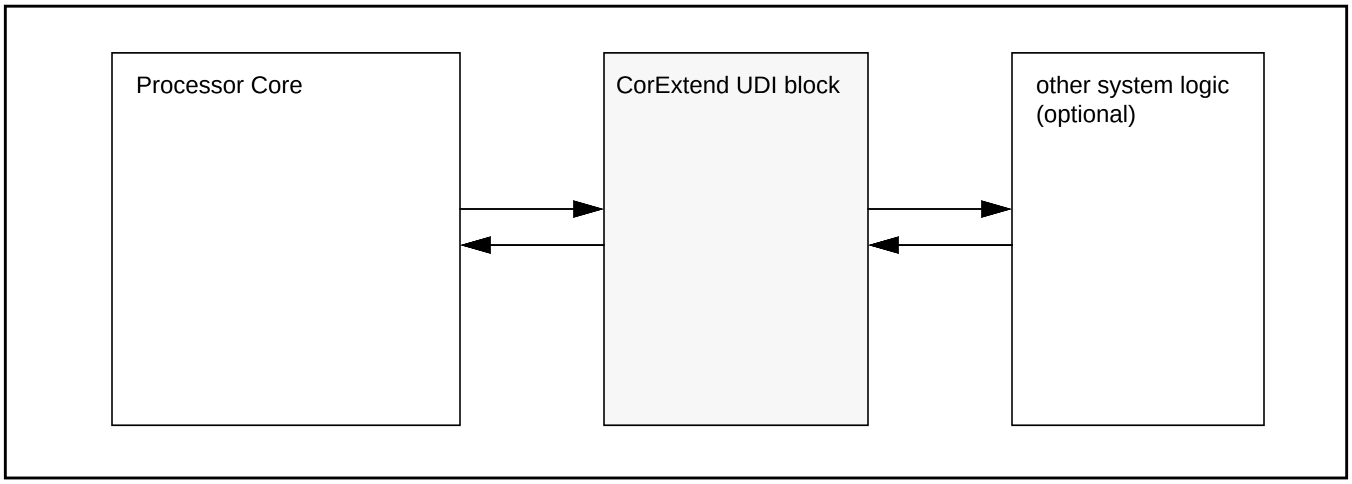

Figure 1 CorExtend® Block Diagram

The CorExtend UDI function is partitioned externally to the core pipeline, which makes it possible to add new instructions by modifying only the CorExtend block. This enables new instructions to be added to a previously hardened core, or can allow a UDI block to access external system logic.

The remainder of this document covers the details of integrating CorExtend instructions with the core RTL. Because the CorExtend interface is tightly coupled to the core pipeline, the reader should first be familiar with the operation of the pipeline as described in the *Software User's Manual* for the appropriate processor core (References [1], [2], [3], [4], [5]).

1 CorExtend® Instruction Specification

This section describes the formats and requirements of CorExtend user-defined instructions (UDI).

1.1 User-Defined Instruction Format for MIPS32® ISA

The general format of a user-defined instruction is shown below:

31 26	25 21	20 16	15 6	5 0
SPECIAL2 011100	rs (optional)	rt (optional)	user-interpretable	01xxxx
6	5	5	10	6

A subset of the SPECIAL2 instructions are allowed for user-defined instructions. A SPECIAL2 instruction is specified when the Opcode field in bits [31:26] of the instruction are 011100. Then the Function field in bits [5:0] of the instruction further defines the SPECIAL2 instruction type. For UDI instructions, bits [5:4] must be 01 and bits [3:0] are available to encode the user's instruction opcode. All other bits in the instruction word are available to the user. Based solely on the encoding of bits [3:0] in the instruction word, 16 UDIs are available to the user. But more than 16 UDIs could be defined if the user encodes the instruction type in other available bits of the instruction word.

1.2 User-Defined Instruction Format for microMIPS™ ISA

If microMIPS ISA is supported, the general format of a user defined instruction is shown below :

31 26	25 6	5 0
POOL32A 0000000	user-interpretable	xxx011
6	20	6

The microMIPS user-defined instruction format is similar to that of MIPS32, but with the following differences:

- The SPECIAL2 major opcode is not used and is replaced by the microMIPS POOL32A major opcode.
- The minor opcode bits are in bits [2:0] are 011.
- The microMIPS ISA supports 7 user-defined instructions through the encoding of bits [5:3] in the instruction word, but more user-defined instructions can be defined if the user encodes the instruction type in the other available bits of the instruction word.

1.3 User-Defined Instruction Requirements

The user-defined instructions should meet the following requirements:

- Only fixed integer instructions are permitted. No jumps, branches, loads or stores are allowed. Multiply or divide operations are permitted, but they may not touch the HI/LO register pair within the MDU.
- The destination of the instruction may be a general-purpose register, or a register inside the UDI block.

- Instructions with a destination of a general-purpose register can complete in a single cycle or multiple cycles. In the case of multi-cycle latency, a signal output from the UDI block is used to stall the core pipeline until the UDI operation completes. Instructions with a destination of a register internal to the UDI block can be fully pipelined, largely independent of the core pipeline. These instructions do not stall the core, and are not stalled by the core, but must obey certain core signals before committing their final results to the internal destination register.

- Instructions can have 2 general-purpose registers as operands, or 1 register and 1 immediate, or internal UDI registers, or any other combination as desired by the user. If general-purpose register operands are desired, then rs must be encoded in bits [25:21] and rt must be encoded in bits [20:16] of the instruction word. The core will always provide the register contents specified by these locations as inputs to the UDI block, but they can be ignored if they are not used. If either rs or rt register operands are not needed, then these bits can also be interpreted freely by the user. Note that a pipeline interlock may occur with these register values when an earlier instruction has a destination that corresponds to the bits in the rs or rt fields, even if the register operands are unused by the UDI block.

- The destination register can be derived from any bits in the instruction format, since the UDI block provides the destination register value back to the core. The signalling of the destination register must occur through combinational decode of the instruction word in the same cycle, to allow the core to check for register dependencies of the UDI destination with source register(s) of the following instruction. If the destination register is inside the UDI block, R0 should be sent to the core as the destination register.

NOTE: Allowing an arbitrary destination register is a feature specific to these cores. This feature is not generally scalable for higher performance pipelines and is not likely to be incorporated on future cores from MIPS Technologies. It is strongly recommended that the destination register be rd, rt, or an internal register. The use of other destination registers may limit the reusability of the UDI with future cores.

- The full 32-bit instruction word is sent to the UDI block. The UDI block is responsible for signaling which of the 16 UDIs, as determined by bits [3:0] of the instruction word, are not implemented. The core will indicate a Reserved Instruction exception if the UDI is not implemented.

- A CorExtend Unusable exception can be indicated by the core when a UDI operation is attempted, if the CP0 *Status.CEE* bit is not set and the *UDI_honor_cee* signal is asserted.

- No other execution exceptions may be generated by a user-defined instruction.

- The CP0 *Config0.UDI* bit (bit [22]) indicates whether user-defined instructions are implemented in the processor. This bit is automatically set based on the value of the *UDI_present* output from the UDI block.

- Tool support:
  - Automatic inclusion in the assembler.
  - Intrinsic and inlining support in the compiler.
  - No direct compiler inference.
  - MIPSsim™ simulator support.

1.4 Sample User-Defined Instruction Formats for MIPS32

The general format of a user-defined instruction, as shown in Section 1.1, "User-Defined Instruction Format for MIPS32® ISA", allows the user a large degree of flexibility in choosing how the available bits of the instruction word (bits [25:6]) are used to encode UDIs. This subsection describes some possible formats which may be used, but is not intended to be exhaustive.

