COS/ELE 375
Spring 2021

Project 3: MIPS ISA Cycle-Accurate Simulation.

Submit via Canvas by May 5th (Dean's Date) 5:00PM ET.

Introduction

In Project 1, you created an ISA-level simulator for the main functionality of the MIPS ISA. While such a simulator is able to execute any MIPS program and give correct results, it is not enough to assess the *performance* of a given implementation. As you know, implementations of a given ISA may differ widely in performance based on their use of pipelining, their cache size and organization, etc.

In Project 3, you are to create a simulator for a five-stage pipelined implementation of the MIPS ISA (as seen in class). As in Project 1, the simulator is to be written in C++, though you may write a C-style implementation in C++ if you wish. The simulator must be capable of handling all the instructions required for the ISA-level simulator from Project 1 (**LL/SC is not included in this project in any way**). The simulator must implement full forwarding and bypassing, and a single branch delay slot (so branches must be resolved in the Decode stage). The simulator must implement a memory hierarchy with one level of cache, with separate I and D caches. It must be capable of running for a specified number of cycles and providing statistics about the execution for the set number of cycles (details are provided later in the specification).

As the poll results suggest, you will work in groups of **four**.

Note: In the rest of the specification, sections whose headings are marked with **(***)** are identical to their counterparts from the Project 1 spec and are provided here for reference.

Overview

As opposed to Project 1, where you executed an instruction all at once, in this project you will simulate how the instructions would actually execute on a five-stage pipelined implementation. As seen in class, the five stages of the pipeline are Fetch (IF), Decode (ID), Execute (EX), Memory (MEM), and Writeback (WB). Your simulator must run cycle-by-cycle - fetching, decoding, executing, and writing back instructions, and accessing memory as appropriate. In order to properly simulate the execution of instructions in the pipeline, you will need to keep track of control signals, hazards, and the like.

Your simulator must accurately simulate the following:

- Full forwarding and bypassing.
    - This includes forwarding from a load to the data input of a store (i.e. register rt) without a stall cycle in between (in other words, a WB->MEM forward from the load to the store). See page 312 in the textbook for an overview of

    such forwarding. All other cases of forwarding to a store should forward when the store is in the EX stage.
  - This also includes forwarding to ID for the execution of branches. You may need to add stall cycles between the forwarding instruction and the branch in ID for the forwarding to work; see page 319 in the text for further details.
- A single branch delay slot (so branches are resolved in the ID stage)
- One level of blocking caches (separate I and D caches) (details provided in the "Caches" section)
- One-cycle load-use stalls, as discussed in class

The final results (registers and memory) of your cycle-accurate simulator for a given program must match those of a correct ISA-level simulator from Project 1. To provide you with a starting point/reference for your cycle-accurate simulator, the source code for the solution of Project 1 has been posted on Canvas.

The instructions that must be supported by your simulator are identical to those that were required for Project 1. For details, please see the annotated MIPS ISA PDF in the Project 3 assignment on Canvas.

The instruction decode portion of the simulator should use bit manipulations to discern the instruction type and relevant fields and values from the binary encoding of the instructions.

All of the state (aside from the state of memory, see "Memory" section for details of the provided memory abstraction) should be mimicked by having appropriate program variables and objects. Note that you will also need to keep track of the required state and values in the I and D caches.

The MIPS ISA does not contain a HALT instruction, so as in Project 1, we use the code 0xfeedfeed to signify the end of the code section of a program. Once the 0xfeedfeed instruction completes its IF stage, instruction fetch should stop fetching instructions and insert nops into the pipeline following the 0xfeedfeed, unless an exception occurs. You should halt execution when the 0xfeedfeed has completed its WB stage. For further details on the requirements for starting and ending execution as well as executing for a specific number of cycles, please see the "Execution Flow" and "Exceptions" sections.

We have provided you with an initial test case, but you should certainly do more. You may find it helpful to write test cases and compare their output from the ISA-level simulator to that from your cycle-accurate simulator. Remember – your cycle-accurate simulator should generate results identical to a correct ISA-level simulator.

Caches

You must implement one level of cache in your simulator for this project, with separate I and D caches. Your caches must be write-back, write-allocate, **blocking** caches. Your work must be capable of simulating a variety of cache and block sizes. Furthermore, they must be capable of simulating either a direct-mapped or 2-way set-associative cache (LRU replacement policy).

The information regarding the cache configuration you must implement on a given run of the simulator is provided in CacheConfig structs, which are defined in CacheConfig.h.

