LEG Processor for Education

System Documentation

Contact:
Max Waugaman (mwaugaman@hmc.edu)
Sam Dietrich (sdietrich@hmc.edu)

Last Modified May 10, 2016

Contents

1 Getting Started

1.1 Installation

LEG processor development begins by downloading the code base to your local machine or a shared class drive. The code is small enough for each user to have a local copy for development and testing. After cloning the LEG repository, only a few off the shelf programs are required to get the processor running. Complete instructions are presented here for Linux systems, but most functionality is available on Windows.

Several programs must be installed to use the full LEG debugger. However, only the LEG repository must be downloaded if you are simply interested in examining the source files.

1. Install git and clone the LEG git repository from https://github.com/MWaug/LEG. The next installation steps only need to be one once per machine you are installing the LEG debugging system on.

2. Install packages required for the debugging tools. We needed to install the packages listed below to run the debugging framework on our 64-bit system, but your requirements may vary. They can be installed using your package manager.

Package	**Needed for**
g++	QEMU, GDB, debug_lockstep
libtool	QEMU
zlib1g-dev	QEMU
libglib2.0-dev	QEMU
libpixman-1-dev	QEMU
libfdt-dev	QEMU
libpython2.7:i386	GDB
lib32ncurses5	GDB
libxft2:i386	ModelSim
libxext6:i386	ModelSim
python (2.7)	debug_lockstep

3. Choose a path on your system to install the required debugging tools. Be sure that this path is readable by everyone who will be using LEG on your system. Throughout the examples in this guide, the path will be `/legproj/`

4. Install the free Starter Edition of ModelSim from http://dl.altera.com or any version of ModelSim you may already have purchased. Note that in Starter Edition performance may be reduced and some minor features such as checkpoints will be unavailable. LEG has been tested on ModelSim version SE 10.3d.

5. Install GDB and GCC for bare metal ARM systems from https://launchpad.net/gcc-arm-embedded and untar into your LEG installation directory. For example, `/legproj/gcc-arm-none-eabi-5_3-2016q1`.
Instructions for this are provided in the readme available at the same website.

6. The QEMU system emulator must be downloaded and installed from source. **NOTE**: The patches required for LEG will possibly break every QEMU system except `qemu-system-arm`. To keep this guide simple the default QEMU build process is used. You can safely remove systems other than `arm-softmmu`.

   (a) Clone the QEMU source into your central LEG directory:
   `git clone git://git.qemu-project.org/qemu.git`

   (b) From this QEMU directory, run the following commands:
   `git checkout -b leg-additions v2.4.0`
   `git am leg/repo/path/qemu_patches/*.patch`
   where `leg/repo/path` is the path to your LEG git repository, not the folder for debugging tools. These commands will apply the custom LEG addons to v2.4.0 of QEMU.

   (c) In the QEMU directory run
   `./configure`
   `sudo make -j20`
   to build QEMU.

7. The Linux executable for LEG debugging can be found in `leg/repo/path/kernel`. LEG uses BusyBox 1.23.1, which can be compiled from source using the instructions in `kernel-setup-info`. Unless you wish to rebuild Linux, the only action necessary is to untar the file `system.tar.gz`:
   In `kernel/`, run `tar -zxvf system.tar.gz`

Now all software necessary to run the LEG testing tools should be installed, but some components must still be configured. These short steps are described in the next section.

1.2 Configuration

Certain variables must be set to provide LEG access to the programs it needs for testing. These steps will get you there.

1. Add the ModelSim tools (vsim) to your system path. These are most likely found in the `modelsim_ase/linuxaloem` directory of your ModelSim installation. For example, using the path name and ModelSim version referenced in the previous section, we would edit our `.bashrc` file and add the following line:
   `export PATH="/legproj/altera/15.1/modelsim_ase/linuxaloem:$PATH"`

2. Navigate to `leg/repo/path/debug_lockstep` and modify the file `configuration.py` with the following. Examples are included in the file.

   (a) set `qemu_path` to the path to `qemu_system_arm`

   (b) set `gdb_path` to the path to `arm-none-eabi-gdb-py`

   (c) set `hasVopt` to 0 if you are using ModelSim SE, or 1 if you have a version of ModelSim with vopt.

(d) set `linux_path` to the path of the Linux executable. This should already be set unless you compiled your own version.

3. For waveform debugging it is useful to have another ModelSim project that can quickly load saved processor state from a certain instruction. See Section 3.2 for information on debugging in ModelSim using this setup.

   (a) Create a folder `MSdebug` in your LEG repository path and create a new ModelSim project named `msdebug` in it. If you use different names you will need to edit `qemuDumpRestore_MS.tcl` after the final step.

   (b) Open the project and add all LEG SystemVerilog files, located in `leg_pipelined`.

   (c) Copy `qemuDumpRestore_MS.tcl` and `addAll.do` from `debug_lockstep` into the MSdebug folder.

2 Basic Testing

This section describes the basic procedure for generating and running tests. It covers running a provided test, useful GDB commands, and compiling handwritten tests.

2.1 Relevant Files

- **debug_lockstep/debug.sh** The entry point for the debugging framework.
- **debug_lockstep/unitTests/*** Randomized tests for every instruction. Each test is named with the instruction or instruction type followed by the number of instructions in the test. Each test has four or five files
  - `test.s`, the randomly generated assembly.
  - `test.dat`, the corresponding hexadecimal machine code.
  - `test.dump`, the disassembled machine code.
  - `test.elf` (optional), which contains information about how interrupts are raised in exception handler tests.
  - `test.bin`, the binary file that runs on the processor and QEMU.
- **debug_lockstep/customTests/compileTest.sh** Script for compiling tests for LEG from assembly source. See Section 2.3.2.
- **debug_lockstep/customTests/makeRandomAssembly.py** Script to generate random assembly instructions to test on LEG. See Section 2.3.1.

command	effect
`stepi`	Execute one instruction
`i r`	Print register contents of the current mode, including cpsr
`x/10i $pc`	Print the next 10 instructions
`x/10w $sp-20`	Print 10 words of memory, starting 20 addresses below the stack pointer
`x/5b 0xfffdc`	Print 5 bytes of memory, starting at `0xfffdc`
`b *0x220`	Set a breakpoint at address `0x220`

Table 1: Examples of common GDB commands

2.2 Basic Operation on Provided Tests

This section demonstrates the most basic operation of the testing framework. It will allow you to run a test from the terminal and debug in gdb at a register and instruction level.

