Dual Processor MIPSfpga

ECE 404

University of Rochester

Contents

1 INTRODUCTION

1.1 CONVENTIONS

1.1.1 FONT

Commands to be run in your shell are marked as follows:

```
$ ls verilog
```

Additionally, the typography will be used to indicate the following:

bold	Used to indicate processor signal names.
italic	Used to provide emphasis and indicate files.
`typewriter`	Used to indicate addresses and shell commands.
`typewriter bold`	Used to indicate registers and labels in assembly code when used outside sample code.
`typewriter italic`	Used to indicate the instantiation path of Verilog modules.

1.1.2 NOTES & WARNINGS

Throughout this document, you will find notes in boxes like the ones shown below. Some provide useful tidbits of information that will make working with this system easier, while others highlight potential dangers.

These notes will provide pointers and shortcuts, in addition to asides that are not essential to the current task. Feel free to gloss over these.

Pay attention to these warnings, as they often mention tripping points.

Information in these boxes is of critical importance.

1.2 PROJECT DIRECTORY

The RTL and supporting programs and tools are organized as follows:

- All Verilog RTL files are in the *verilog* directory.
- Scripts and tool for compiling/assembling files and for simulation are in the *bin* directory.

- Example programs are in the *programs* directory.
- Boot code for the processor (used by the assembler when generating initialization files) is in the *boot* directory.
- Documentation on the MIPSfpga system and MIPS architecture is in the *doc* directory.
- If installed, binaries and supporting files for the LLVM MIPS cross compiler are in the *cross* directory.
- C header files for interfacing with I/O are in the *include* directory.

In addition, there is a top-level *README* and *Makefile*. The *README* contains a summary of how the project is organized, what needs to be installed, and how to simulate the system. The *Makefile* contains the rules necessary to build and simulate the testbench. Of interest will be the following commands:

- `make testbench` will create a model of the testbench described in *verilog/testbench.v*.
- `make simulate` will create and simulate the testbench model, opening the dumped waveforms for viewing.
- `make install-toolchain` will install the LLVM MIPS cross compiler. See Section 3.2.4 for more information.
- `make clean` will remove any generated models, waveform, and initialization files.

By default, the *Makefile* will not echo the full commands used, instead giving an abbreviated command. The full invocations of each phase can be seen by calling

```
$ make V=true [command]
```

2 MIPSFPGA SYSTEM

The uniprocessor MIPSfpga system is a soft core processor defined in about 12,000 Verilog statements. At the core of this system is an enhanced version of the M14Kc processor (called the *microAptiv UP* in some MIPS literature). Figure 1 shows a block diagram of the uniprocessor MIPSfpga system, with the M14K core in the dashed box.

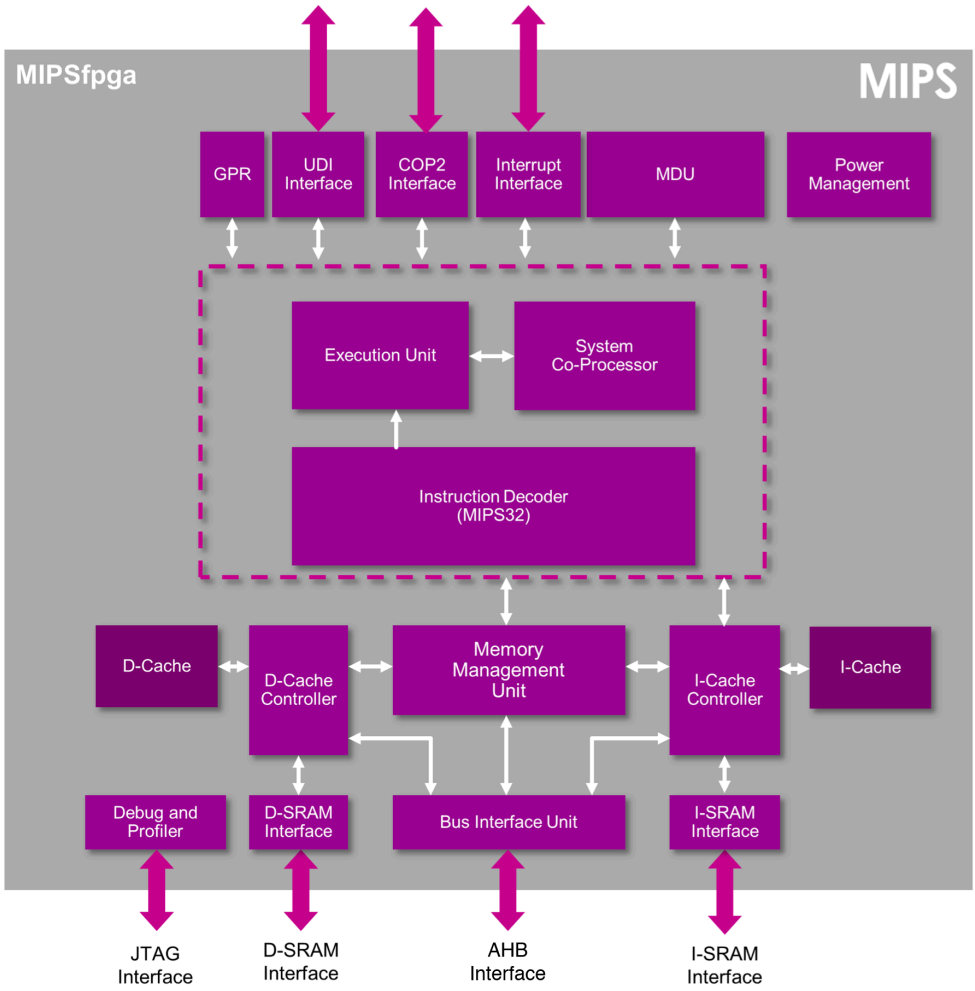


FIGURE 1: MIPSFPGA SYSTEM

Central to the M14K core is the *Execution Unit*, which carries out the instructions as determined by the *Instruction Decoder*. The *System Co-Processor* provides signals such as the system clock and reset, and allows for configuration of the system. Memory requests are processed by the *Memory Management Unit*, which connects to the separate instruction and data caches through their respective cache controllers. These are connected to the *Bus Interface Unit*, which allows the processor to interface with the AHB bus to communicate with I/O and memory.

The interfaces at the top, such as those for interrupts, COP2 and UDI allow for external interrupts, interfacing with a coprocessor (such as a GPU or DSP unit), and executing user defined instructions, respectively. As these are not relevant to the design of a multiprocessor system, they will not be described further - more information can be found in the *MicroAptiv*

UP Datasheet MD00939 document.

Similarly, the interfaces to the JTAG, D-SRAM and I-SRAM are used for accessing debug information and interfacing with scratchpad memory. As scratchpad memory has not been enabled, and we will be using a simulator, obviating the need for JTAG, these will not be described further - more information can be found in the *MicroAptiv UP Integrator's Guide MD00941* document.

The AHB interface is described in detail in Section 2.3.

The dual-processor MIPSfpga effectively replicates all the structures seen in Figure 1, connecting two processors (the entity in the grey box) to the same AHB bus (and through which the same memory and I/O).

2.1 M14K PROCESSOR

2.1.1 PIPELINE

The M14K processor is a simple in-order processor with a five-stage pipeline. The pipeline stages are as follows:

- *Instruction Fetch* (I Stage)
  - The virtual address of the instruction is translated into a physical address.
  - Instructions are fetched from the instruction cache or main memory.
- *Instruction Execution* (E Stage)
  - Instruction operands are fetched from the register file.
  - For arithmetic or logical operations, the ALU begins to compute the result.
  - For loads/stores, the ALU computes the effective address.
  - For branches, the ALU determines the branch target and determines whether a branch is taken.
- *Memory Fetch* (M Stage)
  - Arithmetic ALU operations complete.
  - The virtual effective address of a load/store is translated into a physical address.
  - The data cache (or memory) is accessed.
- *Align* (A Stage)
  - Data for loads is aligned to its word boundary.
- *Writeback* (W Stage)
  - Results of instructions are written back to the register file.

2.1.2 DELAYED BRANCHING

Like other MIPS processors, the M14K utilizes delayed branching, and has a one-cycle branch delay. This means that the instruction following a branch is executed irrespective of whether the branch is taken (this instruction is considered to be in the *branch delay slot*). It is the responsibility of the programmer (for assembly programs) or the compiler (for higher-level programs) to place an instruction following every branch – if no useful instruction can be moved into the branch delay slot, a NOP instruction must be inserted instead.

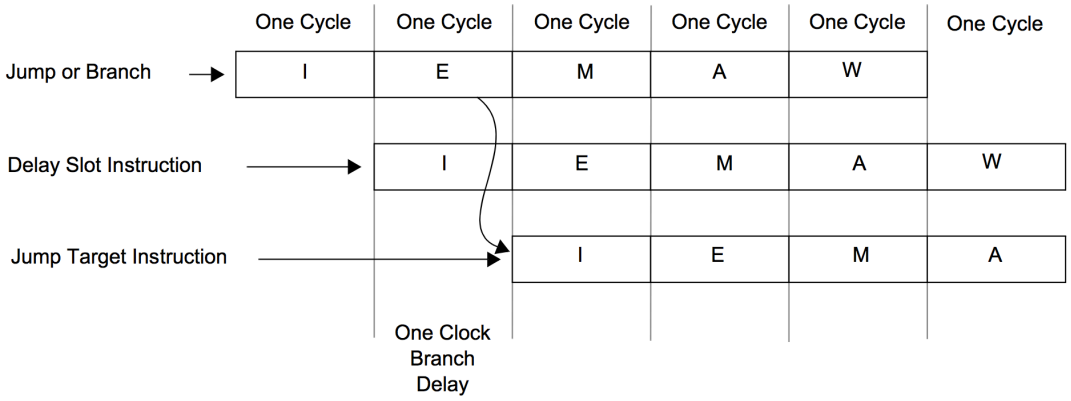


FIGURE 2: TIMELINE OF BRANCH EXECUTION WITH DELAYED BRANCHING

2.2 MEMORY MAP

Like most modern processors, the M14K contains a *Memory Management Unit* (MMU). The MMU translates virtual addresses used by software to physical addresses used by the caches and memory subsystem. The MMU by default supports paging of virtual memory into physical frames, with tranlations cached in *Translation Lookaside Buffers* (TLBs) for performance.