Three general-purpose register operands plus 5 immediate bits

A UDI format with three register-based operands is shown below:

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL2 011100	rs	rt	rd	user-interpretable	01xxxx
6	5	5	5	5	6

There are two source general-purpose register operands, encoded in the rs and rt fields, while the destination general-purpose register is encoded in the rd field. In addition, 5 bits are available to pass immediate data, or encode internal UDI register information, if desired. Immediate data could be used to specify additional source operand information, or to further encode information about the type of user-defined instruction to be executed. UDI register information could be used to specify additional sources from internal registers, or additional destinations to internal registers.

Two general-purpose register operands plus 10 immediate bits

A UDI format with two register operands is shown below:

31 26	25 21	20 16	15 6	5 0
SPECIAL2 011100	rs	rt	user-interpretable	01xxxx
6	5	5	10	6

There are two source general-purpose register operands, encoded in the rs and rt fields. In addition, 10 bits are available to pass immediate data, encode a destination general-purpose register, or encode internal UDI register information. With this format, the destination register could be an internal UDI register, or overwrite one of the source operands, either rs or rt, if desired. The internal UDI register information could also be used to specify additional sources from internal registers.

One general-purpose register operand plus 15 immediate bits

A UDI format with one register operand is shown below:

31 26	25 21	20 6	5 0
SPECIAL2 011100	rs	user-interpretable	01xxxx
6	5	15	6

There is a single source general-purpose register operand, encoded in the rs field. In addition, 15 bits are available to pass immediate data, or encode a destination general-purpose register, or encode internal UDI register information. With this format, the destination register could be an internal UDI register, or overwrite the rs source operand, if desired. The internal UDI register information could also be used to specify additional sources from internal registers.

No general-purpose register operands plus 20 immediate bits

A UDI format with no direct register operands is shown below:

31 26	25 6	5 0
SPECIAL2 011100	user-interpretable	01xxxx
6	20	6

In this format, all 20 bits are available to be interpreted by the user. This format might be used if the source operands for the instruction do not come from general-purpose registers, but instead come from internal UDI registers, or are specified as immediate data within the 20 user-interpretable bits. In addition, the destination register would also be encoded somewhere within the 20 user-definable bits.

2 Incorporating CorExtend® UDIs into the RTL

This section describes some of the considerations for incorporating CorExtend user-defined instructions with the rest of the processor core RTL. Review the *Implementor's Guide* document for the appropriate MIPS32® processor core (References [8], [9], [10]) or the MIPS32® M14K Family processor cores (References [11], [12]) for more general details about the implementation of a soft core.

The CorExtend interface to implement user-defined instructions on a processor core is external to the core itself. Although the CorExtend interface is external, it is tightly coupled to the execution unit within the core. When implementing UDIs, adherence to a specific internal interface must be followed. The functionality and timing of the UDI interface is described in later sections.

2.1 CorExtend RTL Modules

The location of the CorExtend UDI module within the RTL hierarchy is shown in Figure 2. The `m4k_udi_custom` or `m14k_udi_custom` module is the primary location containing the description of CorExtend instructions. If UDI interaction with external system logic is also desired, then variable width input and output external busses are provided so these signals can be added without the need to modify the `m4k_top` or `m14k_top` module itself.

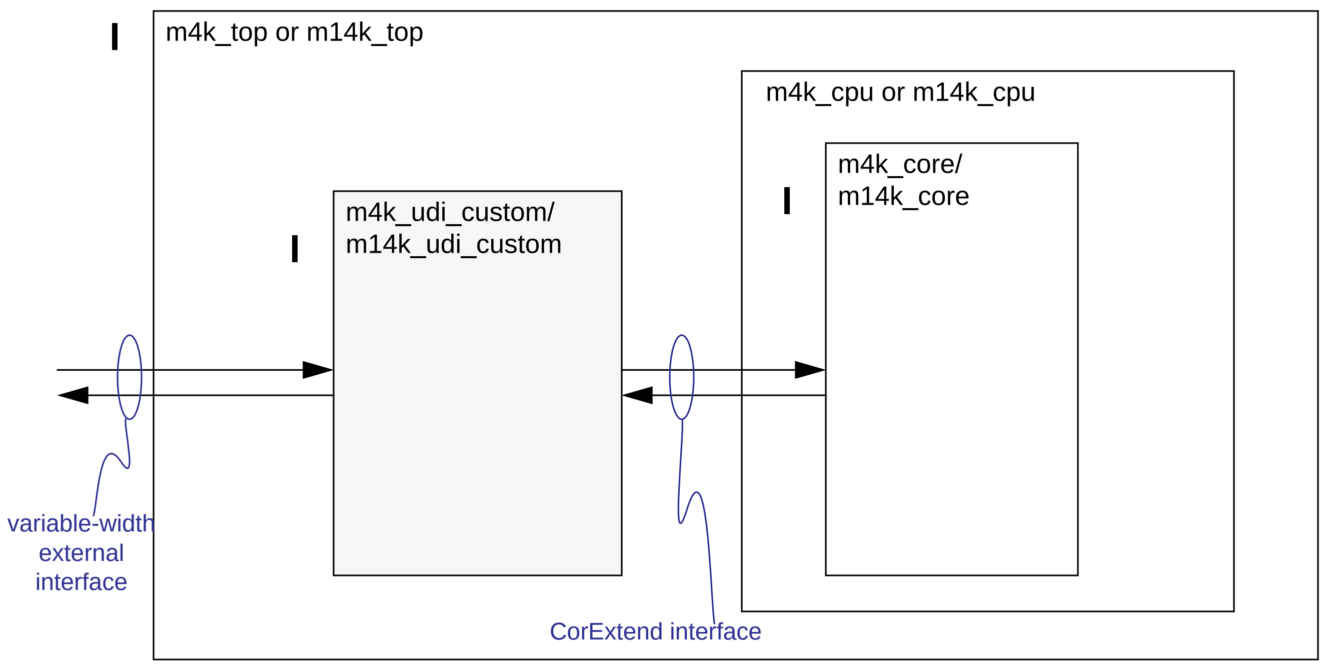

Figure 2 CorExtend® Module Location in RTL Hierarchy

If CorExtend instructions are desired in a processor core, their RTL description must be located in a Verilog module named `m4k_udi_custom` or `m14k_udi_custom`. The module must be located in a file called `m4k_udi_custom.v` or `m14k_udi_custom.v`, which resides in the directory `$MIPS_PROJECT/proc/design/rtl`. Several template files, showing the required interface signals and example implementations, are provided in the release. See Section 6, "CorExtend® UDI Sample Implementations" for more details about the included examples. The implementor should modify or copy a template file to create the Verilog description of the desired UDIs.

If UDI operations are not desired, the `m4k_udi_custom` or `m14k_udi_custom` module can be replaced with a stub module called `m4k_udi_stub` or `m14k_udi_stub`. This stub module, which drives the correct default values to the core interface, is provided in the RTL release.

2.1.1 Module m4k_udi_custom/m14k_udi_custom

The `m4k_udi_custom/m14k_udi_custom` module is instantiated inside the module `m4k_top/m14k_top`, a top-level module that also instantiates the actual processor core (`m4k_cpu/m14k_cpu`). Logically, the CorExtend interface straddles the E and M stages of the pipeline. A description of the signal interface of the `m4k_udi_custom/ m14k_udi_custom` module that must be maintained by the implementor is covered in Section 3, "CorExtend® UDI Signal Interface".

When creating an RTL description for the `m4k_udi_custom/m14k_udi_custom` block, additional module hierarchy can be used as long as the additional modules are defined in the same `m4k_udi_custom.v/ m14k_udi_custom.v` file. Hierarchy defined in this manner is automatically picked up by the simulation and synthesis scripts provided in the core release.

2.1.2 Module m4k_udi_stub/m14k_udi_stub

The `m4k_udi_stub/m14k_udi_stub` module is a default module to the core interface and used when the core implementation does not contain any user-defined instructions. It contains an identical signal port list as the `m4k_udi` or `m14k_udi` module. The stub module sets the output *UDI_ri_e* to 1 and all other outputs to 0. It leaves all the inputs unconnected.