1. From your local copy of the LEG repository, enter the `debug_lockstep` directory.

2. Provided tests are stored in the `unitTests` directory. To run a provided test, type

   ```
   ./debug.sh -t unitTests/testname.bin
   ```

   at the terminal. For example, to run the test consisting of 1000 random `ADC` instructions, enter

   ```
   ./debug.sh -t unitTests/adc_1000.bin
   ```

3. LEG will now be compiled by ModelSim and instances of QEMU and GDB will be started. If there are no compilation errors, you will be at the prompt of gdb modified with LEG-specific debugging commands. Many of these are explained further in Section 3.5.2

   The most basic command is `leg-lockstep`, which runs LEG until a bug is found. Enter this now.

4. The test will print a '`.`' after each successfully executed instruction. If the program completes successfully, a `PASSED` message will appear. Otherwise debugging information will be printed, including the last correct register state, the current incorrect LEG register state, and the expected register state.

5. Debug with any GDB command, or type `leg-stop` to exit.

This guide cannot present a complete tutorial on using GDB, but a few examples of the most useful commands are listed in Table 1 below.

2.3 Creating Tests

Custom tests can be created from Assembly source and run on LEG using the tools in `debug_lockstep/customTests`.

2.3.1 Generating Assembly

The script `makeRandomAssembly.py` can be used to easily create random instructions to test LEG with a vast combination of cases. Because interrupt handlers, a small stack, and all processor modes are automatically initialized, the output of the script can also be extended with your own additional test vectors. For example, to create 1234 random add and load instructions and output the program to `add_load.s`, run

```
python makeRandomAssembly.py -i add ldr -n 1234 > add_load.s
```

Run the script with `-h` to print full usage info.

NOTE: Sometimes immediates produced by the script are not recognized by the compiler. In these cases you must re-run `makeRandomAssembly.py` or manually change the immediate value that fails compilation.

2.3.2 Compiling Assembly

Any assembly code compatible with ARMv5 excluding Thumb can tested on LEG. Compiling the code uses the ARM bare metal development toolchain that was previously installed in Section 1.1. Simply run `compileTest.sh` in `debug_lockstep/customTests` with the basename of the assembly code you want to compile. Note that the `.s` extension is not included.

```
./compileTest.sh add_load
```

The compiled files will be placed in the `tests` directory. The test can now be run on LEG from `debug_lockstep`:

```
./debug.sh -t customTests/tests/add_load.bin
```

2.4 Debugging Tools

3 Advanced Testing and Simulator Operation

This section describes tools for more in-depth debugging than is possible using GDB alone. It also provides more detailed information about the GDB extensions and debugger output files.

3.1 Relevant Files

- **checkpoint.py** Python file that handles creation of checkpoints.
- **checkpoint.tcl** Simple tcl file to create a checkpoint.
- **debug.py** Python file that is executed within GDB and sets up the debugging commands.
- **debug.sh** Script to start the debugging process.
- **debug.tcl** Tcl script that handles lockstep debugging on the ModelSim end.
- **lockstep.py** Python file that handles lockstep debugging on the GDB end.
- **qemuDump.py** Python file that handles dumping Qemu's current state to file.
- **qemuDumpRestore.tcl** Tcl script that manipulates the ModelSim state to match a dumped Qemu state.
- **addAll.do** Adds almost all relevant waveforms for debugging LEG in ModelSim.
- **divide_controller.py** Allows parallel debugging of large executables.

3.2 Waveform Debugging

Finding the root cause of a bug often requires following the values of internal processor signals over time. The procedure described here explains the process of loading processor state into ModelSim for signal level debugging.

1. Processor state can be dumped from the GDB prompt at any point while debugging. The first step is to determine the instruction at which the bug occurs. Running `leg-lockstep` will simulate until a bug is reached and then report the location of the bug, for example `0x258`. Restart the simulation using `leg-restart` and then jump to a location before the bug, for example `leg-jump *0x254`.

2. Now the simulation has been advanced to just before the bug. Dump qemu's state to a form that can be reloaded into ModelSim. Enter `leg-qemu-full-dump` at the gdb prompt.

3. Copy the dump files `qemu_mem_dump.dat` and `qemu_state_dump` to the ModelSim debugging directory `MSdebug` set up in Section 1.2. The dump files can be found in `debug_lockstep`.

4. Open the ModelSim project created in Section 1.2 and ensure all source files compiled and up to date. Then enter

   ```
   source qemuDumpRestore_MS.tcl
   ```

   in the ModelSim command prompt. This will load the state from the dump files and add useful signals to the wave window.

5. type `run xxx` to simulate the processor, where `xxx` is the time to simulate. A value of 200 usually good to start.

6. Watch the program counter, register file, and other signals for the reported bug to appear. Many signal names can be found in the processor description below, and the rest can be found in the source code. Happy debugging!

3.3 Interrupt Testing

Interrupts are enabled by default and tests with interrupts can be generated using `makeRandomAssembly.py` with a nonzero `interrupt_ratio` (see Section 2.3.1). Interrupts are also present in Linux.

Interrupts can be disabled in any test by passing the argument `--noirq` when running `leg-lockstep` or `divide_controller.py`.

3.4 Linux Testing

Since Linux is a large executable, lockstep testing the entire startup process is difficult. Linux can be run from any point by running `debug.sh` with no arguments and then jumping to the desired location or function, for example `leg-jump start_kernel`.