Typically, paging is managed by the operating system. Because we will be writing bare-metal applications, which run on the processor without the support of an operating system, programs should be placed in regions of the virtual address space that are not paged. As shown by the memory map in Figure 3, the `kseg0` and `kseg1` regions are unmapped, and instead undergo a fixed translation (such regions are often called the *Direct Map*).

Virtual addresses in `kseg0` are translated to physical addresses by subtracting `0x80000000` from the virtual address. Thus, it maps from the virtual addresses `0x80000000` to `0x9FFFFFFF` into the physical addresses `0x00000000` to `0x1FFFFFFF`. Similarly, virtual addresses in `kseg1` are translated to physical addresses by subtracting `0xA0000000` from the virtual address. As a result, `kseg0` and `kseg1` are mapped to the same region in physical memory.

Caches are disabled for accesses to addresses in the `kseg1` region, so the physical memory or I/O registers are accessed directly. Caching will be explained further in Section 2.4.

The MIPSfpga system uses memory-mapped I/O, which means that I/O is controlled by

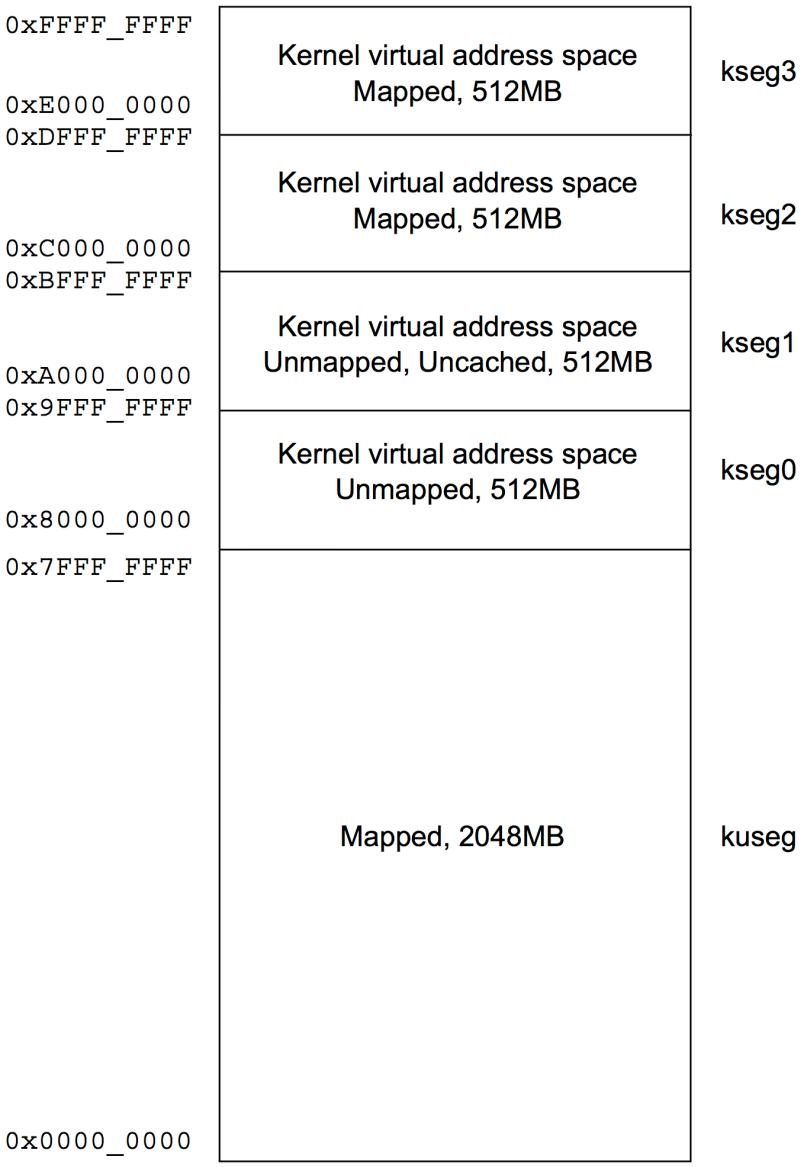


FIGURE 3: VIRTUAL ADDRESS SPACE OF THE M14K PROCESSOR

writing to specific memory locations. The I/O address space is located within `kseg1`, from virtual address `0xBF800000` to `0xBF80000C`. The I/O address space is mapped as follows:

TABLE 1: I/O ADDRESS SPACE

Address	I/O	Read/Write	Description
`0xBF800000`	LEDR	Write Only	Red LEDs
`0xBF800004`	LEDG	Write Only	Green LEDs
`0xBF800008`	SW	Read Only	Switches
`0xBF80000C`	PB	Read Only	Pushbuttons

2.3 AHB BUS INTERFACE

Both processors communicate with memory and I/O over an AHB interface. While the AHB is a split-transaction capable bus, for simplicity the AHB in this system has been implemented with atomic transactions only: once a master acquires the bus, it will not release it until its transaction has completed.

2.3.1 BASIC TRANSFERS

An AHB basic read or write trasfer consists of three phases:

- The *Arbitration* phase, where a master acquires the bus.
- The *Address* phase, where the master places the address to read/write onto the bus. This lasts until the **HREADY** signal is high.
- The *Data* phase, where data is transmitted between the master and slave. This may require several cycles. The **HREADY** signal is used to control the number of clock cycles necessary for the transfer.

In the MIPSfpga system, **HREADY** is hardwired HIGH. Because of this, slaves cannot insert any wait states, so the *Address* phase always lasts one cycle, and the *Data* phase takes one cycle per beat of the transfer.

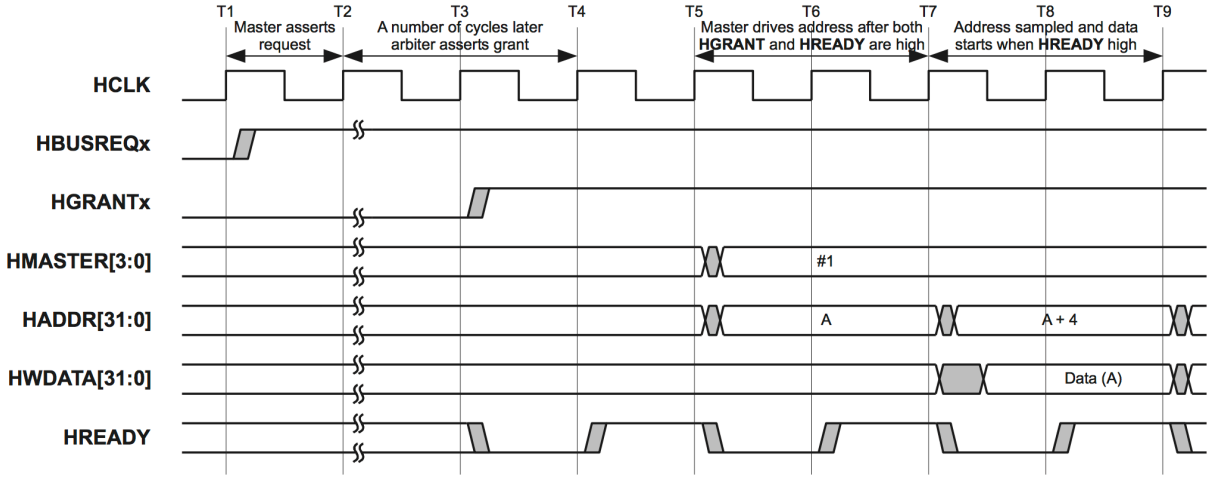


FIGURE 4: TIMING OF AN AHB TRANSFER

Information about the signals used by the AHB interface can be found in Section 6.1.

During the *Arbitration* phase, a master asserts **HBUSREQx** to inform the bus arbiter that it requests access to the bus. Depending on the contention for the bus, the arbiter will grant the bus by asserting **HGRANTx** as soon as the current cycle, or after a delay of multiple cycles. Once the **HGRANTx** and **HREADY** are both high, the *Address* phase begins, where the

master owns the address and control lines (**HADDR**, **HTRANS**, **HWRITE**, **HSIZE**, **HBURST**, and **HPROT**), but not the data bus (cannot write to **HWDATA** and should not read **HRDATA**). At this point, the master no longer needs to assert **HBUSREQx**. As mentioned previously, this phase lasts until **HREADY** is high, after which the *Data* phase begins, where the master owns the address, control and data lines.

When a transfer is about to complete (usually because there is one word left to transfer), the arbiter will deassert **HGRANTx** for the current master and assert **HGRANTx** of the next master (if one exists). An example of bus handoff is shown in Figure 5. Note how **HGRANTx** changes during the last beat of Master #1's burst, while **HMASTER** is not changed until Master #1's burst completes.