2.1.3 Reference CorExtend Modules

Several example CorExtend UDI modules are included in a processor core RTL distribution. These examples demonstrate use of the CorExtend interface, and are described in Section 6, "CorExtend® UDI Sample Implementations". The example modules can be substituted for the `m4k_udi_custom` or `m14k_udi_custom` module shown in Figure 2.

2.2 Top-level Modules m4k_top/m14k_top and m4k_cpu/m14k_cpu

The `m4k_top/m14k_top` module is essentially a wrapper that instantiates `m4k_udi_custom/ m14k_udi_custom` (or `m4k_udi_stub/m14k_udi_stub`) along with the processor core in `m4k_cpu/ m14k_cpu`. For basic CorExtend instructions, the `m4k_top/m14k_top` module should not need modification.

The interface between the `m4k_udi_custom/ m14k_udi_custom` CorExtend module and `m4k_top/ m14k_top` can accommodate interaction with other logic in the system. To facilitate the external communication without the need to modify `m4k_top/m14k_top`, two programmable-width buses are already included in the `m4k_top/m14k_top/m14k_udi_custom` modules. An external input bus, *UDI_toudi*, and output bus, *UDI_fromudi*, are also present. Their widths can be set as desired using Verilog *'define* commands, which is usually performed via the configuration graphical user interface provided with the soft core deliverables. Note that constraints on these ports will need to be set during preparation for synthesis.

The synthesis flows provided by MIPS in the full soft core package target either `m4k_top/m14k_top` or `m14k_cpu/ m14k_cpu` as the top-level module. When targeting the `m4k_top/m14k_top`, the synthesis of the custom CorExtend block is rolled into the synthesis flow for the rest of the core. This generally improves synthesis quality, without the need to express detailed constraints on the CorExtend interface between the CorExtend block and the rest of the core. With `m4k_top/m14k_top` as the synthesis target, the CorExtend functionality must of course be defined when the processor core is built.

Alternatively, `m4k_cpu`/`m14k_cpu` can be chosen as the synthesis target. In this case, the CorExtend block is external to the processor synthesis, so constraints must be specified for the CorExtend interface signals. The `m4k_cpu`/`m14k_cpu` synthesis target might be useful when creating a hard core to which various CorExtend blocks might be connected later. Care should be taken in defining the interface constraints as the core is hardened, however, since the interface signals are not fully registered. When `m4k_cpu`/`m14k_cpu` is used as the synthesis target for the processor core itself, a CorExtend block would then be synthesized and implemented stand-alone, or perhaps in combination with other system logic.

3 CorExtend® UDI Signal Interface

Signals related to the CorExtend UDI are described in Table 1. Signal directions are relative to the CorExtend module, not the processor core. Signals are generally asserted high. Signals with no bit range specified are one bit wide. All signal names include a prefix, *UDI_*, to designate their status as CorExtend UDI interface signals. Many names contain a suffix, *_e* or *_m*, indicating which pipeline stage the signal is related.

Timing is shown relative to the appropriate pipeline stage for each signal, when synthesizing the core for maximum frequency. The cycle is divided roughly in third, and signal timings are binned based on whether inputs are available or outputs are needed early, mid or late in each cycle. The relative timing can be relaxed when targeting the core for less than its maximum frequency.

3.1 Interface Between CorExtend Block and Core

Table 1 CorExtend UDI Interface Signals

Name	Direction	Relative Timing	Description
UDI_ir_e[31:0]	Input	early	This is the complete instruction word. Although the module also gets rs and rt source operands, the full instruction is provided so all or part of the source register fields may be used to hold immediate values. Note that the implementer is responsible for decoding the Opcode and Function fields.
UDI_irvalid_e	Input	early	Indicates whether the value of the instruction word (*UDI_ir_e*) is valid or not.
UDI_rs_e[31:0]	Input	mid	Source operand rs after the bypass mux.
UDI_rt_e[31:0]	Input	mid	Source operand rt after the bypass mux.
UDI_endianb_e	Input	early	Indicates that this instruction is executing in Big Endian mode. This signal is generally not needed unless a) the UDI instruction works on sub-word data that is endian dependent, and b) the UDI block is designed to be bi-endian
UDI_kd_mode_e	Input	early	Indicates that the instruction is executing in kernel or debug mode. This can be used to prevent certain UDI instructions from being executed in user mode.
UDI_kill_m	Input	late	Late arriving kill signal due to an exception generated by an earlier instruction. This signal may optionally be used to deassert the *UDI_stall_m* output for improved interrupt latency on multi-cycle UDIs whose results won't be used.
UDI_start_e	Input	late	This is the *mpc_run_ie* signal coming from the core pipeline control logic.
UDI_run_m	Input	late	This is the *mpc_run_m* signal used to qualify *UDI_kill_m*.
UDI_greset	Input	mid	Reset signal to be used to reset any state machines.
UDI_gclk	Input	N/A	Clock input.

Table 1 CorExtend UDI Interface Signals (Continued)

Name	Direction	Relative Timing	Description
UDI_gscanenable	Input	N/A	Global scan enable.
UDI_ri_e	Output	mid	A one bit signal which when high indicates that the SPECIAL2 instruction currently being executed is illegal (i.e., reserved). This signal is used by the Master Pipeline Control (MPC) block within the core to signal an illegal instruction, however, this signal is sampled by MPC only if the current instruction is within the SPECIAL2 range of user-defined instructions (bits [5:4] of the instruction are 2'b01).
UDI_rd_m[31:0]	Output	mid	The 32 bit result of the executed instruction available in the M stage.
UDI_wrreg_e[4:0]	Output	late	Register to write the result from the execution of this user-defined instruction. This value is also passed on to mpc.
UDI_stall_m	Output	mid	Signals that the UDI block is processing a multicycle instruction and needs to stall the pipeline since the outputs need to be written into the register file. Should be set to 0 for single-cycle instructions. This is an M stage signal.
UDI_present	Output	static	Static signal that denotes whether any UDI support is available.
UDI_honor_cee	Output	static	Indicates whether the core should honor the CorExtend Enable (CEE) bit contained in the *Status* register. When this signal is asserted, *Status.CEE* is deasserted, and a UDI operation is attempted, the core will take a CorExtend Unusable Exception.

3.2 External Interface

If the CorExtend block requires interface signals to an external block outside of `m4k_top` or `m14k_top`, it can do so using the predefined top-level input and output ports described in Table 2. The width of each of these port is configurable through the configuration GUI. These signals are entirely outside of the processor core.

Table 2 CorExtend™ External Interface

Name	Direction	Relative Timing	Description
UDI_toudi[N-1:0]	Input	N/A	External input to CorExtend block.
UDI_fromudi[N-1:0]	Output	N/A	Output from CorExtend block.

3.3 Relative Timing of I/O Signals to UDI Module

Figure 3 depicts the relative timing of inputs and outputs to and from the UDI block, within the pipeline cycle, when maximum frequency is targeted.