You may assume that:

- the block size will always be a power of 2 (in bytes)
- the cache size will always be a multiple of block size * number of ways
- the number of sets in the cache will always be a power of 2

You are required to keep track of the hits and misses in each cache through the execution. You may do this in any way you choose, but you must provide the final tally of hits and misses to the printSimStats(..) function in a SimulationStats struct.

A data cache access is serviced during the MEM phase of the pipeline, while an instruction cache access is serviced during the IF phase. The additional latency of a cache miss (i.e. the miss penalty) is a parameterized number provided by the CacheConfig struct for a given cache. A reasonable starting point for debugging would be to set the miss penalty to 5 cycles. In a typical processor today, the L1 I and D caches would merge their miss streams into a set of requests for a unified on-chip L2. Because we are not asking you to simulate multiple levels of caches, you may assume that main memory can accept up to two requests in any cycle (one from the I cache and one from the D cache), as well as a writeback from the D cache in the same cycle. Furthermore, you should assume that writebacks do not incur any additional latency. Thus, if you miss on an address and have to evict a dirty cache line to make space for it in the cache, you should not add any extra latency for the writeback – only the miss latency of the miss.

As you are implementing separate I and D caches, you may assume that your simulator will never be tested with self-modifying code. In other words, instructions will never be modified by data operations. You may also assume that your caches will never need to service unaligned accesses.

Finally, in the event of a cache miss, you should stall the **entire** processor until the data is available, whether instructions in a given stage depend on the stall or not. So for instance, when there is an I-cache miss in IF, all the other stages in the pipeline are also stalled until the I-cache provides the data, even though the other stages may not need the I-cache miss to complete in order to generate correct results.

Execution Flow

In Project 1, you were responsible for writing the simulator functionality as well as setup and teardown – in other words, you wrote both the simulation functionality and the main() function that called into that functionality. This setup does not allow automatic graders (or you) to stop the simulator part of the way through the execution and dump the current state, and so is not conducive to testing the correctness of a pipelined implementation. As such, for Project 3, the execution flow is somewhat different.

The major difference in execution flow is that you do not write main() in the cycle_sim.cpp that you turn in. Instead, you are required to implement four functions (initSimulator, runCycles, runTillHalt, and finalizeSimulation) that allow us to initialize, run, and finalize your simulation in a variety of different ways. The code for main() (which calls these functions to accomplish the simulation) is contained in a “ driver” C++ file, which is provided by us for the test cases you will be graded on. We provide an example driver file to you for the load_use.asm test case. You will want to write your own drivers to test your code across a variety of cache configurations and at various points in a program's execution.

The driver is responsible for the following:

-Creating a MemoryStore and reading the contents of a provided binary file into it
-Setting the parameters of CacheConfig objects provided to initSimulator(..) according to the caches to be tested
-Calling into runCycles(..) and/or runTillHalt(..) to execute the simulation in chunks or in one go
-Calling finalizeSimulation() to generate register and memory output, as well as statistics

We will use different drivers for different test cases (and possibly test a given test case with multiple drivers to check functionality across different cache organizations). You should write your own drivers when testing to check different scenarios.

The driver organization allows simulation to be stopped at an arbitrary cycle and statistics to be printed out. We provide you with a function that prints out the provided state of the pipeline at a given cycle, which you are required to call at certain points with a proper parameter (see below). This function is

dumpPipeState(PipeState & state)

The parameters to the function provide the cycle to be printed out as well as the five instructions currently in the pipeline (in binary format) (through the PipeState struct). The format of the output is similar to the example output below:

```
Cycle: 23
-----------------------------------------------------------------
lw $t4, 8($t1) | nop | addi $s0, $s3, 0x4 | lui $t0, 0x4 | nop
-----------------------------------------------------------------
```

The dumpPipeState function appends this output to the pipe_state.out file in the directory the simulator is called in. (The pipe_state.out file should be initialized to an empty file by a driver before calling into the simulator for the first time, to erase any old content. All drivers we use will obey this requirement.)

If no instructions have been loaded into a stage of the pipeline yet (for example, when the I-cache is fetching instructions from main memory at the start of simulation), then you should report a nop (sll $zero, $zero, 0) as being present in that stage. Similarly, as the halt instruction passes through the pipeline, you should report nops following it through the pipeline. Furthermore, if you have an unknown instruction in IF (for instance, if you have an I-cache miss when fetching the next instruction), use the value 0xdeefdeef for the instruction in IF, which will print out as "UNKNOWN".