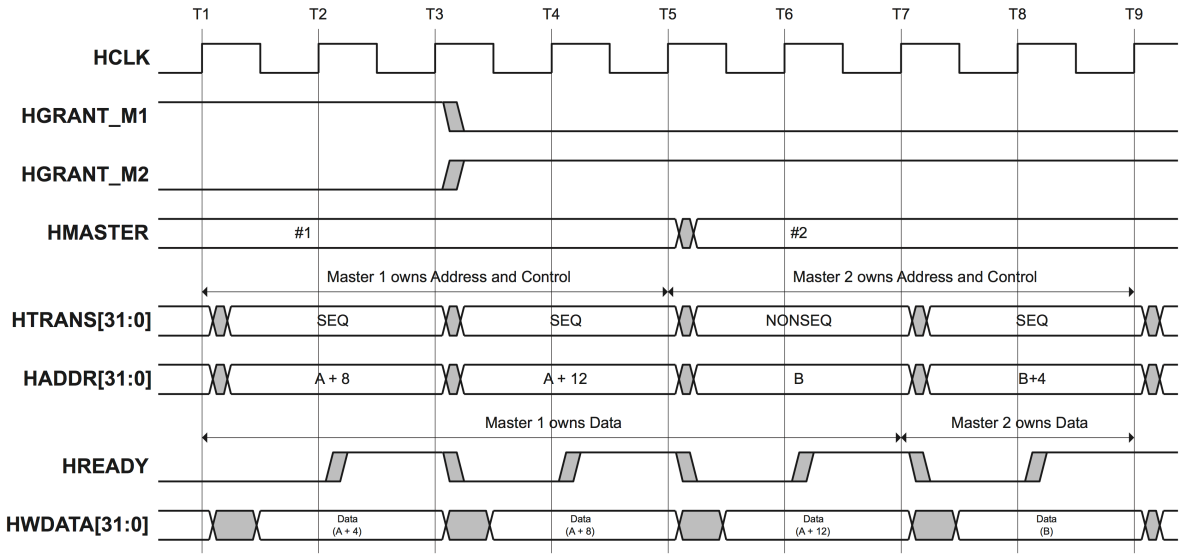


FIGURE 5: OWNERSHIP OF THE AHB BUS

2.3.2 BURST OPERATION

The burst type is indicated using the **HBURST** signal. While the bus supports all bursts described in the AMBA Rev. 2 AHB specification, the M14K bus interface unit will only generate *SINGLE* and *WRAP4* bursts. A *SINGLE* burst will transfer one word of data, whereas the *WRAP4* transfers four words of data in the same four-word line, starting at the specified address, wrapping around the line boundary as needed. For example, a *WRAP4* burst of word accesses starting at `0x34` would consist of four transfers to addresses `0x34`, `0x38`, `0x3C`, and `0x30`.

TABLE 2: BURST OPERATION TYPES

Burst Operation	**HBURST**	Description
SINGLE	3'b 000	Single Burst (one word of data)
WRAP4	3'b 010	Wrapping Four-Beat Burst

2.3.3 WRITE BUFFER

Data from the core is accumulated in a write buffer to coalesce memory requests, reducing bus traffic. The write buffer is implemented as two 16-byte buffers. One buffer contains data being transferred to the AHB interface, while the other accumulates data from the processor. These buffers allow byte, halfword, tri-byte or word writes from the core to be accumulated into a 16 byte value before sending the data to the bus. The buffer also holds dirty cache lines during an eviction.

Because write buffers can cause violations in sequential consistency, programs that expect sequentially consistent behavior may need to flush the write buffer before some loads and after some stores. The M14K processor supports flushing the write buffers using the `SYNC` instruction, which is explained in detail in the document *MIPS32® Architecture For Programmers Volume II: The MIPS32® Instruction Set*.

The write buffer's hardware design is described in Section 5.2.1.

2.4 CACHES

Each processer in the system has its own separate instruction and data caches. Both the instruction and data caches are 4KB 2-way set associative caches with 16 byte lines. Caches are virtually indexed and physically tagged. On processor reset, caches are uninitialized, and caching is disabled for all memory segments; the caches must be initialized and each segment must be made cacheable in software. The provided assembler can include necessary code into the *ram_reset_init.txt* to initialize the caches; the flags required are explained in section 3.2.3. The code required to initialize the caches can be found in *boot/init_cache.s*.

When enabled, caches use a write-back write-allocate policy (the hardware also supports write-through write-allocate and write-through no-write-allocate, though the boot code does not configure it to use these policies). On a cache miss (for either a load or store), the processor is stalled while the data is fetched from main memory.

Memory requests caused by cache missing operations take the form of a *WRAP4* transfer, with the first word being the one that was requested by the missing load/store. Once this word is returned from the memory system, the pipeline restarts, allowing instructions to execute while the rest of the cache line is being fetched. To accomodate this feature, the caches employ a fill buffer, the design of which is described in detail alongside the rest of the cache in Section 5.3.

3 SIMULATION

3.1 TOOLS

3.1.1 ICARUS VERILOG

Icarus Verilog is a Verilog compiler, which generates binaries executable by the `vvp` simulator. It has support for Linux, Windows and MacOS.

Instructions to build from source are provided in 7.1.1, though most package managers can install Icarus more easily than building from source. On Windows, an installer can be found at `http://bleyer.org/icarus/` (it compiles using the MinGW toolchain).

This project was developed using Icaraus Verilog version 10, though any future updates should not break compatability.

Icarus Verilog provides a compiler `iverilog`, and a simulator `vvp`. The `iverilog` compiler accepts as arguments a list of Verilog files, the name of the top level module (if it cannot be inferred), and the desired ouput. For example the following commands could be used to compiler the testbench for this project:

```
$ cd verilog
$ iverilog -s testbench -o testbench *.v
```

The `vvp` simulator can be used to simulate the compiled Verilog module, dumping waveforms if the module uses the `$dumpvars` command. Because of the complexity of this project, standard VCD waveform files are large (on the order of hundreds of MB), so we will instead dump the waveforms in FST format, which is accomplished by the `-fst-space` flag.

```
$ vvp -fst-space testbench
```

When simlating a module with that loads a memory from a file (using `$readmemh`, for example), `vvp` may issue the following warning:

```
WARNING: ram_reset_dual_port.v:23: $readmemh: Standard inconsistency, following 1364-2005.
WARNING: ram_reset_dual_port.v:23: $readmemh(ram_reset_init.txt): Not enough words in the
 ↪  file for the requested range [0:32767].
WARNING: ram_dual_port.v:19: $readmemh: Standard inconsistency, following 1364-2005.
WARNING: ram_dual_port.v:19: $readmemh(ram_program_init.txt): Not enough words in the file
 ↪  for the requested range [0:65535].
```

This is indicating that the size of the file is too small to fill the memory. Ususally this is of no concern, and the regions not specified in the initialization files will be initialized to don't-care values.

For full details on the usage of `iverilog` and `vvp`, consult their `man` pages. In general it should not be necessary to invoke either `iverilog` or `vvp` directly, as the provided *Makefile* has commands to build and simulate the processor.

3.1.2 GTKWAVE

GTKWave is a free waveform viewer that can read the waveforms dumped by Icarus's `vvp` simulator.

Once a waveform file (dump.fst) has been generated by Icarus, it can be viewed by calling

```
$ gtkwave dump.fst
```

This will open GTKWave and load the *dump.fst* waveform file. An example with a few open waveforms is shown in Figure 6.

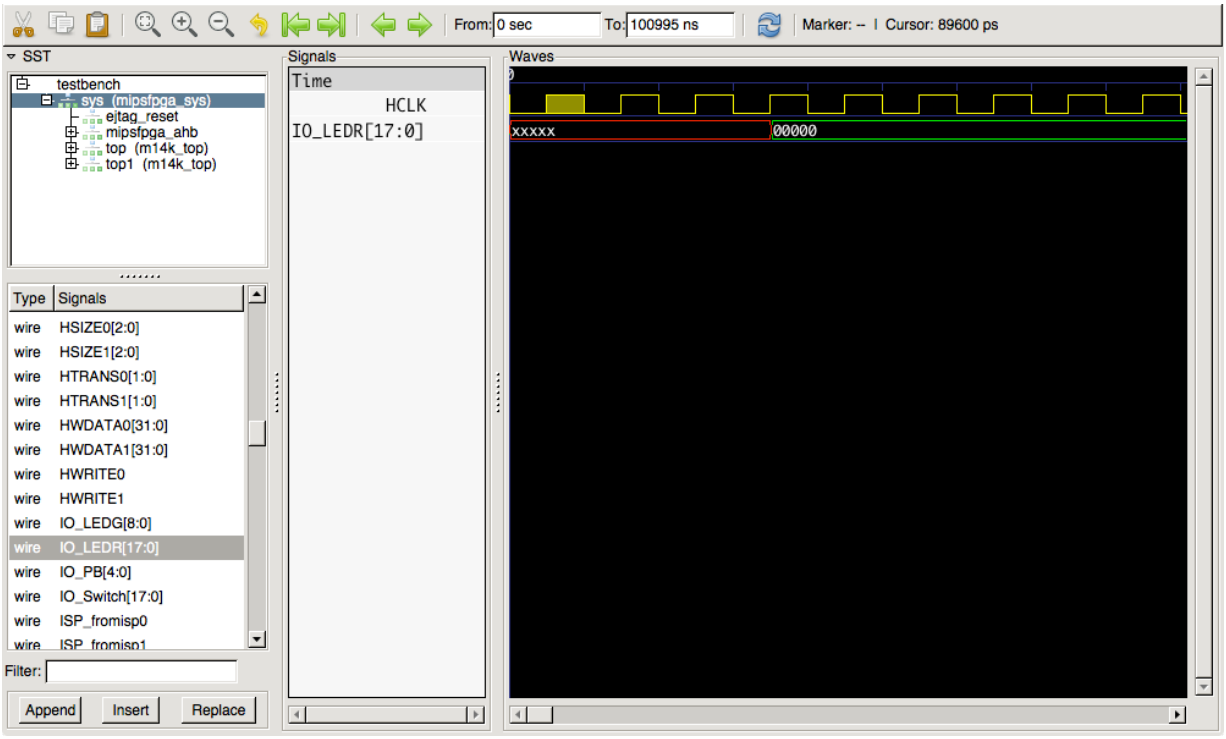


FIGURE 6: GTKWAVE WAVEFORM VIEWER

In the top-left box are the instantiated modules whose waveforms have been dumped, with the module name shown in parenthesis if it differs from the instantiation name. Clicking on a module will reveal the signals for that module in the bottom-left box. Selecting these signals and pressing "Append" will make the signals viewable in the waveform window. Additionally, there is a filter box which enables searching for a specific signal and supports regular expressions in queries.