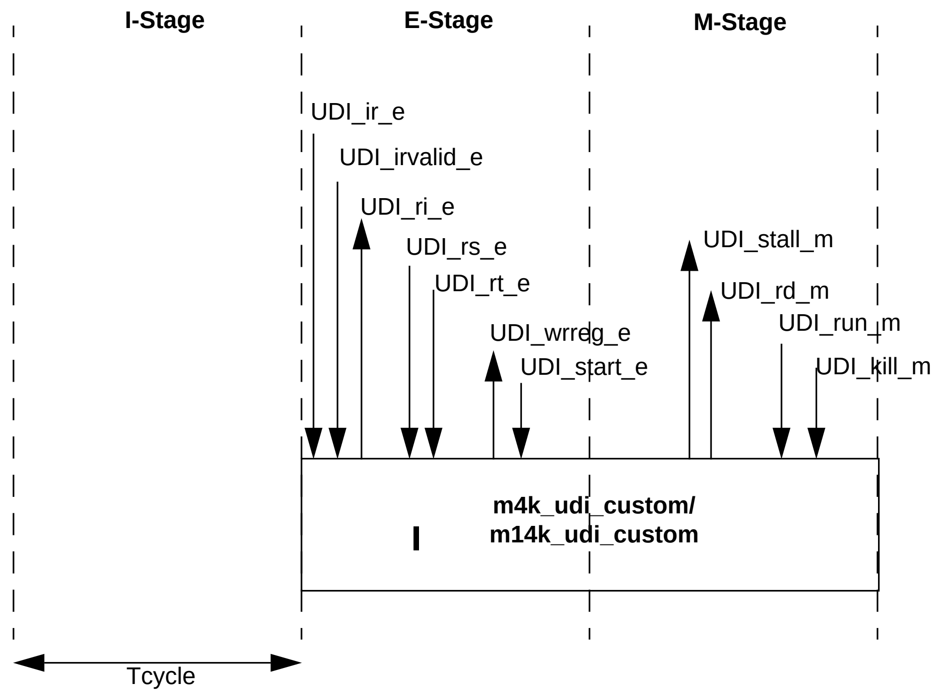

Figure 3 Relative Timing of Interface Signals with Respect to Pipe Stages

4 CorExtend® UDI Pipeline Interaction

Figure 4 depicts how the CorExtend UDI block logically interacts with the rest of the core's internal pipeline.

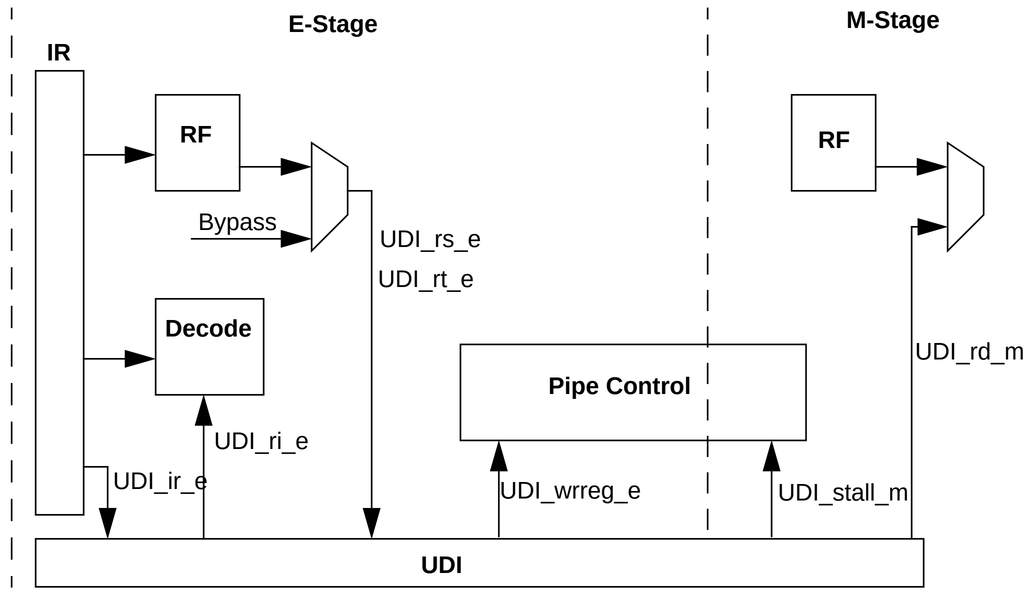

Figure 4 Interaction of the CorExtend® UDI Module with the Pipeline

The interaction of user-defined instructions with the core pipeline is handled implicitly by the pipeline control logic. Additional information about the state of the core pipeline is required for instructions with a destination of a register internal to the UDI block.

4.1 Source Operands

The full register values specified by the rs and rt operands, as encoded in the instruction word, are always supplied to the UDI module during the E stage, regardless of whether a particular UDI format actually uses those registers as source operands. These rs and rt values participate in dependency checking versus the destination register of the previous instruction. A pipeline interlock may occur if the previous instruction's destination matches the rs and/or rt register of the UDI, and that destination register is not available due to a stall in the pipeline.

For simplicity, the pipeline control logic always assumes that rs and rt source operands are encoded in the instruction word in the "normal" location, because this is likely to be the most prevalent case. If a UDI format does not actually use these source operands, then some unintended stalling may occur based on the contents of these bits in the instruction word if false dependencies are detected, but functionally this will not present a problem. The UDI module will need to handle instructions requiring source operands from internal UDI registers. Even in this case, the rs and rt operands are supplied by the core to the UDI module.

The source operands may be provided directly from the register file, or from bypass sources, depending on the execution of older instructions within the pipeline.

4.2 Destination Register

The UDI module is required to send the destination register back to the pipeline control logic during the E stage, even if the destination is not to a general-purpose register. Since the user is free to encode the destination register anywhere within the available bits of the UDI, this is the only way that the pipeline control logic can be informed of the destination register. If the destination register is not a general-purpose register, but an internal UDI register, the R0 register should be sent back to the core as the "destination register".

4.2.1 General-Purpose Register as Destination

The destination register is used by the pipeline control logic to determine if the UDI's destination is also a source operand for later instructions. If necessary, the UDI result will be bypassed to a following dependent instruction, instead of coming directly from the register file.

Normally, the pipeline control logic assumes that a UDI has single-cycle latency, implying that the result will be provided one clock after the instruction was presented to the UDI module. Multi-cycle operations are supported via assertion of the *UDI_stall_m* signal by the UDI module, and is used when the destination is to a general-purpose register, but the results are not available yet. When asserted, *UDI_stall_m* indicates that the UDI result on the *UDI_rd_m* bus is not yet valid, and will cause the M stage and all earlier stages to be frozen until the result is valid. Later pipeline stages can continue to operate. Technically, this pipeline situation is referred to as a *slip*, but in this document the terms *stall* and *slip* are generally used interchangeably.

The pipeline support allows UDIs with single-cycle latency to issue and complete in back-to-back cycles with no pipeline slip penalties, even if the UDI destination is supplied to a second consecutive UDI as a source operand.

4.2.2 Internal UDI Register as Destination

The UDI module can also progress largely independent of the core pipeline, if the instruction's destination is to an internal UDI register. In this case, the core pipeline sees the UDI module as having single-cycle latency. This is accomplished by sending R0 to the core as the destination register. The core can then progress, and not wait on the UDI instruction completing.

Special consideration needs to be taken when writing the final results to the internal UDI register. The UDI module needs to avoid writing data before the instruction is guaranteed to complete. When the instruction is in the "M stage" of the core pipeline, it can still be killed and restarted due to an exception. This can be successfully handled by watching two specific core signals: *UDI_run_m* and *UDI_kill_m*. When the UDI instruction is in the equivalent of the "M stage" in the core pipeline, and *UDI_run_m* is asserted (with *UDI_kill_m* being de-asserted), the instruction can no longer be killed, and the UDI can freely write its results to the internal register (whenever the data is available).

4.3 Local UDI State and Context Switches

If the UDI block has internal state, that state may need to be saved and restored on a context switch. If there is a significant amount of state and not all processes use UDI, it may be advantageous to only save the UDI state for processes that use UDI. The software implications for handling context switches are beyond the scope of this document, but this section describes a few of the hardware mechanisms on the CorExtend interface that can be of help.

4.3.1 Local UDI Enable

One way to handle the saving of UDI state is to create a UDI enable within the UDI block that can only be written by the kernel. The *UDI_kd_mode_e* signal is available to indicate that the E-stage instruction is executing in kernel or debug mode. On a new process, the UDI block is disabled; if the user code attempts to execute a UDI instruction, an

RI exception will be taken. The kernel can then enable the UDI block, mark this process as using UDI, and save/restore the UDI state on context switches.

Note that this method incurs a fairly significant overhead, and in many cases, it may be more efficient to always save and restore the UDI state.