Note the following points about the output of dumpPipeState:

-Immediates and jump address fields of instructions are displayed as unsigned hex numbers, apart from offsets for load and store operations, which are displayed as positive or negative decimal numbers.

-Branches and jumps do not show the actual computed branch address or jump address, but the raw value in the "immediate" or "address" field of the branch or jump.

The four functions you are required to implement in your simulator as hooks for the driver are explained below. Note that these should not be the only functions in your simulator. Good software design involves structuring your code for reuse and clarity.

int initSimulator(CacheConfig & icConfig, CacheConfig & dcConfig, MemoryStore *mainMem)

When called, this function should set up and initialize all state required by your simulator, including an I-cache and a D-cache configured as specified in icConfig and dcConfig respectively (see the "Caches" section), but NOT begin execution. The main memory that you should use is provided as a pointer to a MemoryStore. Unlike Project 1, you are not responsible for reading a binary file representing the program into the MemoryStore. This is taken care of by the driver (as stated above). We do not pay attention to the return value of this function in our grading. The signature allows a return value so that you can return success/failure codes to aid in your debugging if you wish.

int runCycles(uint32_t cycles)

When called, this function should run your simulation for the number of cycles specified in the parameter, and call the dumpPipeState function with the instructions present in the pipeline during the **last** cycle of the run. So for example, if runCycles(40) is called at the start of simulation, the simulator should run for 40 cycles (i.e. cycles 0 through 39), then call dumpPipeState with the instructions present in the pipeline during cycle 39, and then return. Note that being able to check the state of your simulator after each cycle is very useful from a debugging perspective.

If the halt instruction 0xfeedfeed completes its writeback stage before the simulator has run for the specified number of cycles, your simulator should call dumpPipeState after the cycle in which the halt instruction finishes its writeback stage (so the halt instruction should be in WB in the dumped pipe state), and then return from runCycles(..).

If the specified number of cycles were simulated, the function should return 0. If the end of the program was reached prior to the total number of cycles being simulated, the function should return 1.

int runTillHalt()

This function is similar to runCycles(..) above, except that the function simulates execution until the halt instruction has reached its writeback stage. You must call dumpPipeState after the cycle in which the halt instruction finishes its writeback stage before returning from this function.

As in initSimulator, we do not pay attention to the return value of this function, but allow for return values to make it easier for you to debug your simulator.

int finalizeSimulator()

When called, this function should do the following:

- Record the number of cycles simulated, I and D cache hits and misses in a SimulationStats struct, and call printSimStats(..) with this struct. The printSimStats function prints the provided information in a specific format to a sim_stats.out file in the directory the simulator is called in.
- Write back all dirty values in the data cache to main memory. This final draining of the data cache should not be simulated cycle-accurately. It is only necessary to ensure that your dumped memory state is identical to that which would be produced by an ISA-level simulator.
- Call dumpRegisterState with an initialized RegisterInfo struct (as you did at the end of simulation in Project 1).
- Call dumpMemoryState with the MemoryStore representing main memory. As in Project 1, the dumpMemoryState function will dump the contents of a region of memory that will **vary** on a **test by test basis**. It is your responsibility to ensure that all of memory is appropriately updated by the running of your program, as you will not know which part of memory we will be examining for correctness.

You have a large amount of freedom with regard to your implementation of the pipeline and caches. You may use a C-style implementation or a C++ object-oriented implementation, as long as you correctly implement the requirements, including the four functions above. One strategy might be to develop the pipeline and cache simulators separately, and then hook them together. For example, you could build a "free-standing" cache simulator and test it with a known reference trace as input, before you get into the complexities of considering its interactions with pipelined execution. Likewise, you could do most of the pipelined simulation assuming an idealized memory hierarchy (all hits) as long as you are ready to deal later with the issues of plugging in a non-idealized I and D cache module.

<u>Other Implementation Notes:</u>

-For consistency, initialize all registers to have a value of 0 upon startup. (The MemoryStore already internally initializes all memory locations to have a value of 0 upon startup.)
- Once the 0xfeedfeed instruction completes its IF stage, instruction fetch should stop fetching instructions and insert nops into the pipeline following the 0xfeedfeed, unless an exception occurs. You should halt execution when the 0xfeedfeed has completed its WB stage.
-The jal instruction modifies the PC at the end of its ID stage, but only updates the return address register when it reaches the WB stage.
-Dependencies on the zero register do not cause stalls.