It is possible to change the data format and color of a signal by right-clicking it within the "Signals" box.

When viewing waveforms, it often is helpful to zoom out as much as is shown in Figure 6 to be able to see how signals change across multiple clock cycles. Sometimes, it can be useful to zoom out to view the entire simulation, which can be done with the leftmost zoom button.

More information on GTKWave can be found in *GTKWave 3.3 Wave Analyzer User's Guide*.

3.2 WRITING PROGRAMS

3.2.1 INTRODUCTION

The MIPSfpga system uses the MIPS32 Release 2 instruction set. Information on the instruction set can be found in *MIPS32 Architecture For Programmers Volume II The MIPS32 Instruction Set* and *MIPS32_QuickReferenceCard*. The system is a big-endian architecture, which means that the address of a word or halfword refers to the most significant byte.

Programs and data are loaded into memory using two files: *ram_reset_init.txt*, which contains system initialization code and *ram_program_init.txt*, which contains assembled user programs and data. These files contain a hexadecimal representation of the assembled programs to be run on the system. They can be generated using a provided assembler program, or (if a full LLVM toolchain supporting MIPS as a target is installed) using the provided scripts to compile simple C programs. To enhance readability, comments can be placed in these files using a double forward slash (as in C/C++).

While the MIPSfpga system has 4GB of virtual address space as described in Section 2.2, it only has 384KB of physical memory: a 256KB RAM from physical addresses `0x00000000-0x0003FFFF`, and a 128KB RAM from physical addresses `0x1FC00000-0x1FC1FFFF`. Physical addresses `0x1F800000-0x1F80000C` are used for GPIO.

The 256KB RAM is initialized using *ram_program_init.txt*, and the 128KB RAM is initialized using *ram_reset_init.txt*.

TABLE 3: INITIALIZATION FILES

File	Physical Address Range
ram_reset_init.txt	`0x1FC00000-0x1FC1FFFF`
ram_program_init.txt	`0x00000000-0x0003FFFF`

3.2.2 CHANGES FOR A DUAL-PROCESSOR DESIGN

Since MIPSfpga was originally designed as a uniprocessor system, its M14K processor lacks the coprocessor registers traditionally used by software to know which core it is running on. Instead, CPU0 and CPU1 were given separate reset vectors (where they begin fetching from after a reset). On startup, CPU0 will fetch from `0xBFC00000` and CPU1 will fetch from `0xBFC00008` (these are virtual addresses: the physical addresses will be `0x1FC00000` and `0x1FC00008`, respectively).

The assembler provided with this project is aware of the starting vectors of each processor, and makes writing multiprocessor programs for this processor simple by use of the `CPU0` and `CPU1` labels.

Take for example the following program, *onoff.s*, which repeatedly turns on and off one of the red LEDs.

```
# CPU 0 will start executing here
CPU0:
    addiu   $s0, $0, 1
    lui     $s1, 0xbf80       # LEDR addr
LEDR_ON:
    sw      $s0, 0($s1)       # Turn on LEDR
    j       LEDR_ON           # Loop
    nop

# CPU 1 will start executing here
CPU1:
    lui     $s1, 0xbf80       # LEDR addr
LEDR_OFF:
    sw      $0, 0($s1)        # Turn off LEDR
    j       LEDR_OFF          # Loop
    nop
```

When this program is assembled with the provided assembler using the command

```
$ bin/assemble programs/onoff.s
```

the resulting *ram_reset_init.txt* will look like the following:

```
// Initialization Code
0BF00004 // [0xbfc00000] j $boot-CPU0
3C1AA000 // [0xbfc00004] lui $k0, %hi(CPU0)
0BF00007 // [0xbfc00008] j $boot-CPU1
3C1AA000 // [0xbfc0000c] lui $k0, %hi(CPU1)
375A0000 // [0xbfc00010] $boot-CPU0: ori $k0, $k0, %lo(CPU0)
03400008 // [0xbfc00014] jr $k0
00000000 // [0xbfc00018] sll $0, $0, 0 # (pseudoinstruction of nop)
375A0014 // [0xbfc0001c] $boot-CPU1: ori $k0, $k0, %lo(CPU1)
03400008 // [0xbfc00020] jr $k0
00000000 // [0xbfc00024] sll $0, $0, 0 # (pseudoinstruction of nop)
```

This contains the code necessary to initialize the processor, and jump to the program code, defined in *ram_program_init.txt* (shown below). Because this program is being initialized into uncached memory, there is no need for any processor initialization, and control is simply transferred to the program.

```
24100001 // [0xa0000000] CPU0: addiu $s0, $0, 1
3C11BF80 // [0xa0000004] lui $s1, 0xbf80
AE300000 // [0xa0000008] LEDR_ON: sw $s0, 0($s1)
08000002 // [0xa000000c] j LEDR_ON
00000000 // [0xa0000010] sll $0, $0, 0 # (pseudoinstruction of nop)
3C11BF80 // [0xa0000014] CPU1: lui $s1, 0xbf80
AE200000 // [0xa0000018] LEDR_OFF: sw $0, 0($s1)
08000006 // [0xa000001c] j LEDR_OFF
00000000 // [0xa0000020] sll $0, $0, 0 # (pseudoinstruction of nop)
```

3.2.3 ASSEMBLER USAGE

The provided assembler will be used to create the *ram_reset_init.txt* and *ram_program_init.txt* files. It supports a majority of the instructions executable by the M14Kc processor.

The assembler accepts a few flags to customize its behavior; calling `bin/assemble --help` will list them. Of interest will be the `--cached` flag, which adds cache initialization to the boot code and places the program into a cacheable region of memory. The `--init-registers` flag will add boot code to initialize `$sp`, `$fp`, and `$ra`. Finally, the `--list-supported-instructions` flag will cause the assembler to list its supported instructions rather than assembling.

While the provided assembler is designed to work on multiprocessor programs written for the dual-processer MIPSfpga system, it is also capable of correctly assembling uniprocessor programs. When the `CPU0` and `CPU1` labels are not specified in the program, the assembler will produce boot code that will have CPU 1 spin, while CPU 0 executes the program.

3.2.4 WRITING SIMPLE C PROGRAMS

Using the LLVM toolchain, it is possible to generate MIPS assembly from C programs which can then be assembled by the provided assembler.

To be able to use this functionality, exposed through the provided `mcc` script, LLVM will need to be installed with the MIPS target.

The LLVM toolchain for the MIPSfpga system can be installed using the provided Makefile.

```
$ make install-toolchain
```

This command will install the toolchain into the *cross* directory, and will not interfere with any preinstalled software. Be aware that the toolchain is about 6GB when installed and will temporarily use more space during installation.

Installing using the provided Makefile will take some time. To speed up the installation process, pass the `-jN` flag, where `N` is the number of cores on your machine.

Because the applications will be running without the support of an operating system, `malloc`, `printf`, or any other libc functions will not work.

An example program, *onoff.c* is shown below.

```
#include <mipsfpga.h>

// CPU 0 exexutes this
void CPU0()
{
    while (1)
    {
        *LEDR = 1;
    }
}

// CPU 1 executes this
void CPU1()
{
    while (1)
    {
        *LEDR = 0;
    }
}
```

The variable `LEDR` is defined in the *mipsfpga.h* header file, and is of the type `volatile unsigned int *const`. In general, any variables shared between processors will need to be declared `volatile`.

Like with assembly programs, if `CPU0` and `CPU1` are not defined, CPU 1 will spin and CPU 0 will execute `main`.

3.3 RUNNING PROGRAMS

After compiling/assembling your program, it can be run on a model of the system by calling

```
$ make simulate
```

This will recompile the hardware model (if necessary), then run the simulator, loading the program into memory. By default, it simulate for 100 cycles before starting the processor (by asserting **SI_Reset_N**), then for 10,000 additional cycles.

After reset, each processor will begin to fetch from its start vector (`0xBFC00000` for CPU 0, and `0xBFC00008` for CPU 1); these requests can be viewed by observing the AHB signals of the system. After executing code necessary for initialization, each processor will jump to their respective parts in the program. When caching is enabled, this is at address `0x800xxxxx`, and when caching is disabled, at `0xA00xxxxx`, with the lower order bits of the address depending on the program being executed.

Remember, the addresses mentioned here are *virtual* addresses, so the physical addresses will be offset by `0xA0000000`. The addresses seen on the AHB bus are always *physical* addresses.

Until caches are enabled by software, all memory requests will be made to main memory. As a result, the instruction throughput of each processor is significantly diminished: only one instruction (per processor) is fetched about every five cycles. Specifically, this delay comes from:

- 1 cycle to recognize that the instruction is not in cache
- 1 cycle to send the request to the *Bus Interface Unit* (BIU)
- 1 cycle for the BIU to gain access to the AHB bus
- 1 cycle for the BIU to place a read request on the bus
- 1 cycle for main memory to return the data via the bus
- 1 cycle to return data from the bus to the CPU

Because both processors will be making requests to main memory, it may take more than one cycle to acquire the bus; refer to Section 2.3 for full details.

3.3.1 EXAMPLE SIMULATION: CACHING DISABLED

Part of a simulation of the *onoff.s* program is shown in Figures 7 and 8. For reference, the source of the program is in Section 3.2.2.