An alternative option that allows parallel simulation of the entire boot process is `divide_controller.py`. With an input argument of `parallel-divisions.txt`, worker instances of `debug.sh` are started for each instruction range in that file. An easy to use monitoring interface is provided. This works best on a system with many cores, and `parallel-divisions.txt` can be customized to the number of parallel instances desired.

`divide_controller.py` can also be used with your own executables and custom `parallel-divisions.txt` files to debug any large program.

3.5 Simulator Operation

This section provides more detailed information on the operation of the LEG debugging framework.

3.5.1 Output File structure

The debugging scripts place all of their output in the `debug_lockstep/output` directory. For each run of a script, a new subdirectory within this directory is created, named according to the appropriate timestamp. Within this directory, the `bugs` directory contains the full debug output of each found bug, and `runlog` contains an abbreviated summary of all bugs found in the given run. Duplicate bugs are ignored and do not appear in these files.

Created checkpoints appear in `output/checkpoints/` with the provided name.

3.5.2 LEG GDB extensions

When you are at the GDB prompt, you can run arbitrary GDB commands, but additional commands are enabled:

- `leg-lockstep`: Starts lockstepping at the current instruction. This dumps the current qemu state to a file, and then starts ModelSim initialized with the current state of qemu. It then begins to single step in qemu and advance time in ModelSim, ensuring that all register changes match between the two. As soon as there is a mismatch, or a ModelSim change takes too long to occur, it outputs bug information and returns control to the GDB prompt.

- `leg-lockstep-auto`: Same as `leg-lockstep`, except that it immediately resumes lockstepping after every found bug, initializing ModelSim with the correct state.

- `leg-lockstep-gui`: Opens the ModelSim GUI and allows wave configuration before running `leg-lockstep`. Signals appear in real time in ModelSim, and the simulation is stopped when a bug is reached. The procedure described in Section 3.2 is preferred over this command for its increased speed and versatility when debugging processor signals.

- `leg-jump` *`BREAK_LOC`*: Convenience function to jump to a given location in the kernel. This sets a breakpoint at `BREAK_LOC`, continues to it, and then automatically removes the breakpoint.

- `leg-frombug` *`BUGFILE`*: Jumps to the last matching state before a bug. `BUGFILE` should be a path to a bug output file, specified relative to the `debug_lockstep` directory. The resulting state is the last state for which qemu and ModelSim changed together correctly, before the given bug was detected. You can run `leg-lockstep` to check if the bug still occurs, or run `leg-checkpoint` to create a checkpoint for ModelSim debugging.

- `leg-count`: Prints an estimate of the current instruction count.

- `leg-checkpoint` *`NAME`*: Dumps the current qemu state, then opens ModelSim and converts the qemu state to a ModelSim checkpoint. `NAME` gives the desired filename of the created checkpoint. Note that this command can only be used if your version of ModelSim supports checkpoints. Student Edition does not. NOTE: This command has been mostly superseded by `leg-qemu-full-dump`.

- `leg-qemu-full-dump`: Saves qemu's current register state as `qemu_state_dump` and saves qemu's memory as `qemu_mem_dump.dat`. These files can be used by `qemuDumpRestore.tcl` to initialize the processor simulation to a known state.
- `leg-restart`: Stops qemu and restarts it at the beginning of the kernel execution.
- `leg-stop`: Gracefully shuts down qemu and then exits GDB. You should use this instead of `quit`, because otherwise the qemu process will continue in the background and will have to be killed manually.

4 LEG Processor Overview

This section presents an overview of the LEG processor, including the instruction set and memory architecture. LEG has a five stage pipeline with Fetch, Decode, Execute, Memory, and Writeback stages. Most control logic is implemented in the decode stage, and conditional execution is checked in the execute stage.

4.1 Instruction Set

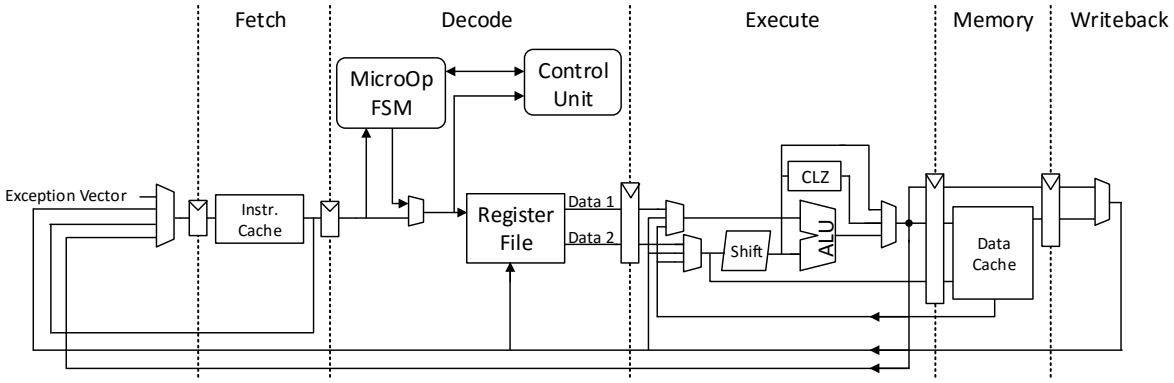

Figure 1: LEG processor core overview. More detailed diagrams are shown in the relevant sections below.

LEG supports a subset of the ARMv5 instruction set. Known exceptions are the entire Thumb instruction set, LDM(2), and the rotate functionality of memory instructions. The rotate functionality is not implemented because it appears to be unsupported in the version of qemu used to debug LEG. All other addressing modes and instructions are supported, and any bugs that are discovered can be reported to the corresponding authors listed on the first page of this document.

The processor core is split into three main components. The datapath contains the register file and hardware to execute arithmetic and memory operations, the controller sets multiplexers and other datapath control signals to create the specific behavior of each instruction, and the hazard unit detects dependencies between instructions and stalls the pipeline to resolve them. A simplified overview of the processor core is summarized in Figure 1. More information about these subsystems can be found in later sections. The datapath is described in Section 5, the controller in Section 6 and the hazard unit in Section 7.