4.3.2 CorExtend Enable Bit in Status Register

Alternatively, the *CEE* (CorExtend Enable) bit in the *Status* register can be used to signal the operating system that the current process is using CorExtend operations, so the local state can be tracked as necessary. The *Status.CEE* bit can be written by privileged software. Whether *CEE* has any effect on the core is determined by the *UDI_honor_cee* signal on the CorExtend interface. If the CorExtend block deasserts this signal, the value of the *CEE* bit has no effect on the core.

If local state exists and *UDI_honor_cee* is asserted, the intended usage is as follows:

1. Software initializes the *CEE* bit to 0, to indicate that CorExtend operations are not enabled.

1. The first time a UDI operation is attempted, the core will take a CorExtend Unusable exception. This dedicated exception, instead of a general Reserved Instruction exception, can be used to inform the operating system on a per-process basis that UDI operations containing local state are desired.

2. The OS kernel does whatever it needs to prepare for saving/restoring CorExtend state, and then sets the *CEE* bit to 1.

3. Upon return from the handler, the CorExtend instruction can now be executed.

4. Repeat for other processes.

5 CorExtend® UDI Timing Diagrams

The following figures describe the pipeline timing for several typical CorExtend UDI operations:

5.1 UDI with Single-Cycle Latency

Figure 5 on page 18 shows signal timing for a single-cycle user-defined instruction. In cycle 2, the user-defined instruction word reaches the E stage and is presented to the UDI module. If this was an unsupported UDI operation, then the UDI module would have asserted *UDI_ri_e* in that same cycle. The UDI module returns the result data in the following cycle. The *UDI_run_m* input signal can be ignored, and is not included in the diagram.

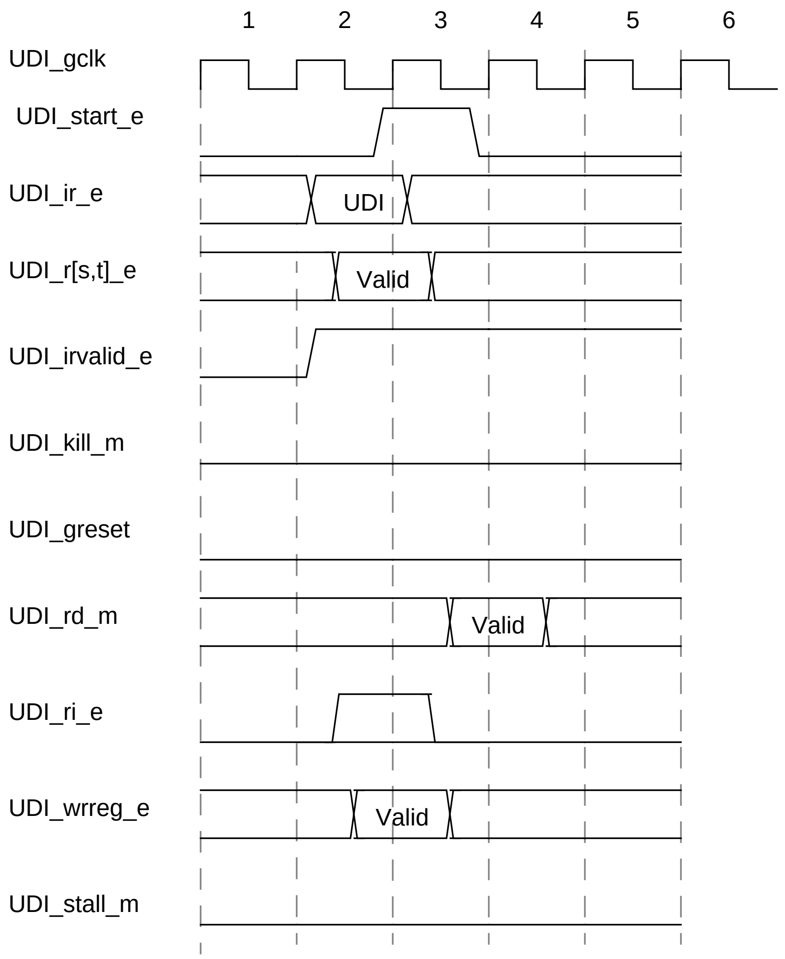

Figure 5 Single-Cycle UDI Operation

5.2 Back-to-Back UDIs with Single-Cycle Latency

Figure 6 on page 19 shows timing for two back-to-back user-defined instructions, UDI0 and UDI1, both with single-cycle latency. Multiple UDIs can be issued and completed in consecutive cycles with no pipeline stalls. The *UDI_run_m* input signal can be ignored, and is not included in the diagram.

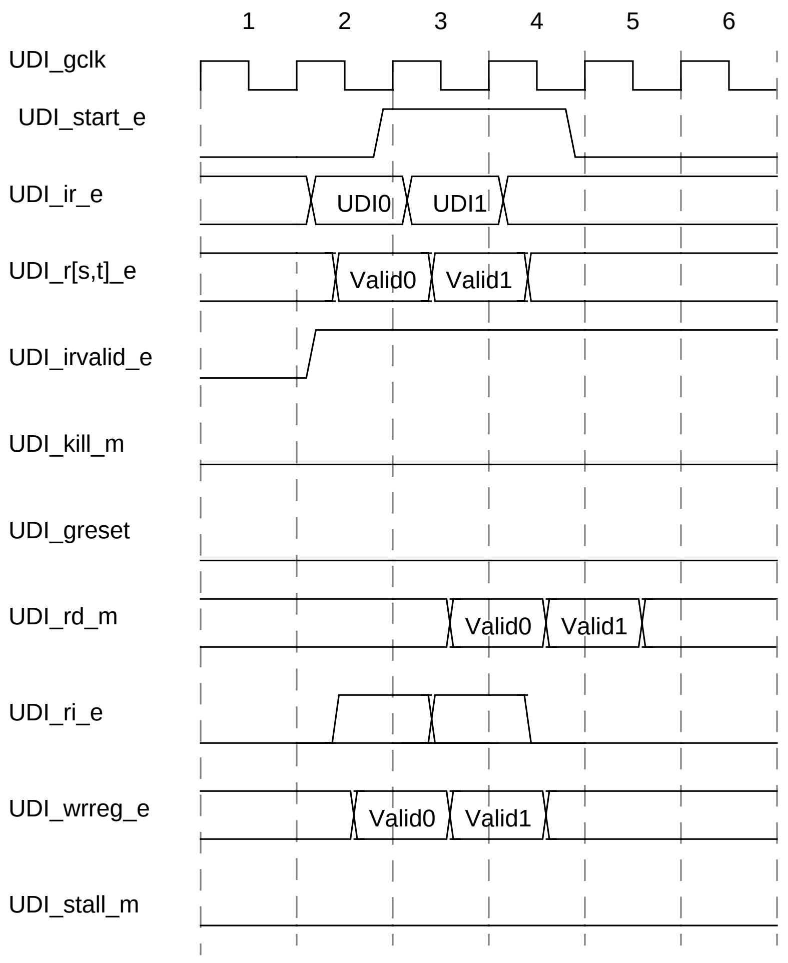

Figure 6 Back-to-Back Single-Cycle UDI Operation

5.3 UDI with Multi-Cycle Latency

A user-defined instruction with 4-cycle latency is shown in Figure 7 on page 20. This is similar to the single-cycle case shown in Figure 5, but now in cycle 3, the UDI module signals *UDI_stall_m*, indicating that the result data is not yet ready. This continues for three additional cycles. Finally in cycle 7, the result data is available, so *UDI_stall_m* is deasserted and the result data is driven on *UDI_rd_m. The UDI_run_m input signal can be ignored, and is not included in the diagram.*