For this example, it was was assembled without any special options, using

```
$ bin/assemble programs/onoff.s
```

Figure 7 shows the waveforms of the system immediately after **SI_Reset_N** is asserted. The top three signals (in green) are the system clock (**SI_ClkIn**), the active-low soft reset (**SI_Reset_N**) and the red LEDs (**IO_LEDR**). The next five signals (in purple) are the AHB signals to and from the memory subsystem. Details on the meaning of each signal can be found in Section 6.1. The next five signals (in yellow) are the AHB signals to and from CPU 0, and the final five signals (in orange) are the AHB signals to and from CPU 1. The AHB signals to memory are multiplexed from either CPU 0 or CPU 1, depending on which owns the bus (see Section 2.3 for more information).

The **SI_ClkIn** and **SI_Reset_N** signals are in the *testbench/sys* module, while the AHB signals are in the *testbench/sys/mipsfpga_ahb* module. The *testbench/sys/top* module is CPU 0 and *testbench/sys/top1* module is CPU 1.

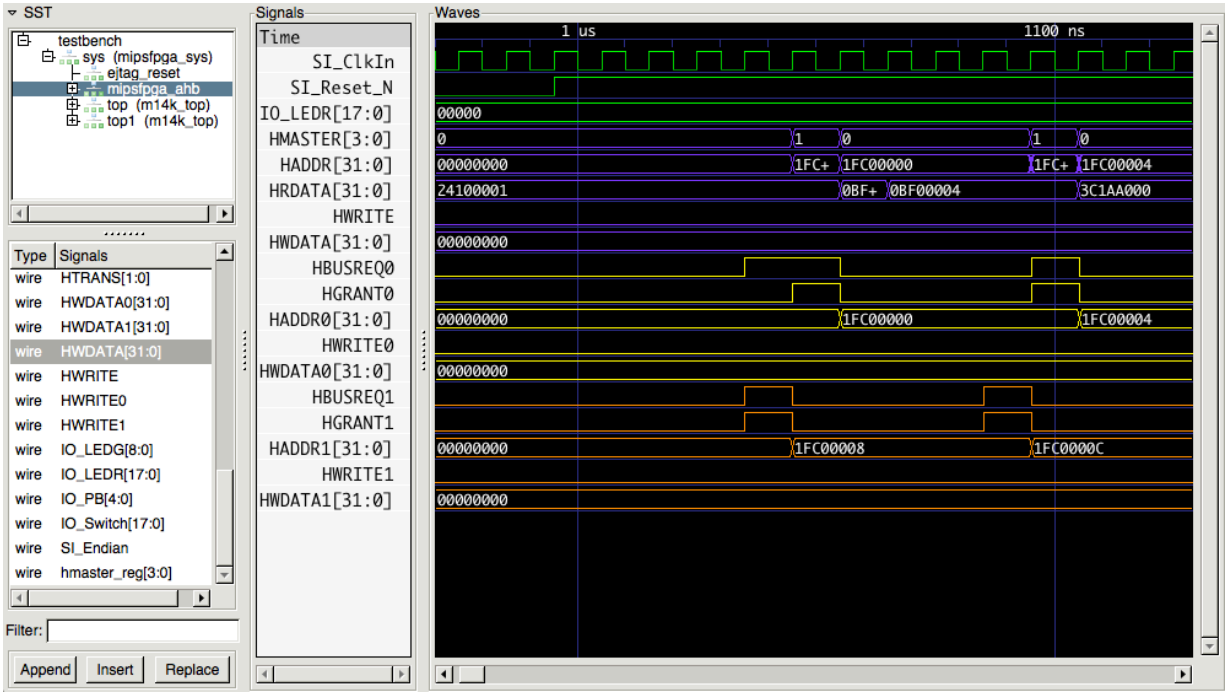


FIGURE 7: INITIALIZATION OF SYSTEM RUNNING ONOFF

As shown in Figure 7, a few cycles after reset, both processors assert their respective **HBUSREQ** lines, and CPU 1 is given access to the bus by the arbiter. The next cycle, CPU 1 places its address onto the address bus, and CPU 0 is given the bus grant. One cycle later, CPU 0 places its address onto the address bus, and CPU 1 gets the word corresponding to its address on the **HRDATA** bus. CPU 0 gets the word at its address the next cycle. A few cycles later, this process repeats.

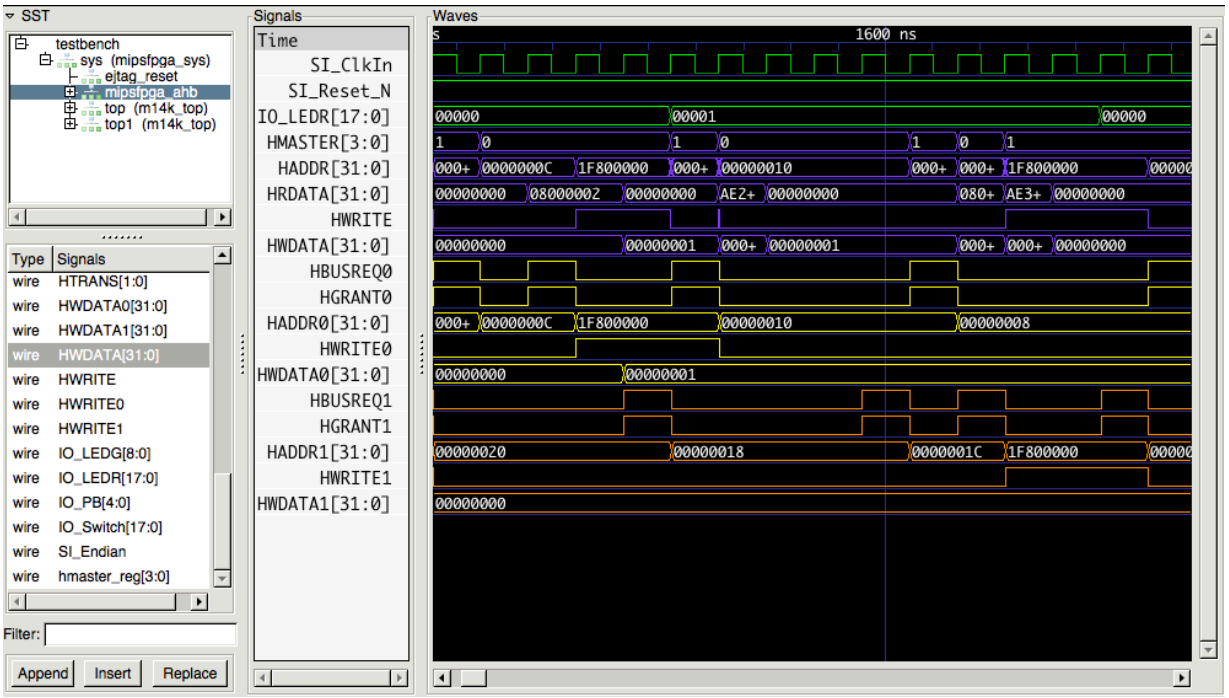


FIGURE 8: EXECUTION OF ONOFF PROGRAM

The waveforms shown in Figure 8 show the waveforms after each processor has jumped into its program code. To write a value to `LEDR`, CPU 0 first makes a bus request (just as it would for any read request). Once acquiring control of the bus, it places the address of `LEDR` (`0x1F800000`) onto the address bus and asserts its **HWRITE** signal, indicating that it this is a write request. In the next cycle, when it has control of the data bus, it places the desired value onto the write data bus, **HWDATA**. In the following cycle, **IO_LEDR** changes value.

One might note how CPU 0 made a request to access the bus even though **HMASTER** was set to CPU 0. This request was necessary because **HMASTER** was only set to CPU 0 because it was the last master to use the bus, not because there was an active transaction. This is because the AHB is permitted to arbitrarily set **HMASTER** when the bus is idle.

3.3.2 EXAMPLE SIMULATION: CACHING ENABLED

The simulation of *onoff.s* is significantly different when caching is enabled. In this example, the program was assembled using

```
$ bin/assemble --cached programs/onoff.s
```

A full view of the simulation is shown in Figure 9. The first section of the simulation, between the start and marker *A* is similar to the uncached version: both processors sit idle for 100 cycles, **SI_Reset_N** is asserted, and both processors begin fetching their initialization code. Marker *A* marks the start of initializing the instruction cache. For roughly 6000 cycles until

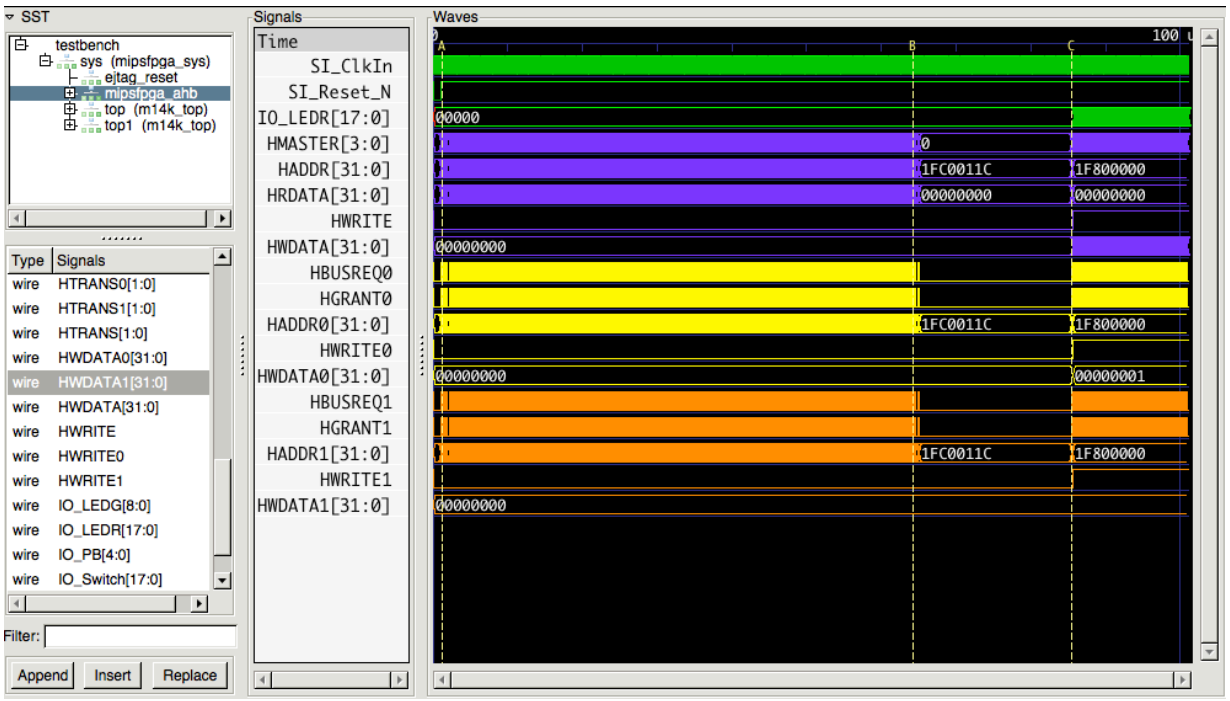


FIGURE 9: EXECUTION OF CACHED ONOFF PROGRAM

marker *B*, each line in the instruction cache is cleared and invalidated. At marker *B*, the processor jumps to the `kseg0` cacheable address space and begins to initialize the data cache. Because