The full processor core diagram is shown in Figure 2 and is also included as a high quality PDF in the `documentation/diagrams` directory.

4.2 Memory System

The memory system includes L1 instruction and data caches, a TLB, and corresponding address translation hardware shown in 3. The bus connecting the peripherals, translation

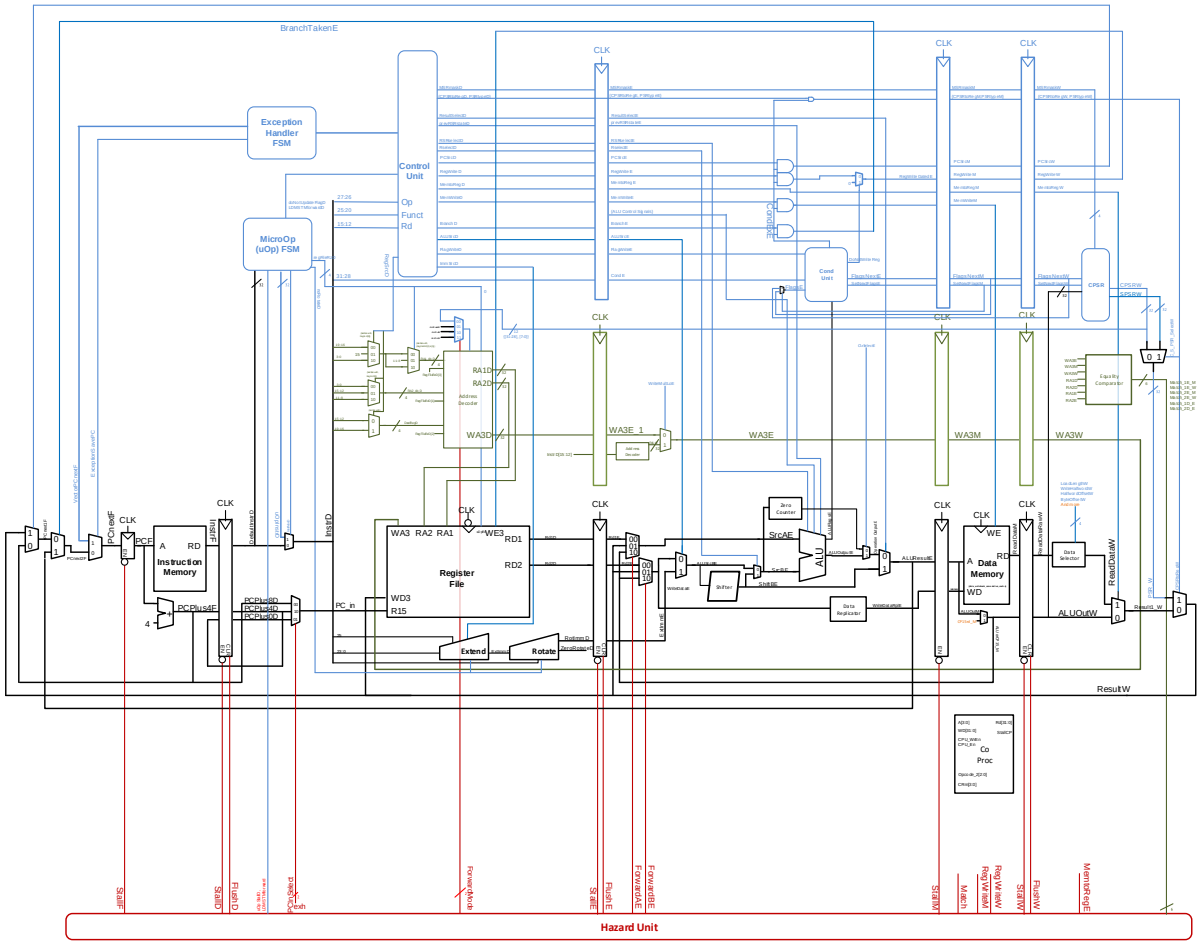

Figure 2: LEG pipeline showing datapath (black), controller (blue), hazard unit (red), and a register file decoder also called the addresspath (green). A high resolution PDF is included in the diagrams directory.

hardware, and caches is compatible with the AHB-Lite protocol, but does not implement burst mode, protection control, or HRESP. Descriptions of several of the caches can be found in sections 8 and 9.

4.3 Exception and Interrupt Handler

The LEG interrupt handler supports all exception types and privileged modes with the Base Restored Abort Model. The interrupt handler is implemented as a finite state machine in `leg_pipelined/exception_handler.sv`. This state machine stalls the correct pipeline stages and controls the datapath to branch to the corresponding exception vector. More information about the exception handler can be found in Section **??**