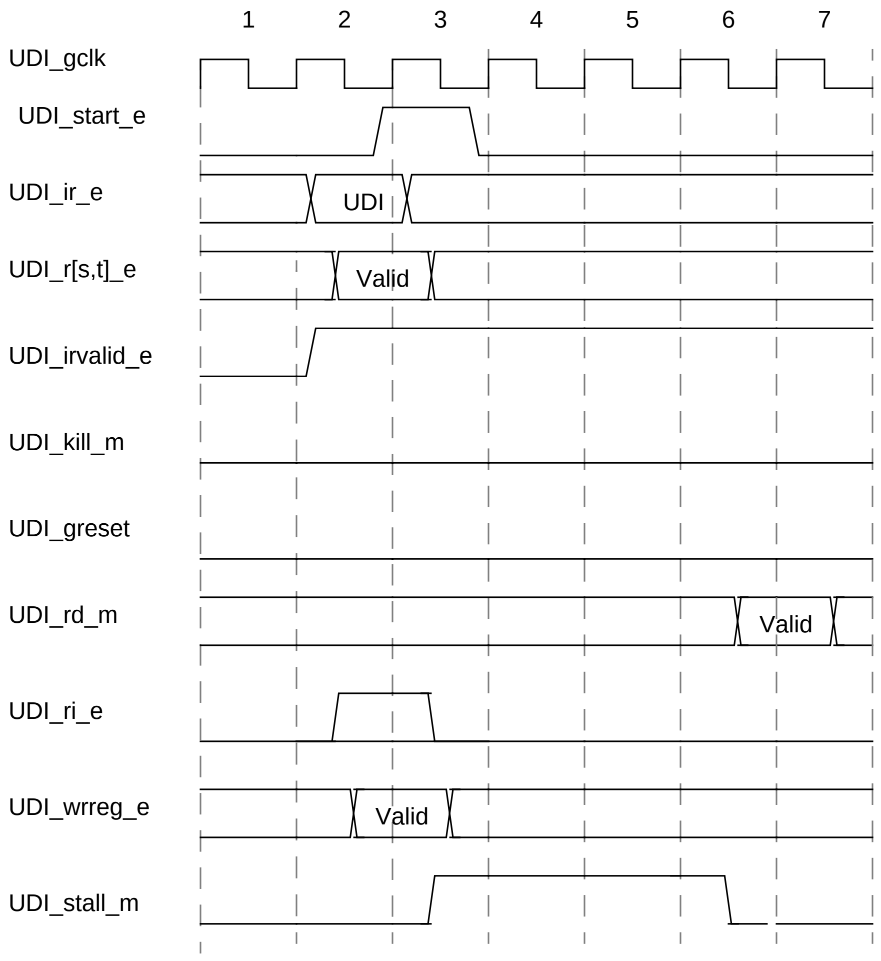

Figure 7 Multi-Cycle UDI Operation

5.4 Multi-Cycle UDI Killed by Earlier Exception

Figure 8 shows the signal timing for a user-defined instruction that is killed due to an exception on an earlier instruction in the pipeline. In this case, *UDI_kill_m* is asserted in cycle 4. The UDI module responds to this in the following cycle, by deasserting the *UDI_stall_m* signal. The *UDI_run_m* input signal can be ignored, and is not included in the diagram.

The *UDI_kill_m* signal is provided to the UDI module to allow faster processing of exceptions when a multi-cycle UDI operation will be aborted. The core control logic automatically takes care of blocking the register file write of a UDI destination when the UDI will be aborted due to an exception on an older instruction in the pipeline, but the core pipeline is stalled and the exception is processed until the *UDI_stall_m* signal deasserts; thus, exception latency can be improved if the UDI module uses the *UDI_kill_m* signal to deassert *UDI_stall_m* on long latency UDI operations, though the results of the UDI operation will be aborted anyway. Since *UDI_kill_m* is available late in the cycle, it is recommended that it be used sequentially to clear *UDI_stall_m* only in the next cycle.

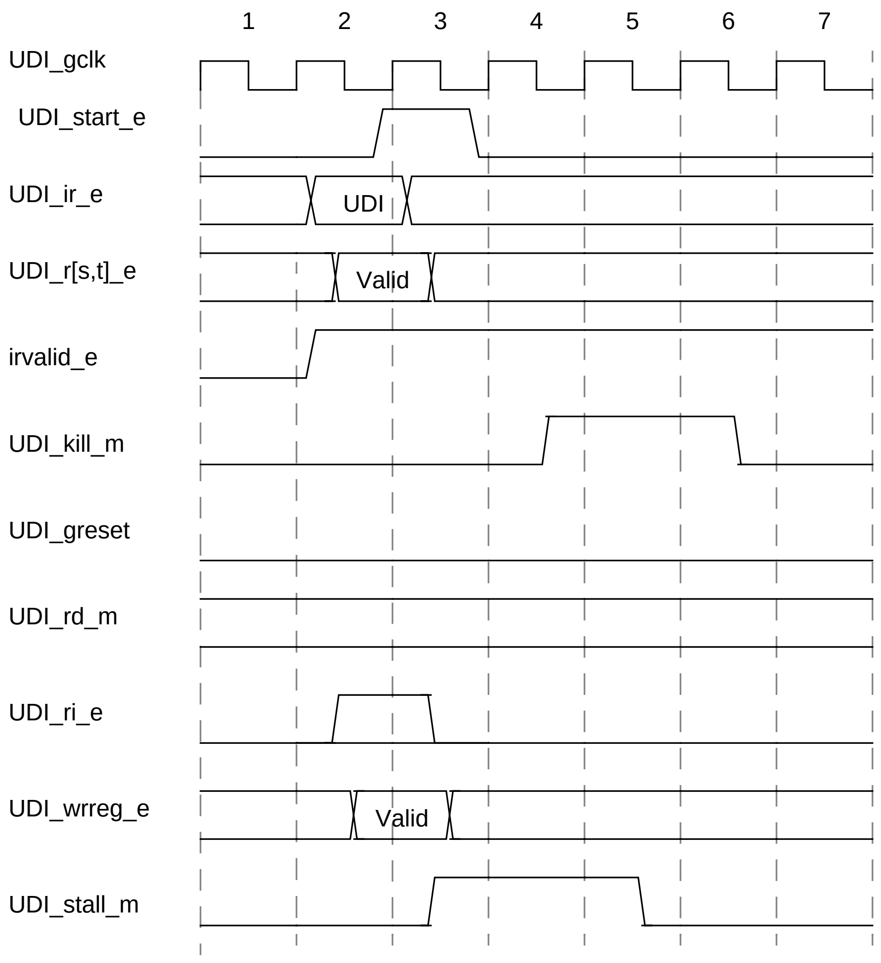

Figure 8 Killed UDI Operation

5.5 Pipelined UDI Using Internal Register as Destination

Figure 9 shows the normal operation of a pipelined UDI instruction whose destination is an internal UDI register. Notice the *UDI_run_m* signal is generally asserted the cycle after the *UDI_start_e* signal. This might not be the case if the pipeline is stalled for another reason.

After *UDI_run_m* is seen asserted (without *UDI_kill_m* asserted) the instruction can no longer be killed and restarted by the pipeline, and it is safe to commit its results to the internal register. As can be seen in the figure, because the instruction destination is an internal register, R0 is sent to the core on *UDI_wrreg_e*. Then, in the “M-stage”, any data can be on *UDI_rd_m*. Also, notice that *UDI_stall_m* does not need to be asserted for the pipelined UDI instruction.