instructions are cacheable at this point, there is no bus activity for most of the range between markers *B* and *C*, as the instructions necessary to invalidate the data cache are loaded into the instruction cache shortly after marker *B*. The instruction cache significantly lowers the latency of instruction fetch, so it takes only 2000 cycles to initialize the data cache, even though the data cache is the same size as the instruction cache. Marker *C* is placed at the end of the initialization code, and marks the start of executing the onoff program.

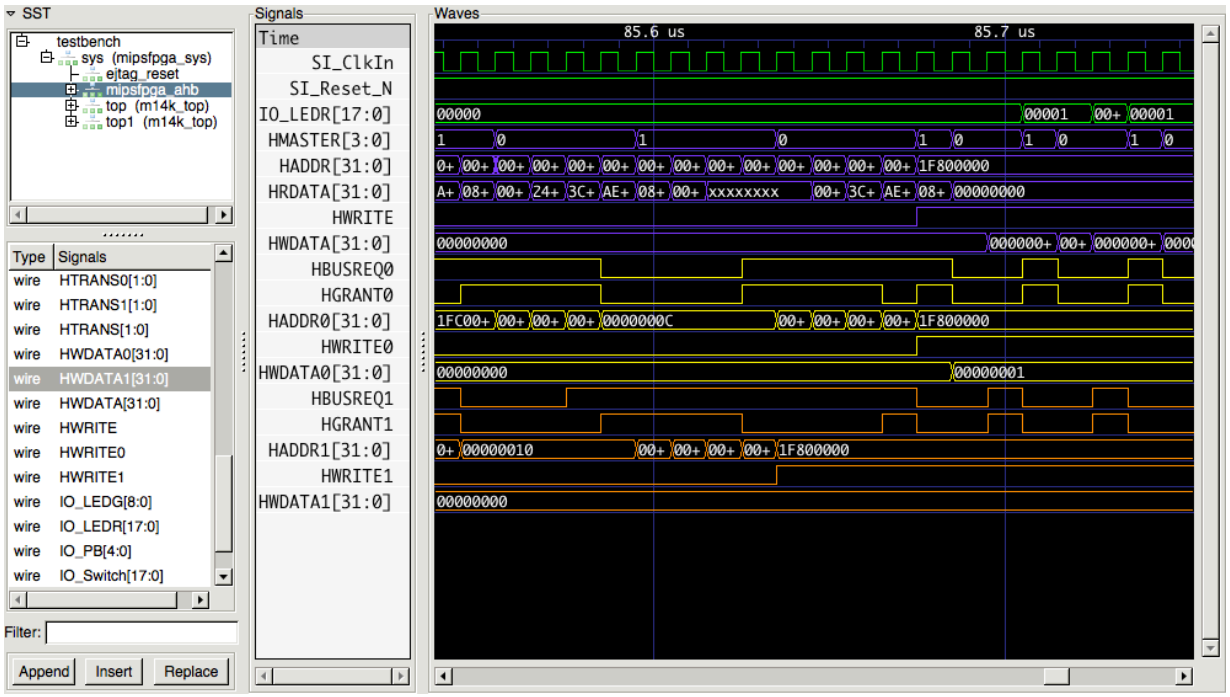


FIGURE 10: ZOOMED EXECUTION OF CACHED ONOFF PROGRAM

A section of the simulation after marker *C* is shown in Figure 10. Immediatelty prior, CPU 0 requested access to the bus; it acquires the grant at the beginning of this section. Unlike in the uncached case, it makes a *WRAP4* burst, requesting addresses `0x0`, `0x4`, `0x8`, and `0xC` in one transaction. In the last cycle of its burst, CPU 1 is granted the bus (recall that control of the address and control lines changes one cycle later). Note how CPU 1 maintains a request for the bus during two transactions: this is because the processor pipeline restarts after getting the first instruction word, and the processor executes a store instruction before the instruction fetch completes. CPU 1 is forced to wait for CPU 0 to make a request before it can make the second request because the AHB arbiter gives priority to the master which does not currently own the bus. Also, note how each CPU holds **HWRITE** high after making their first request to write to `LEDR`. This is not a correctness concern, as repeating the same transaction (i.e. storing the same data to the same location) is the same as leaving the bus idle, and if the requested address by either CPU were to change, **HWRITE** would adjust accordingly.

3.4 EXCEPTION VECTORS

In general, a software exception will cause the processor to jump to `0xBFC00380`, although there are other exception vectors that the processor may jump to depending on the exception (see *MicroAptiv UP Software User's Manual MD00942*, table 5.7 for more details). When either processor encounters an exception, it is a good indicator that either there is a flaw in the software being run, or (if the software has worked previously) in the hardware.

When an exception occurs, the Coprocessor 0 Cause register (register 13, select 0) will contain information about the exception. Tables 6.27 and 6.28 in *MicroAptiv UP Software User's Manual MD00942* provide more information. If the provided assembler is invoked with the `--debug` flag, it will add boot code so that on an exception, the PC of the instruction that caused the instruction will be placed in register `$k1`, and the exception code will be placed in register `$k0`.

4 ADDING CACHE COHERENCE

4.1 MODIFICATIONS TO AHB

The AHB was not designed with coherency support: it will need to be extended to add coherency features. Namely, it will need an additional line so masters can assert a Read Exclusive (RdX) request, which will cause any snoop units to invalidate their cache lines associated with the address being exclusively read. It will also need a mechanism to allow other bus masters to intervene, or supply data in place of main memory; a data bus multiplexed into **HRDATA** would be sufficient. To enable snooping, each master will need to be aware of when it is the current bus master, so it will not snoop itself, and will need to have the multiplexed **HADDR** directed back to each master.

4.2 MODIFICATIONS TO CACHE

To support snooping capability, the tagram and dataram within the data cache, and the fill buffer within the data cache controller will need to be multiported.

> In a real processor implementation, multiporting would prove *far* too expensive. A more reasonable (but more difficult) implementation would bank the cache, and rely on the unlikely nature of a snoop (which only occurs on a S to M transition or a cache miss) occuring at the same time as a data cache request where both requests use the same bank. In the case of both a snoop and data cache access requiring the same bank on the same cycle, the data cache access would be stalled.
>
> Since this processor will never be implemented in silicon, there is no problem with multiporting for this project.

The coherency status will need to either be included in one of the existing RAMs (likely the tagram since it already contains the valid bit), or placed into a separate RAM instance. If the coherency status is included in an existing RAM instance, it will need to have a second write port added in addition to a second read port.

When a line is first placed in the cache, depending on the status of the line in the fill buffer, it may be inserted in any of the three coherency states. In the case where the fill buffer entry is invalid, the line should be placed in the cache in the Invalid state (there exists a small timing window in which this is possible). When the fill buffer is valid and dirty, the line should be placed in the Modified state, and in the Shared state when the fill buffer is valid and not dirty.

4.3 SNOOPING

With the above modifications in place, the steps for snooping depend on where the data resides.

Take care to ensure that the intervened data and intervene request line have the same timing as **HRDATA** (one cycle after **HADDR**).

4.3.1 CACHE

1. Read the tagram, check the valid bit and compare the tags to determine if there is a hit in any of the ways.

2. If there is a hit in the cache, use the information on which way hit to select the way for the coherency status (if using a separate RAM, otherwise this is handled in step 1).

3. If the bus request is a Read Exclusive and the line is in Shared or Modified state, invalidate the line and service the data. It is important to invalidate the line *before* servicing the intervention to prevent any concurrent stores to the line from being squashed by the invalidation. There is no risk of a premature eviction because the processor servicing the intervention does not have control of the bus, and could thus not fetch a line to cause an eviction.

   If the bus request is not a Read Exclusive and the line is in Modified state, change to Shared state and service the data. As with the above case, it is important to change state before servicing the intervention so that any concurrent stores will be stalled until they can generate their own Read Exclusive requests (otherwise it would be possible to have the same line be in Modified state in one cache and Shared state in the other).

   If the bus request is not a Read Exclusive and the line is in Shared state, simply service the data.

4.3.2 FILL BUFFER

1. Use a similar process as the one used to generate **hit_fb** to determine if there is a snoop hit. When comparing tags, it may be easier to use **fb_tag** as opposed to comparing the high and low portions of the tag as is done to determine **hit_fb**. Unlike with the cache, each of the four words in the fill buffer has its own valid bit.