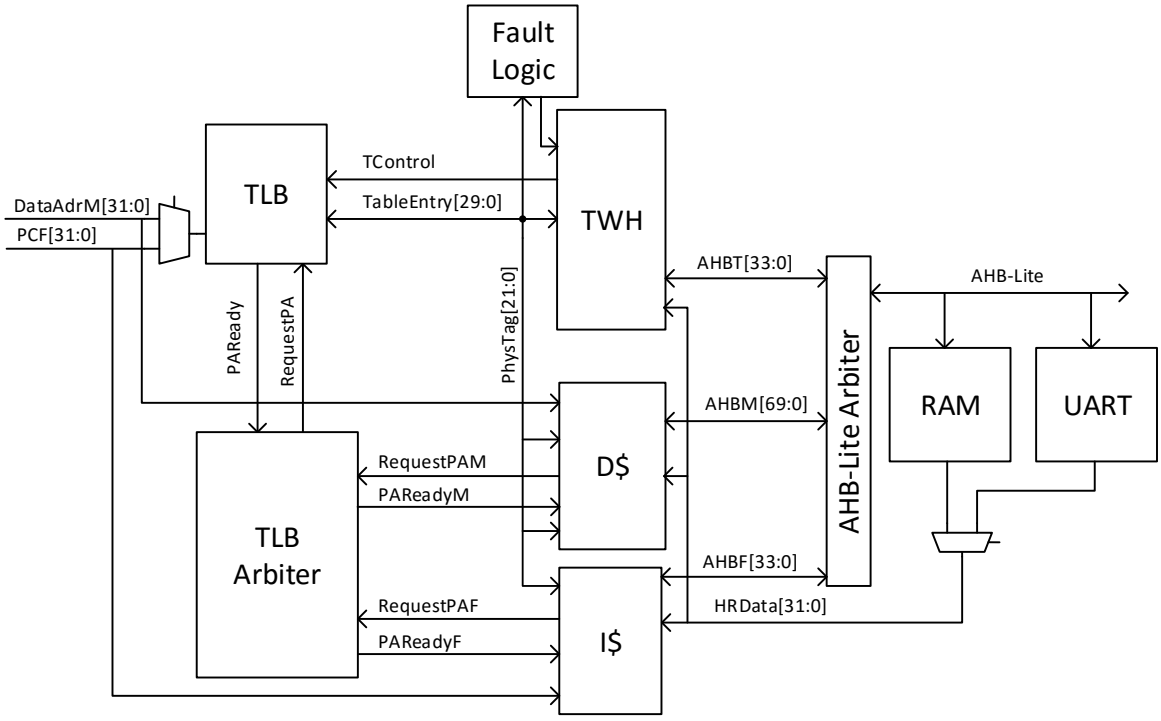

Figure 3: Memory system including physically tagged caches, address translation hardware, and bus. The translation walk hardware (TWH), data and instruction cache all desire access to the bus (right). The AHB-Arbiter arbitrates bus control among those three modules. Instead of using two TLBS, one for each cache, one (top left) is shared among the instruction cache and data cache to conserve space. Similar to the AHB-Arbiter, the TLB arbiter mediates control over the TLB request and ready signals.

5 Datapath

This section gives an overview of the LEG datapath. The datapath is pipelined with five stages: Fetch, Decode, Execute, Memory, and Writeback. The stages can be flushed or stalled by the Hazard unit, and data flow is selected by the Controller module.
Datapath components are illustrated in black on the processor diagram
`documentation/diagrams/pipelinedfull.pdf`

5.1 Relevant Files

File	Description
datapath.sv	Contains all datapath logic and signals for the LEG processor.
adder.sv	A parameterized adder.
addressdecode.sv	Decodes a 4 bit register number and the current processor mode to a one-hot register file select signal.
regfile.sv	A 32 entry by 32 bit register file containing the 31 general purpose registers and one scratch register for micro operations. Located in the Decode stage.
extend.sv	Handles extending immediate values of varying width to 32 bits. Located in the Decode stage.
rotator.sv	A funnel shifter that rotates the 32 bit immediate in data processing instructions. Located in the Decode stage.
barrel_shifter.sv	An efficient two stage logarithmic barrel shifter that implements LSR, ASR, LSL, ROR, and RRX shift types. Located in the Execute stage.
alu.sv	Arithmetic Logic Unit that computes and selects between addition, AND, OR, and XOR operations. Located in the Execute stage.
zero_counter.sv	Outputs the number of leading zeros in its input. This module is a good target for optimization in class projects. Located in Execute stage.
data_replicator.sv	Selects the size of data required for a memory operation and replicates it to fill the available byte positions. Located in Execute stage.
data_selector.sv	Masks data words read from memory and extends them to 32 bits. Located in Writeback stage.

5.2 Fetch Stage

The Fetch stage reads an instruction from instruction memory using the register 15, the program counter (PC). This PC is selected from several sources based on branch or exception status. The instruction memory is shown in the datapath in the processor diagram, but is actually implemented externally to the processor core. See Section **??** for more information.

5.3 Decode Stage

In the Decode stage registers are read and immediates are extended and rotated to the form required for arithmetic processing. Note that the instruction processed in the decode stage may not match the instruction read in the execute stage due to micro operation decoding (see Section 6.4).

5.4 Execute Stage

The Execute stage processes instruction operands into a resultant value. First forwarded register values are optionally selected based on signals from the Hazard unit (Section 7.4). Shifts or rotates are applied to the second operand and results are computed by the zero counter and ALU. The second operand is also processed as data for a potential memory operation. Finally, the output is selected from the correct functional unit based on the instruction type.

5.5 Memory Stage

The values computed in the Execute stage may be used as data and address in the Memory stage. The result may also bypass the data memory when a store operation is not performed. As with the instruction memory, the data memory is drawn in the datapath but is actually a separate functional unit described in Section **??**.

5.6 Writeback Stage

In the Writeback stage the result of the Memory stage is fed back to be written to registers or the PC.

6 Controller

The LEG controller maintains the flags and status register and handles control flow of the datapath. The controller has the same pipeline stages as the datapath, but most signals are generated in the Decode stage. These signals then advance through the processor in step with the corresponding instruction.

6.1 Relevant Files

File	Description
controller.sv	Contains instruction decode logic to control datapath flow. Also includes micro operation state machine, exception handler, and program status registers.
micropsfsm.sv	Mealy finite state machine that decodes complex instructions into sequences of simpler instructions. The instruction output by this module is used in the remaining controller decode logic.
shift_control.sv	Selects shift types, decodes shift amount, and determines shifter carry out from datapath barrel shifter.
alu_decoder.sv	Generates control signals for the ALU operation, inputs, and flags.
memory_mask.sv	Generates memory mask for storing different subsets of complete data words.
conditional.sv	Checks conditional execution and kills writeback signals if an instruction should not be executed. Also generates the resultant flags of an instruction.
exception_handler.sv	A Mealy state machine that choreographs stalls, flushes, and branching when interrupts or exceptions are detected.
cpsr.sv	Contains current and saved program status registers. Handles flag updates and verifies mode changes.