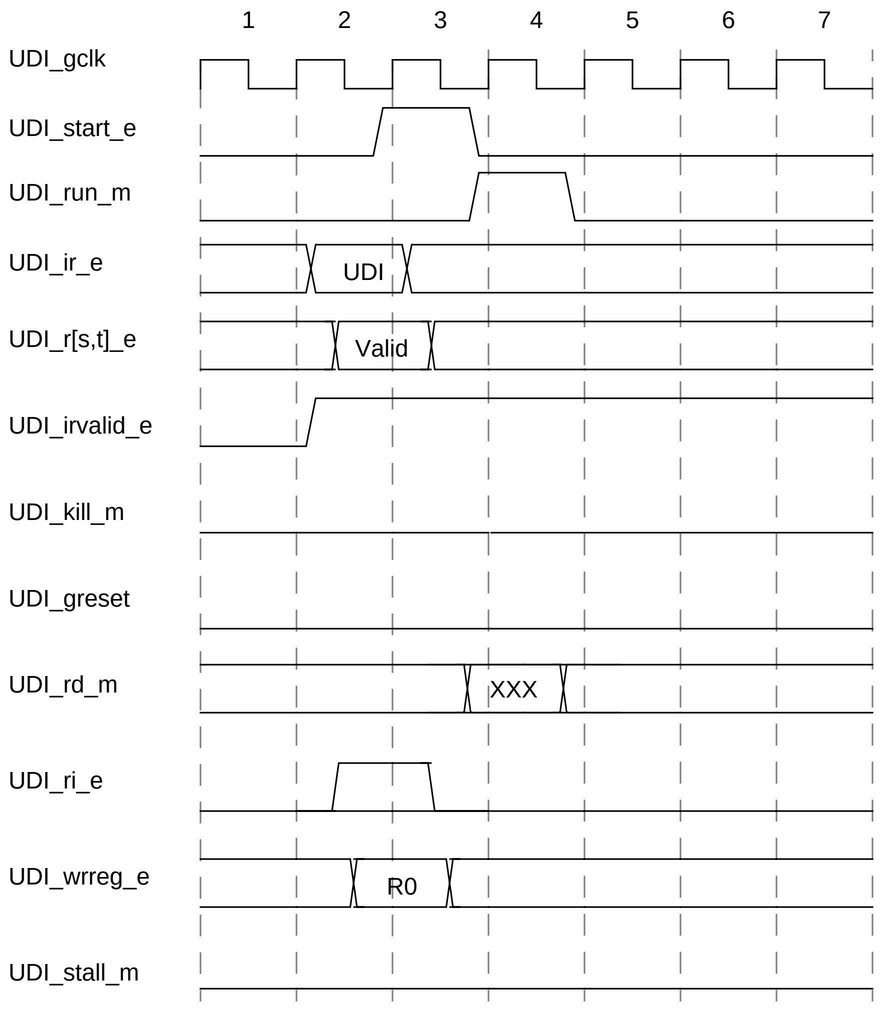

Figure 9 Pipelined UDI with Internal Result Registers

5.6 Pipelined UDI Killed by Exception Before Being Committed

Figure 10 shows a pipelined UDI that receives a delayed *UDI_run_m*. When *UDI_run_m* is finally asserted, *UDI_kill_m* is also asserted. The UDI results should not be committed to the internal register because the instruction could be restarted.

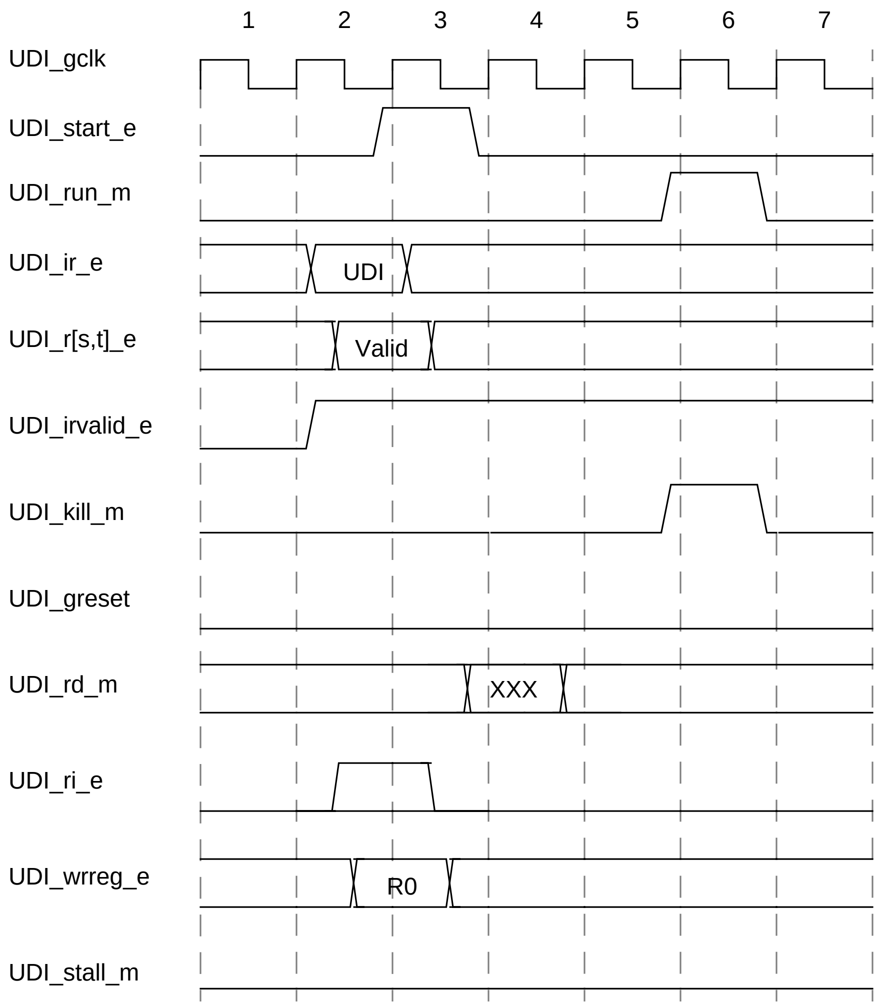

Figure 10 Pipelined UDI killed by exception

6 CorExtend® UDI Sample Implementations

Several sample Verilog modules are included in the RTL release, showing example implementations of CorExtend™ UDI. The Verilog files are located in the directory `$MIPS_PROJECT/proc/design/rtl` and `$MIPS_HOME/$MIPS_CORE/proc/design/rtl` of a core release.

6.1 One/Zero Count

Several sample modules implement the same two instructions, but with varying latency. The instructions either count the number of zeroes (if *ir_e[5:0]* == 6'b010000) or ones (if *ir_e[5:0]* == 6'b010001) on the word provided via the *rs_e* source operand and return the count as the result, with a general-purpose register as the destination. The instruction format incorporated in these examples is described in "Three general-purpose register operands plus 5 immediate bits", on page 8, although the *rt_e* source operand is unused.

The one/zero count modules are named:

- `m4k_udi/m14k_udi`: Implements one/zero count in a single cycle.
- `m4k_udi_2cycle/m14k_udi_2cycle`: Implements one/zero count in two cycles.
- `m4k_udi_multicycle/m14k_udi_multicycle`: Implements one/zero count in three cycles. Shows the use of the *kill_m* input to deassert *stall_m*.

6.2 Pipelined Bit Swap with Local UDI State

Another example implements a pipelined UDI module with internal result storage. It does basic bit swapping, and includes six different instructions:

1. *Move from Hi*: Move data from internal UDI register "hi" to General-Purpose Register.
2. *Move from Lo*: Move data from internal UDI register "lo" to General-Purpose Register.
3. *Move to Hi/Lo*: Move General Purpose-Register data to internal UDI registers.
4. *Swap*: Use General-Purpose Register data as source operands, Execute a bit swap, Store results in internal UDI registers.
5. *Swap Accumulate*: Use General-Purpose Register data *and* UDI internal registers as source operands, execute a bit swap, store results in UDI internal registers.
6. *Swap Accumulate GPR*: Use General-Purpose Register data *and* UDI internal registers as source operands, execute a bit swap, store results in a General-Purpose Register.

Note: The above hi/lo registers do *not* refer to the MDU hi/lo register pair. These are simply internal UDI registers.

This example module is:

- `m4k_udi_pipe/m14k_udi_pipe`: Implements simple bit shifting in a pipeline with internal HI/LO registers to store results for use by other UDI instructions.