   Be aware that there exists a (small) timing window where the fill buffer may need to be checked before it is completely filled. In this case, it may be necessary to intervene for only part of the transaction. This does not present a problem for correctness, as any dirty words will be present in the fill buffer before any other requests can be placed on the bus.

Note that it is also possible for the coherency state to change while the data is within the fill buffer: interventions need to respect the coherency state at all points.

2. If there is a hit, and the fill buffer is dirty (**fb_dirty** will be set), the fill buffer needs to transition from the modified state to the shared state by ignoring **fb_dirty** when initializing the coherency status. Because the dirty bit still needs to be set (it *is* newer than the data in main memory, and there is no guarentee that the other processor will write the data back to main memory), **fb_dirty** cannot simply be cleared.

4.3.3 WRITE BUFFER

1. Check for a hit in both the front and back write buffers. This is accomplished by comparing the tags (see Section 5.2.1 and checking whether the entry is valid. Each write buffer maintains a valid bit within the tag in addition to a valid bit for each byte in a separate register (this is used to track validity when write-through caches are enabled); it will be sufficient to check only the valid bit within the tag.

2. If there is a hit, service the intervention. If the bus request was a Read Exclusive and the hit was in the front write buffer, invalidate the entry. There is no need to invalidate entries in the back write buffer (the processor cannot hit in the back entry of its own write buffer).

4.4 GENERATING COHERENCY TRAFFIC

When transitioning from either the Invalid or Shared state to the Modified state, the other processor needs to be alerted so it can invalidate its copy. This is most easily handled by generating a Read Exclusive request to the bus. When in the Invalid state, a store will generate a store allocate request (see the logic for **store_allocate** and **other_req** in *m14k_dcc.v*). A cache hitting store can be detected similarly to how **store_allocate** detects a cache missing store.

Initially, it will be easier to generate a Read Exclusive on *all* stores (both store allocate and cache hitting, irrespective of state) until the coherency state machine is working. Then, an additional check for Shared state can be made before generating a request on cache hitting stores.

When generating a request for a cache hitting store, it will be easiest to make use of the existing mechanisms for generating read requests in the case of a store allocate (see **other_req** in *m14k_dcc.v*). However, care must be taken to disable the fill buffer in this case (more information can be found in Section 5.3.3), otherwise the data in the cache will be overwritten.

5 HARDWARE STRUCTURES AND SOURCE FILES

5.1 REGISTER FILE

The register file for each core is instantiated as *sys/top(1)/cpu/core/rf/rf_reg0*. Viewing register contents may help with debugging.

5.2 BUS INTERFACE UNIT

The Bus Interface Unit (BIU) allows each processor to communicate with memory and IO through the AHB bus. Information on the AHB interface can be found in section 2.3. Defined in *m14k_biu.v*, it is instantiated as *sys/top(1)/cpu/core/biu*.

5.2.1 WRITE BUFFER

The write buffer is defined within the BIU. It contains two entries, *front* and *back*; *front* is used to accumulate data from the processor, and *back* is used to hold data being communicated to the AHB bus. The data present in *front* can be accessed from **wr_buf_f**, and the data present in *back* can be accessed from **wr_buf_b**. The tags for the front and back entries, named **wr_buf_tag_f** and **wr_buf_tag_b** are formatted as shown in Table 4 Of the fields within the tag, only the valid and address fields are used to determine if there is a hit in the write buffer.

TABLE 4: WRITE BUFFER TAG FORMAT

Name	Bits	Description
Address	31:4	Bits 31:4 of the Physical Address (Cache Index & Tag)
Zero	3	Hardcoded to 0
Atomic	2	Used by LL/SC to detmine whether a read-modify-write was atomic
Probe Space	1	Request is to Probe Space
Valid	0	1 if the data is valid, 0 otherwise

5.2.2 SIGNIFICANT SIGNALS

Table 5 lists the names and meanings of signficant signals in the Bus Interface Unit.

TABLE 5: BIU SIGNALS

Name	Source	Description
wr_buf_addr_match Write Buffer Hit	BIU	Used to indicate whether there was a hit in the fron write buffer (1 indicates a hit, 0 a miss).

Continued

TABLE 5: BIU Signals (continued)

Name	Source	Description
wr_buf_tag_f Write Buffer Tag	BIU	Contains metadata concerning the data within the front write buffer. The format is given in Table 4.
wr_buf_f Write Buffer Data	BIU	Contains the data within the front write buffer.
wr_buf_tag_b Write Buffer Tag	BIU	Contains metadata concerning the data within the back write buffer. The format is given in Table 4.
wr_buf_b Write Buffer Data	BIU	Contains the data within the back write buffer.
dcc_exaddr Address	DCC	Used to indicate the address for read requests from the data cache controller.
dcc_exrdreqval Read Request	DCC	Asserted for one cycle to request a *WRAP4* read to the address given by **dcc_exaddr**.
dreq_a Read In Progress	BIU	Indicates that a data read request is currently in progress. This signal is rises and falls with the same timing as **HMASTER**, though it may fall sooner (since the AHB may assign a master when the bus is idle).
dreq_val Read Request	BIU	This signal rises with **dcc_exrdreqval**, but remains high until the request is completed.

5.3 CACHES

The instruction and data caches are defined in *m14k_ic.v* and *m14k_dc.v*. Each instantiates a dataram (*dataram_2k2way_xilinx.v*) and tagram (*tagram_2k2way_xilinx.v*) instance. The data cache also includes a wsram (*d_wsram_2k2way_xilinx.v*), which holds LRU data for eviction and whether a line is dirty. Each cache is controlled by a cache controller (*m14k_icc.v* and *m14k_dcc.v*). The data cache controller contains a fill buffer (*m14k_dcc_fb.v*) which is used to hold data being fetched from main memory before being placed in cache.

5.3.1 TAG RAM

The tagram is accessed to determine whether a request will hit in the cache. The fields of particular interest in the tagram are shown in Table 6.

Both **rd_data** and **wr_data** contain a bit vector of the following format for each of the two ways. To align with the hardware present on a Xilinx FPGA, this is split among two ram instances per way (see *tagram_2k2way_xilinx.v*).

TABLE 6: TAGRAM FIELDS

Parameter Name	Name	Description
clk	Clock	Clock of the RAM
line_idx	Line Index	Index used to access the line of the RAM
wr_mask	Byte Write Mask	Used to indicate which way is read/written
rd_str	Read Strobe	Set to HIGH to indicate a read request
wr_str	Write Strobe	Set to HIGH to inidcate a write request
wr_data	Write Data	Data (tag) to be written
rd_data	Read Tata	Data (tag) that was read

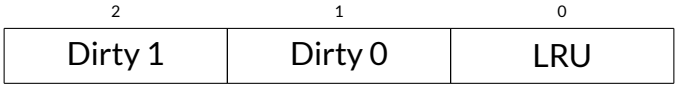
TABLE 7: CACHE TAG FORMAT

Name	Bits	Description
Tag	23:2	Bits 31:10 of the Physical Address
Lock	1	Used to prevent a line from begin evicted
Valid	0	Whether the tag is valid

The logic that controls whether a cache hit occurs is present in *m14k_cache_cmp.v* and in the files for each cache controller: *m14k_icc.v* and *m14k_dcc.v*. Because the associativity of the caches is configurable at build-time (up to 4-way), many of the bit vectors in these files are larger than necessary for our configuration.

5.3.2 WAY SELECT RAM

The way select RAM or wsram holds LRU and dirty information about each cache line. Within the way select ram, the data format is simple, though it is made more complicated once it is routed to the data cache controller. The format of a line within the way select RAM is as follows:



The respective dirty bits are set when a line has been written to and are cleared on insertion. The LRU bit identifies the least recently used cache line (0 for way 0, 1 for way 1).

Within the data cache controller, the wsram line (**dcc_wswrdata**) is 14 bits wide (which is the maximum size of a wsram line of any configuration). Rather than use that line directly, it will be easier to use the **dcc_wswrdata_dirty** signal, which contains the dirty bits of each way, and the **ws_dirty_en** signal, which indicates in which way the dirty bit will be set/unset.

5.3.3 FILL BUFFER

When a data is returned from the BIU after a data cache miss, it is not immediately placed in the data cache. Instead, it is first moved into a fill buffer until the entire cache line has been transferred. On a store miss, the stored data is placed into the correct portion of the fill buffer while the memory subsystem services the miss. It is possible for the processor to hit (with both loads and stores) in the fill buffer before its data is moved into the cache. Within the data cache controller, the **req_out** signal is used to indicate to the fill buffer that it should expect data from the BIU and that it should transfer this data to the data cache after the bus transaction completes. The fill buffer has valid bits for each of the four words that the fill buffer holds; it is possible for some words to be valid while others are not. Additionally, the **fb_dirty** signal is used to indicate whether the dirty bit will be set on a cache line after the data is moved into cache.

5.3.4 SIGNIFICANT SIGNALS

Table 8 lists the significant signals in the data cache controller and fill buffer.

TABLE 8: DCC SIGNALS

Name	Description
dcc_tagwrdata_fb Cache Fill Tag	Tag to be written when fill buffer is transferred into cache. Generated from the fill buffer tag, and has the same format as a data cache tag.
other_req Other Request	Asserted on a prefetch or store-allocate request. One of the signals used to generate **dcc_exrdreqval**.
dcc_wswrdata_dirty Way Select Dirty	Dirty bits to be written into the Way Select RAM. Generated from **fb_store_dirty**.
store_allocate Store Allocate	Set when a cache-missing store occurs with a write-allocate cache policy.
dcc_wswstb Way Select Write	Write strobe signal for the Way Select RAM. Writes occur on the rising edge of the clock when this is sampled to be HIGH.
dcc_twstb Tag Write	Write strobe signal for the Tag RAM. (See **dcc_wswstb**)
dcc_tagwren Tag Write Way Enable	Bitmask (one bit per way) of write enables for each way.
ws_dirty_en Way Select Dirty Way Enable	Bitmask (one bit per way) of write enables (for dirty bits) for each way.