6.2 Pipeline stages

The controller Decode stage receives an instruction from the micro op FSM and generates datapath control signals. Many control signals are needed for the execute, memory, and writeback stages. These are pipelined to follow the corresponding instruction, including any flushes or stalls.

Conditional execution is checked in the Execute stage. If an instruction fails the condition check it continues to propagate through the datapath but all writeback and forwarding signals are killed. Thus the instruction has no effect on processor state.

6.3 CPSR and Flags

After a CPSR update the new flags and mode bits are forwarded to the next instruction before propagating to the writeback stage. The pipeline is also stalled as necessary so that the correct registers are read after any mode change. This ensures correct operation of conditional and privileged instructions.

6.4 MicroOp State Machine

The micro operation state machine is responsible for decoding complex instructions into sequences of simpler instructions. It takes the fetched instruction as input and outputs the instruction that is used in the rest of the processor logic. Additionally, the state machine outputs extra signals to enable special functionality of some instructions, such as preserving

a carry bit from a previous micro op stage. The state transition diagram and instruction outputs of the micro op state machine is shown in Figure 4. Table 2 lists the number of instructions issued for most micro operations.

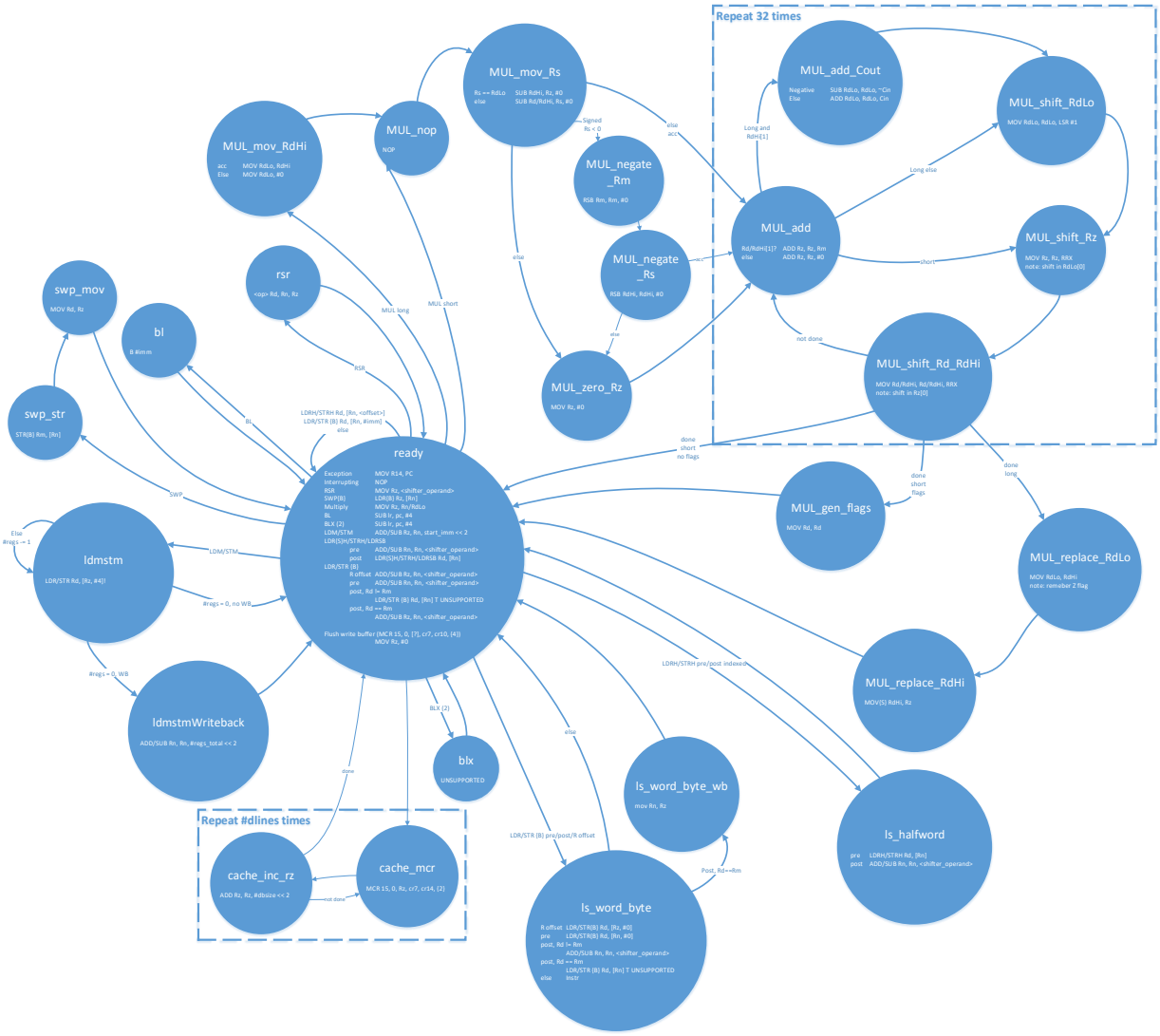

Figure 4: State transition diagram for the micro op state machine. A high quality PDF is available as `documentation/diagrams/micropfsm_drawing`

6.5 Exception Issue State Machine

The LEG exception handler is implemented as a state machine that allows interrupts and exceptions to be prioritized. When exceptions are detected the state machine allows the instructions before the exception-generating instruction to finish propagating through the pipeline but blocks any further instructions. Once the final pre-exception instruction has completed Writeback the state machine triggers saving of the program counter, mode switching, and

Instruction	Operations
Exception	1
Write buffer flush	2 x #dlines + 1
LDRH / STRH	1-2
LDR / LDRB / STR / STRB	1-3
Register shifted register	2
BL / BLX	2
SWP / SWPB	3
LDM / STM	2-18
MUL / MLA	98-100
UMULL / UMLAL	133-166
SMULL / SMLAL	135-168

Table 2: Number of micro-operations corresponding to each complex instruction

branching to the configured exception vector. This delay is necessary because pre-exception instructions may change the processor state including the behavior of the exception.