7 Verifying CorExtend® Instructions

The ability to add instructions to a processor core provides a great deal of flexibility to the CorExtend implementor, but also poses some functional verification challenges. Since the function of CorExtend instructions is completely defined by the implementor, the verification of added instructions is generally beyond the scope of deliverables provided by MIPS. MIPS recommends that this verification usually be handled in the context of how the core and accompanying CorExtend block are instantiated in the implementor's full SOC. However, some additional verification deliverables are provided with a processor core that a CorExtend implementor may find helpful.

7.1 AVP Environment

Internally, MIPS uses a suite of Architectural Verification Programs (AVPs) to test the compatibility of various implementations of the MIPS Architecture. A subset of this AVP setup, consisting of the kernel environment and sample diagnostics used to test the CorExtend reference designs, is included with a processor core delivery in source form. This environment can be extended and used for creating assembly-level tests that exercise the CorExtend implementor's actual instructions in conjunction with the core pipeline.

7.1.1 AVP Documents

The *MIPS® Architecture Verification Programs Release Notes* [15] provides an overview of the AVP environment and installation instructions. The *MIPS® Architecture Verification Programs User's Manual* [16] provides details about the use of the AVP environment. MIPS recommends that a user review these documents to understand the AVP infrastructure. These documents are included in the `$MIPS_HOME/$MIPS_CORE/doc` area of a processor core release.

7.1.2 AVP Installation

AVP-related tar files are located in the `$MIPS_PROJECT/CorExtend_AVP` area of a processor core release. This directory contains two tar files, one for the general AVP environment and another for the core-specific CorExtend example diags. These tar files should be installed as described in the *AVP Release Notes* [15]. Note that the general AVP tar file must be installed first.

7.1.3 Setup to Run CorExtend AVPs on Soft Core Testbench

After following the instructions for AVP installation, the CorExtend AVPs may be run using the soft core simulation environment by following a few basic steps:

1. From the `$MIPS_PROJECT/proc/verification` directory, generate a `DiagInfo.CorExtend` file by typing:

   ```
   % perl $MIPS_HOME/$MIPS_CORE/bin/buildCorExtendDiagInfo
   ```

2. In that same directory, edit the `DiagInfo.user` file. In that file is an example *include* statement. Un-comment that statement and using the example as a guide, replace <absolute path to DiagInfo file> with the absolute path to the generated `DiagInfo.CorExtend` file.

In addition to diagnostic attributes needed for running individual AVPs, a diagnostic group of CorExtend AVPs in all modes, as specified by the AVP release, is also generated. This group, *Pro_CorExtend*, may be used for regression purposes. General details about using the soft core simulation environment can be found in the appropriate *Implementor's Guide* document ([8], [9], [10]), [11], [12].

7.2 Reference AVPs for Sample Modules

Once the AVP environment has been installed, the directory `$MIPSARCHROOT/Diag/AVP/CorExtend/MIPS32-4KE` holds several subdirectories containing diagnostics that exercise the example CorExtend modules discussed in Section 6, "CorExtend® UDI Sample Implementations".

Generally, one diag corresponds to the example RTL module of the same name.

7.2.1 AVP Suggestions

A detailed description for writing suitable assembly verification tests is beyond the scope of this document. However, some suggestions and hints to keep in mind when verifying CorExtend instructions are presented, especially as they relate to some interesting cases illustrated in the example diags.

Self-checking Diags

MIPS generally suggests making user-written CorExtend diags self-checking to the extent possible. This means that the diag should “know” the result to expect for a UDI operation and ensure that any state update occurs appropriately.

GPR Dependencies

Be sure to exercise dependencies between adjacent CorExtend instructions, in which one CorExtend instruction produces a GPR result that is used as a source operand by the next CorExtend instruction. The internal core pipeline logic should take care of handling these dependencies, but it is still a recommended sequence to test.

Destination of GPR R0

When a CorExtend instruction targets R0 as a destination, make sure that *UDI_wrreg_e* is not asserted. This case should be explicitly tested for GPR-targeted CorExtend instructions.

Local State

If the CorExtend block writes its results to locally held storage, verify that the state is properly updated in the presence of pipeline stalls and kills.

Changing Kernel Mode and Endianness

If the CorExtend block is making use of the *UDI_kd_mode_e* and/or *UDI_endianb_e* signals, note that there are hazards around the changing of these signals. Refer to the appropriate *Software User’s Manual* [References 1, 2, 3] for more details about hazards. Hazards can generally be eliminated by interposing one of the hazard barrier instructions between the cause of the hazard and its consumer.

8 References

This appendix lists other documents available from MIPS Technologies, Inc. that are referenced elsewhere in this document. These documents may be included in the `$MIPS_PROJECT/doc` area of a typical soft or hard core release, or in some cases may be available on the MIPS web site, under **http://www.mips.com/publications/index.html**.

1. MIPS32® 4KE™ Processor Core Family Software User's Manual
   MIPS document: MD00103

2. MIPS32® 4KSd™ Processor Core Software User's Manual
   MIPS document: MD00319

3. MIPS32® M4K™ Processor Core Software User's Manual
   MIPS document: MD00249

4. MIPS32® M14K™ Processor Core Software User's Manual
   MIPS document: MD00668

5. MIPS32® M14Kc™ Processor Core Software User's Manual
   MIPS document: MD00674

6. MIPS32® M14KE™ Processor Core Software User's Manual
   MIPS document: MD00813

7. MIPS32® M14KEc™ Processor Core Software User's Manual
   MIPS document: MD00821

8. MIPS32® 4KE™ Processor Core Family Implementor's Guide
   MIPS document: MD00114

9. MIPS32® 4KSd™ Processor Core Implementor's Guide
   MIPS document: MD00321

10. MIPS32® M4K™ Processor Core Implementor's Guide
    MIPS document: MD00250

11. MIPS32® M14K™ Processor Core Implementor's Guide
    MIPS document: MD00667

12. MIPS32® M14Kc™ Processor Core Implementor's Guide
    MIPS document: MD00673

13. MIPS32® M14KE™ Processor Core Implementor's Guide
    MIPS document: MD00811

14. MIPS32® M14KEc™ Processor Core Implementor's Guide
    MIPS document: MD00819

15. MIPS® Architecture Verification Programs Release Notes
    MIPS document: MD00120

16. MIPS® Architecture Verification Programs User's Manual
MIPS document: MD00121

9 Document Revision History

Revision	Date	Description
01.00	November 12, 2002	Initial release.
01.01	March 5, 2003	Added note about choosing destination registers that will work with future cores.
01.02	November 3, 2004	• Major updates to reflect hierarchy and signal name prefix changes due to externalization of CorExtend interface. • Changed title from "Implementor's Guide" to "Integrator's Guide". • Added CorExtend Enable capability and CorExtend Unusable exception. • Clarified types of exceptions that are possible with CorExtend instructions. • Added new section on functional verification. • Used ® symbol for registered MIPS32® references. Made trademark usage more consistent.
01.03	August 29, 2008	• Updated document template
01.04	March 14, 2010	• Added M14K and M14Kc cores to this document • Generalized the term "Pro Series" to "Processor Core" to cover M14K & M14Kc cores. • Added microMIPS user-defined instruction format description (1.2) which is supported by M14K & M14Kc. • Changed title from "MIPS32® ProSeries® CorExtend® Instruction Integrator's Guide" to "CorExtend® Instruction Integrator's Guide for M4K/4KE/4KS/M14K/M14Kc Cores".
01.05	June 12. 2011	• Added M14KE and M14KEc cores to this document • Changed title from "CorExtend® Instruction Integrator's Guide for M4K/4KE/4KS/M14K/M14Kc Cores" to "CorExtend® Instruction Integrator's Guide for M4K/4KE/4KS and M14K Family Cores".

Template: nW1.03, Built with tags: 1D