Continued

TABLE 8: DCC Signals (continued)

Name	Description
dcc_tagwrdata Tag Write Data	Data to be written into the Tag RAM. When **dcc_twstb** is high, on the rising edge of the clock, the data in **dcc_tagwrdata** will be written into the ways specified in **dcc_tagwren**.
cache_hit Cache Hit	HIGH indicates that there was a hit in any way of the cache (tag match & valid).
req_out Active Request	HIGH indicates that there is currently a request outstanding, and that the fill buffer should be expecting data. When this signal falls, the fill buffer will commence transferring data into the cache.

TABLE 9: FILL BUFFER SIGNALS

Name	Description
hit_fb Fill Buffer Hit	HIGH indicates that there was a hit in the fill buffer.
valid_word_here Fill Buffer Valid	4-bit mask (1 per word) that indicates whether each word in the fill buffer is valid. One cycle delayed version of this signal, **valid_word_here_reg** is used in **hit_fb** calculation.
raw_fb_valid Fill Buffer Valid	Another 4-bit mask (1 per word) that indicates whether each word in the fill buffer is valid.
fb_data{0-3} Fill Buffer Data	Four signals (i.e. **fb_data0**, ...) that contain each of the four words in the fill buffer.
dval_m_sel Lower Physical Address	Bits 19:2 of the Physical address used to query the fill buffer. Bits 3:2 comprise the index into the fill buffer.

6 APPENDIX

6.1 AHB SIGNALS

The following is a description of the AMBA AHB signals. For greater details, consult the *AMBA Specification*.

All signals are prefixed with the letter **H** to differentiate them from other signals in the system. Signals that end in **x** are generic signals (for example, there is no **HGRANTx**, but rather a **HGRANT_CPU0** and **HGRANT_CPU1**).

TABLE 10: AHB SIGNALS

Name	Source	Description
HCLK Bus Clock	Clock Source	The clock for the bus. All timings are on the rising edge.
HRESETn Reset	Reset Controller	The signal used to reset the bus. This is the only active LOW signal.
HADDR Address Bus	Master	The Address Bus (32 bits).
HBUSREQx Bus Request	Master	Signal from a bus master to the arbiter to indicates that the bus master requires the bus.
HLOCKx Locked Transfer	Master	When HIGH this signal indicates that the master requires locked access to the bus and no other master should be granted bus access until this signal is LOW.
HTRANS Transfer Type	Master	Indicates the type of the current transfer (2 bits). Can be IDLE (00), NONSEQUENTIAL (10), or SEQUENTIAL (11).
HWRITE Transfer Direction	Master	When HIGH, indicates a write transfer; LOW indicates a read transfer.
HSIZE Transfer Size	Master	Indicates the size of the transfer (3 bits). Can be byte (000), halfword (001), or byte (010).
HBURST Burst Type	Master	Indicates if transfer is part of a burst (3 bits). Described in greater detail in Section 2.3.2.

Continued

TABLE 10: AHB Signals (continued)

Name	Source	Description
HPROT Protection Control	Master	Provide additional information about a bus access for the purposes of implementing protection (4 bits). Signals are used to indicate the type of transfer (opcode fetch, data access, etc.). The Bus Interface Unit of the M14K processor does not generate all protection information, and will set to 0010 for opcode fetch and 0011 for data access.
HWDATA Write Data Bus	Master	Used to transfer data from the master to the slave (32 bits).
HSELx Slave Select	Decoder	Used to select which AHB slave the transfer is indended for.
HRDATA Read Data Bus	Slave	Used to transfer data from the slave to the master (32 bits).
HREADY Transfer Done	Slave	When HIGH, indicates that a transfer on the bus has completed. May be driven LOW to extend a transfer.
HRESP Transfer Response	Slave	Provides status on a transfer (1 bit). May be OKAY (LOW) or ERROR (HIGH).
HGRANTx Bus Grant	Arbiter	Signal from the bus arbiter to a master to indicate that it is currently the highest priority master. Ownership of the address/control signals changes from the current master at the and of a transfer (when **HREADY** is HIGH), so a master gains access to the bus when both **HREADY** and **HGRANTx** are HIGH.
HMASTER Master Number	Arbiter	Indicates which master is currently performing a transfer. Has the same timing as address and control signals.
HMASTLOCK Locked Sequence	Arbiter	Indicates that the current master is performing a locked series of transfers. Has the same timing as the address and control signals.

6.2 SIGNAL CONVENTIONS

The following naming convention is used for signals throughout the system. More information can be found in the *MicroAptiv UP Integrator's Guide MD00941*.

TABLE 11: SIGNAL TYPES

Name	Description
In	Input to the core (unless otherwise noted). This is sampled on the rising edge of the clock.
Out	Output of the core (unless otherwise noted). This is generated on the rising edge of the clock.
AIn	Asynchronous input to the core.
SIn	Static input to the core. These must not change state after **SI_ColdReset** is deasserted.
SOut	Static outputs from the core. These never change state.

To differentiate between different interface signals, each interface signal is prefixed with a letter according to the interface to which it corresponds. In the following table, signals marked with an asterisk (*) should not be of any significance when implementing a cache coherency protocol.

TABLE 12: SIGNAL PREFIXES

Name	Description
H	Signals to the AHB interface (see Section 6.1
SI	Non-AHB system interface signals.
EJ*	EJTAG interface signals.
TC*	EJTAG Trace interface signals.
CP2*	Coprocessor 2 interface signals.
UDI*	CorExtend user-defined instruction interface signals.
ISP*	Instruction StratchPad RAM interface signals.
DSP*	Data ScratchPad RAM interface signals.
PM*	Performance monitoring signals.
gscan/bist*	Testing Signals, for scan or memory Built-In-Self-Test (BIST).
gmb*	Integrated memory BIST signals.

7 INSTALLATION GUIDES

7.1 LINUX

The instructions below have been tested on Ubuntu 14 and Ubuntu 16. On other flavors of Linux, the instructions should be the same, with `apt-get` replaced with the preferred package manager.

7.1.1 ICARUS VERILOG

Ensure that the version installed is at least 10.0, otherwise the project will not compile. The following command will reveal the version of Icarus Verilog that is installed.

```
$ iverilog -V | head -n 1
```

Most package managers can install Icarus Verilog automatically.

```
$ sudo apt-get install iverilog
```

Before installing Icarus Verilog, first ensure that its dependencies are satisfied by running:

```
$ sudo apt-get install flex bison gperf autoconf
```

Download Icarus Verilog by running:

```
$ curl -OL https://github.com/steveicarus/iverilog/archive/v10_1.tar.gz
```

Unpack the tarball and `cd` into the unpacked directory.

```
$ tar xzf v10_1.tar.gz
$ cd iverilog-10_1
```

Configure

```
$ sh autoconf.sh
$ ./configure
```

Build

```
$ make
```

Verify that the build succeeded

```
$ make check
```

The output "Hello World" should be displayed at the end of `make check`'s output.

Install

```
$ sudo make install
```

7.1.2 GTKWAVE

> Most package managers can install GTKWave automatically. Unlike with Python and Icarus Verilog, there are no strict requirements for the GTKWave version.
>
> ```
> $ sudo apt-get install gtkwave
> ```

Download GTKWave by running:

```
$ curl -OL http://gtkwave.sourceforge.net/gtkwave-3.3.79.tar.gz
```

Unpack the tarball and `cd` into the unpacked directory.

```
$ tar xzf gtkwave-3.3.79.tar.gz
$ cd gtkwave-3.3.79
```

Configure

```
$ ./configure
```

Build

```
$ make
```

Verify that the build succeeded

```
$ make check
```

Provided that there are no errors, it can be installed with

```
$ sudo make install
```

The installed version can be verified with

```
$ make installcheck
```

7.1.3 PYTHON

> Most package managers can install Python more easily than building from source. Ensure that the installed version is at least 3.5

Download Python

```
$ curl -OL https://www.python.org/ftp/python/3.5.3/Python-3.5.3.tgz
```

Unpack the tarball and `cd` into the unpacked directory.

```
$ tar xzf Python-3.5.3.tgz
$ cd Python-3.5.3
```

Configure

```
$ ./configure
```

Build

```
$ make
```

Install

```
$ sudo make install
```

Verify that the version of Python is correct

```
$ python3 --version
Python 3.5.3
```

7.2 MACOS

7.2.1 ICARUS VERILOG

> A compatible version Icarus Verilog can be installed using `brew`. When using any other package manager, ensure that the version installed is at least 10.0
>
> ```
> $ brew install icarus-verilog
> ```

Icarus verilog can be built from source as in Section 7.1.1, except dependencies cannot be installed using `apt-get`.

7.2.2 GTKWAVE

GTKWave can be installed using `brew`. Depending on Homebrew's version, the command will either be

```
$ brew install gtkwave
```

If the above succeeds, no further steps (including the *.bashrc* modification) are needed. If it fails, try

```
$ brew cask install gtkwave
```

It will be necessary to modify your *.bashrc* (or equivalent for your shell) as mentioned below to be able to use the `gtkwave` command.

Download the GTKWave application from `http://gtkwave.sourceforge.net/gtkwave.zip`. Unzip and move to your Applications directory.

Then, add the following to your *.bashrc* (or equivalent for your shell)

```
alias gtkwave='open -a gtkwave'
```

7.2.3 PYTHON

Python 3.5 can be installed using `brew`

```
$ brew install python3
```

Python 3.5 can be installed via an installer. Download the installer from `https://www.python.org/ftp/python/3.5.3/python-3.5.3-macosx10.6.pkg`. Double-click the installer and follow the instructions.

Verify that the version of Python is correct

```
$ python3 --version
Python 3.5.3
```

You may need to modify your path to ensure that `python3` points to the newly installed version.