7 Hazard Unit

The Hazard unit detects potential conflicts between pipeline stages and stalls, flushes, or forwards values between stages to resolve them.

7.1 Relevant Files

File	Description
hazard.sv	Generates flush, stall, and forward signals for the datapath and controller based on instruction types and processor state.
eqcmp.sv	Compares register numbers at different pipeline stages to inform forward signals.

7.2 Stalls

All pipeline stages can be stalled, but the Execute, Memory, and Decode stages are stalled only because of data and instruction cache behavior. The Fetch stage is additionally stalled during micro ops, coprocessor and CPSR operations, PC writes. The Decode stage can be stalled for micro ops and by the exception handler.

Both Fetch and Decode are stalled during a read after write (RAW) hazard when a register is loaded from memory. Other RAW hazards are handled by forwarding.

7.3 Flushes

Various pipeline stages are flushed in the same circumstances as they are stalled. This behavior kills writeback signals during stalls so actions are not performed multiple times.

Additional flushes take place whenever a branch is taken since LEG does not implement branch prediction.

7.4 Forwarding

Read after write (RAW) hazards occur when a register that is written by one instruction is read by a later instruction before the change has propagated through to the Writeback stage. Most RAW hazards can be solved by forwarding values from the Memory or Writeback stages to the Execute stage rather than stalling. These hazards are detected by comparing the register number being used as an operand in each stage and then selecting the appropriate value to forward. This forwarding is gated by the corresponding stage's register writeback signal so values from unexecuted or flushed instructions are not used.

8 Data Cache

The data cache is a writeback, 2 way associative cache. Figure 5 includes a diagram of the data cache. The cache uses physical tagging and virtual indexing, so the number of bytes in each way is limited to the size of a tiny translation page (1024 bytes). The default cache size is 64 lines $\cdot$ 4 words/line $\cdot$ 4 bytes/word = 1024 bytes. The number of lines per way is parameterized, and the replacement policy implemented is Least Recently Used (LRU).

8.1 Relevant Files

Table 3 shows the files that are used by the data cache and Table 4 lists the top level inputs and outputs.

8.2 Data Cache States

Below is an explanation of the data cache states.

1. READY The ready state is the default state for the data cache. Upon a cache hit, the data cache will remain in the READY state.

2. WRITEBACK The cache enters the writeback state to writeback each word from the line to memory. If the data cache attempts to replace a dirty line, then it is written back first. The data cache stays in this state for at least three cycles to writeback the first three words from the dirty line.

3. LASTWRITEBACK: The last writeback state in the cache. During this state, the data cache does not request a new memory access. It waits for the last writeback to complete.

4. MEMREAD: Data cache enters this state on a cache miss to read data from memory. The cache reads four words sequentiall from the AHB Bus.

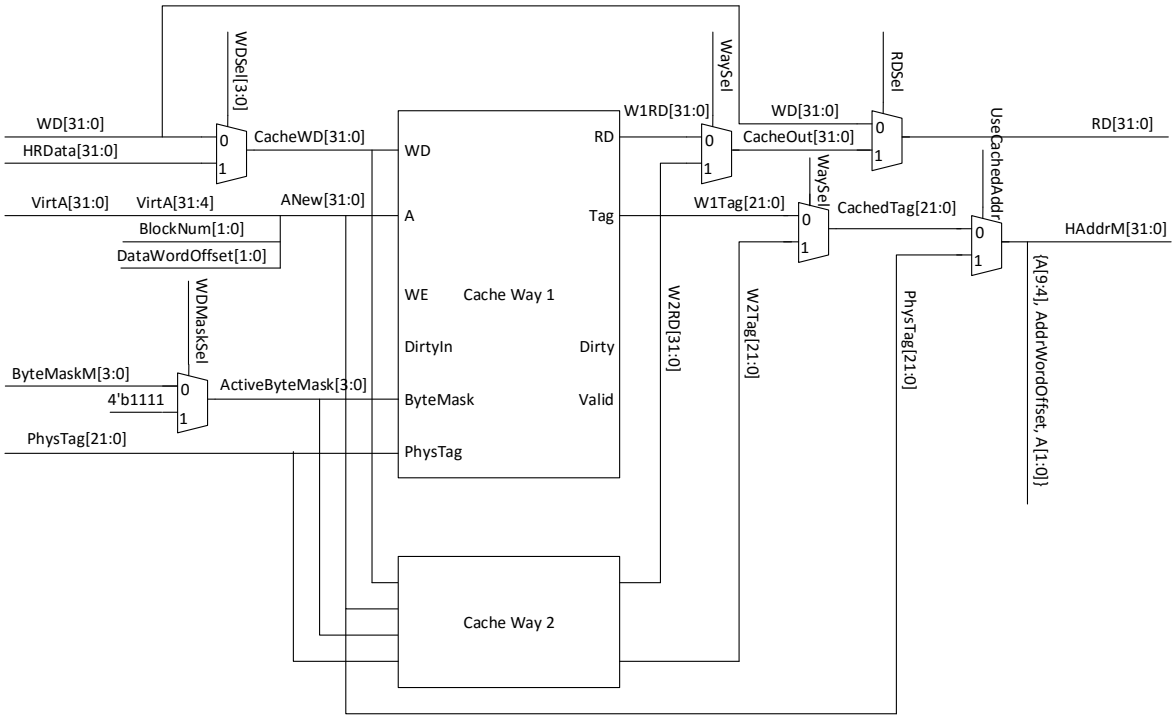

Figure 5: Diagram of 2 way set associative data cache. Note that many of the control signals are not shown in this diagram for simplicity. Also, the second cache way is identical to the first.