<u>**Simulator Compilation**</u>

To compile your simulator, you will need to link your code with a driver (see the "Execution Flow" section) and the implementation of the memory abstraction and utility functions provided in UtilityFunctions.o as follows (assuming your file is called cycle_sim.cpp, as the submission requires):

```
g++ -o cycle_sim cycle_sim.cpp driver.cpp UtilityFunctions.o
```

where driver.cpp is the driver you want to use. So for example, if you want to compile your simulator with the provided example driver, you can do the following (assuming you're in the src/ directory of the provided tarball):

```
g++ -o cycle_sim cycle_sim.cpp ../test/example_driver.cpp UtilityFunctions.o
```

<u>Provided Files:</u>

To allow you to concentrate on building the core of the simulator, we provide you with a number of files containing abstractions, utilities, and utility functions. These files build on those provided for Project 1, and are contained in the project tarball on Canvas. The following is a list of the relevant files ordered by directory structure:

```
bin/
        mips-linux-gnu-as
        mips-linux-gnu-objcopy
        mips-linux-gnu-objdump
src/
        EndianHelpers.h
        example.cpp
        MemoryStore.h
        RegisterInfo.h
        DriverFunctions.h
        CacheConfig.h
        UtilityFunctions.o
test/
        load_use.asm
        load_use_mem_state.out
        load_use_reg_state.out
        example_driver.cpp
        load_use_pipe_state.out
        load_use_sim_stats.out
```

Below is a short description of each of the above files.

mips-linux-gnu-as – MIPS assembler. Converts MIPS text assembly to binary ELF files.

mips-linux-gnu-objcopy – MIPS object file copier/translator. Used to convert ELF files to flat binary files that are read by your simulator.

mips-linux-gnu-objdump – MIPS object file disassembler. Used to inspect the instructions in a binary ELF file.

EndianHelpers.h – Signatures of the functions to convert unsigned integers from little-endian to big-endian.

example.cpp – An example C++ file showing use of the memory abstraction.

MemoryStore.h – The interface to the memory abstraction.

RegisterInfo.h – Signature of the register file dump function and definition of the RegisterInfo struct passed to it as an argument.

DriverFunctions.h – Signatures of the initSimulator, runCycles, runTillHalt, and finalizeSimulator functions which you must implement (see the "Execution Flow" section). Also contains the definitions of the PipeState and SimulationStats structs, which you must fill and provide to the dumpPipeState(..) and printSimStats(..) functions (whose signatures are also in this file) as outlined in the "Execution Flow" section.

CacheConfig.h – Struct that allows the configuration of a cache (size, block size, associativity, miss latency) to be specified.

UtilityFunctions.o – A binary file containing the implementations of the memory abstraction and other utility functions. This file is not intended to be human-readable.

load_use.asm – An example MIPS assembly test program.

load_use_mem_state.out – The memory state dump of running the compiled version of load_use.asm.

load_use_reg_state.out – The register state dump of running the compiled version of load_use.asm.

example_driver.cpp – A driver (see the "Execution Flow" section) which can be used to run load_use.asm, dumping its pipeline state at one point in its execution.

load_use_pipe_state.out – The register state dump of running the compiled version of load_use.asm with the driver example_driver.cpp.

load_use_sim_stats.out – The overall simulation statistics from running the compiled version of load_use.asm with a simulator and the configuration of the driver file example_driver.cpp.

Exceptions

Your simulator must support handling two types of exceptions: arithmetic overflow and illegal instructions. You are NOT required to implement the MIPS exception registers like the EPC. The only thing your simulator must do on an exception is to jump to the exception address and begin executing the instructions at that address. In MIPS, the exception address is 0x80000180, but the address space for your simulator is only 0x10000 bytes large, so we require you to jump to address 0x8000 instead.

When an exception occurs, you should NOT update state for the instruction triggering the exception. So for instance, if an add triggers arithmetic overflow, you should not update the destination register. Simply jump to the exception address and continue execution as stated above.

From an implementation standpoint, illegal instruction exceptions should be detected in ID and overflow exceptions in EX. When an exception is detected, the simulator must squash both the excepting instruction and all instructions after it and replace them with nops. It should also modify the PC to point to the exception address so that the pipeline begins fetching from the exception address the cycle after the exception occurred. Furthermore, all instructions that were before the excepting instruction must complete execution successfully, unless they trigger exceptions themselves. Note that it is possible for two instructions to trigger exceptions in the same cycle.