File	Description
data_writeback_associative_cache.sv	Top level D$ module
data_writeback_controller.sv	Controller logic for the D$. Contains the primary state machine described in section 8.2
data_writeback_associative_memory.sv	Memory module containing the both cache ways, the LRU memory, and way selection mux's. This module is used in both the instruction and data caches. The instruction cache fixes the dirty and clean inputs to zero, because it is a read only cache.
data_writeback_associative_cache_way.sv	Contains the memory associated with one cache way. This includes four words per line along with the valid, dirty, and tag bits.
word_memory.sv	Byte addressable word memory

Table 3: Data cache files

Port	Description	I/O
clk	Clock input	I
reset	Global reset signal	I
MemWriteM	Write signal from datapath	I
MemtoRegM	Read signal from datapah	I
IStall	Instruction cache stall. Used to avoid multiple data acesses for the same instruction	I
VirtA[31:0]	Virtual address from the leg datapath	I
WD[31:0]	Write data from LEG datapath	I
CP15en	Enable signal from the coprocessor	I
Inv	Invalidate line signal from the coprocessor	I
AddrOp	Indicates coprocessor is invalidating or cleaning using a virtual address instead of a set index. When high, only clean or invalidate on a hit in the data cache.	I
InvAllMid	When InvAllMid and Inv are high, invalidate all lines from cache. This signal is driven by the coprocessor	I
CurrCBit	Cachable bit for the current TLB entry	I
PAReady	Indicates TLB entry is valid for the data cache. The data cache uses the physical address and control bits from the TLB	I
PhysTag[tbits-1:0]	Physical Tag from the TLB	I
ByteMaskM[3:0]	ByteMask from leg controller	I
HRData[31:0]	Data from the AHB Bus	I
BusReady	AHB Ready signal	I
MSel	Indicates data cache has control of the AHB Bus	I
Stall	Stall signal from the data cache controller	O
RequestPA	Request a physical address from the TLB	O
HWData[31:0]	AHB Write data	O
RD[31:0]	Data output from the cache. This is the same as HWData.	O
HAddr[31:0]	AHB write address	O
HSizeM	AHB write size	O
HRequestM	Request AHB control	O
HWriteM	AHB write enable	O

Table 4: Data cache I/O (data_writeback_associative_cache.sv)

5. LASTREAD: The last memread state in the cache. During this state, the data cache does not request a new memory access. It waits for the last memory read to complete.

6. NEXTINSTR: The next instr state removes the stall on the pipeline and allows the instructions to move one stage down the pipeline. If this stage did not exist, then the instruction at the data stage would remain the same and after the requested data is retrieved, the same data would be retrieved again.

7. WAIT: The data cache enters this state when simultaneous data and instruction stalls occur. The data cache has bus precedence, so after it is done using the bus it waits for the instruction cache to retrieve data before handling the next request. This state avoids repeating data acesses when the processor stalls.

8. DWRITE: This state handles disables full word writes to main memory.

File	Description
instr_cache.sv	Top level I$ module
instr_cache_controller.sv	Controller logic for the I$. Contains the primary state machine described in section 9.2
data_writeback_associative_memory.sv	Memory module containing the both cache ways, the LRU memory, and way selection mux's. This module is used in both the instruction and data caches. The instruction cache fixes the dirty and clean inputs to zero, because it is a read only cache.
data_writeback_associative_cache_way.sv	Contains the memory associated with one cache way. This includes four words per line along with the valid, and tag bits.
word_memory.sv	Byte addressable word memory

Table 5: Instruction cache files

9 Instruction Cache

The instruction cache is a read only, 2 way associative cache. The instruction cache uses physical tagging and virtual indexing, so the number of bytes in each way is limited to the size of a tiny translation page (1024 bytes). The default cache size is 64 lines $\cdot$ 4 words/line $\cdot$ 4 bytes/word = 1024 bytes. The number of lines per way is parameterized, and the replacement policy implemented is Least Recently Used (LRU). The instruction cache uses the same memory modules as the data cache, but disables the dirty and clean signal. The controller logic is much simpler for the instruction cache, because it is read only.

9.1 Relevant Files

Table 9.1 shows the files that are used by the instruction cache and Table 9.1 lists the top level inputs and outputs.

9.2 Instruction Cache States

Below is an explanation of the instruction cache states.

1. READY The ready state is the default state for the instruction cache. Upon a cache hit, the instruction cache will remain in the READY state.

2. MEMREAD: Instruction cache enters this state on a cache miss to read data from memory. The cache reads four words sequentially from the AHB Bus.

Port	Description	I/O
clk	Clock input	I
reset	Global reset signal	I
uOpStallD	Micro-Op stall signal. This signal is used to prevent repeated memory assesses when the pipeline is stalled	I
A[31:0]	Virtual address from the leg datapath	I
CP15en	Enable signal from the coprocessor	I
AddrOp	Indicates coprocessor is invalidating using a virtual address instead of a set index. When high, only invalidate on a hit in the instruction cache.	I
InvAllMid	When InvAllMid and Inv are high, invalidate all lines from cache. This signal is driven by the coprocessor	I
Inv	Invalidate signal from the coprocessor	I
PhysTag[tbits-1:0]	Physical Tag from the TLB	I
PAReadyF	Indicates TLB entry is valid for the instruction cache. The instruction cache uses the physical address and control bits from the TLB	I
FSel	Indicates instruction cache has control of the AHB Bus	I
BusReady	AHB Ready signal	I
HRData[31:0]	Data from the AHB Bus	I
RD[31:0]	Data output from the cache. This is the same as HWData.	O
IStall	IStall signal from the instruction cache controller	O
HRequestF	Request AHB control	O
HAddrF[31:0]	AHB address	O
RequestPA	Request a physical address from the TLB	O

Table 6: Instruction cache I/O (instr_cache.sv)

3. LASTREAD: The last memread state in the cache. During this state, the instruction cache does not request a new memory access. It waits for the last memory read to complete.

4. NEXTINSTR: The next instr state removes the stall on the pipeline and allows the instructions to move one stage down the pipeline. If this stage did not exist, then the instruction at the fetch stage would remain the same and after the requested instruction is retrieved, the same data would be retrieved again.