Note that an exception may occur while the halt instruction 0xfeedfeed is in IF or ID. In such a case, if IF has stopped fetching instructions and is inserting nops into the pipeline, the IF stage should be restarted with the exception address because the halt instruction has been squashed. You should only stop execution when the halt instruction reaches its WB stage.

What to Hand in

Your simulator should be written in C++. The compiler for grading will be g++ .

You only need to submit your implementation of the core simulator, which should be in one file named cycle_sim.cpp. You should not submit any drivers that you may have written for testing.

Please use the Canvas site to turn in your cycle_sim.cpp file.

Your cycle_sim.cpp file must compile properly on the Nobel cluster using:

```
g++ -o cycle_sim cycle_sim.cpp driver.cpp UtilityFunctions.o
```

where driver.cpp is a valid driver file as outlined in the “Execution Flow” section.

If we cannot compile your source file, we cannot grade your assignment and you will have to resubmit; the delay will be taken as late days.

Grading

Grading will consist mainly of running test MIPS binaries (including but not limited to any sample binaries we give you) through your simulator. These test binaries will thoroughly test the operation of instructions. Particular attention will be paid to “corner cases”.
Correctness will be determined both by the final register values, memory results, and statistics of the simulator’s program execution as well as by snapshots of pipeline state at various cycles during the execution. This allows you to get partial credit even if your final results are not completely correct. You will get back a “score sheet” indicating what instructions had problems and what kinds of problems. This represents 75 points of your grade.

We will also look at your source code to determine whether instructions are implemented correctly (i.e. we will not rely solely on test cases). Code which is difficult for us to understand (i.e. uncommented or incorrectly commented) will lose some points. Clarity of code and proper commenting represents 15 points of your grade.

Finally, efficiency matters. Excessively slow or needlessly inefficient simulators will be penalized. This represents 10 points of your grade.

Test case structure (***)

Test cases for this project are MIPS assembly files consisting entirely of a single .text section. Any data values that must have specific values for the program to work must be manually listed at the end of the file (after 0xfeedfeed) using .word directives. For example, consider the following assembly program:

```
 la $t1, 20
 la $t2, 24
 lw $t3, 0($t1)
 sw $t3, 0($t2)
.word 0xfeedfeed
.word 0xac
.word 0xdb
```

Each instruction in the above program is 32 bits = 4 bytes. Thus, the last instruction is located at bytes 12-15.

A ".word" directive fills the corresponding location in the binary file with the raw data provided to the directive. For instance, the .word directive for 0xfeedfeed is at address 16 = 0x10, so the directive causes bytes 16-19 to get filled with 0xfeedfeed. Likewise, bytes 20-23 are filled with 0xac, and bytes 24-27 are filled with 0xdb. (This can vary slightly if the MIPS assembler adds a word or two of padding – see the "Compilation" section.)

Thus, when the third instruction reads from address 20, it will read the value 0xac (since no other instruction has written to it beforehand). Likewise, when the fourth instruction stores to address 24, it will overwrite the value of 0xdb. Note that it is quite possible for a program to access memory locations in its execution other than those locations initialized by .word directives.

At some point, you may also need the .align directive in order to skip bytes till you reach a required alignment. For instance, if the previous lines of the file reach address 0xa, then a ".align 4" directive will skip bytes (filling them with 0) until it reaches an address aligned for 4 bytes (in this case, address 0xc).

Test Case Compilation (***)

To compile test cases, you should use a combination of mips-linux-gnu -as and mips-linux-gnu-objcopy. For example, to compile the example test case (assuming you have the provided binaries in your PATH), run the following commands:

```
mips-linux-gnu-as -march=mips32 fib.asm -o fib.elf
mips-linux-gnu-objcopy fib.elf -j .text -O binary fib.bin
```

The first command assembles the text assembly file into a binary ELF file. This ELF file then needs to be translated into a flat binary file that only contains the code and any .word directives you may have added. This is done by calling mips-linux-gnu-objcopy on the .text section of the ELF file as shown in the second command. The fib.bin file created by the second command should be the input to your simulator.

We suggest you create a script or Makefile to automate the running of the above two commands and make it easy for you to compile multiple test cases.

The MIPS assembler may add an extra word or two of zero padding at certain points in your code, after the code and before your .word directives, and after your .word directives as well. Make sure any offsets you use to read/write values to the .word directive locations take this padding into account.

To inspect the code in an ELF file, you can use the mips-linux-gnu-objdump utility. For example, running the following command

```
mips-linux-gnu-objdump -D -j .text fib.elf
```

disassembles the .text section of the ELF file, allowing you to see the instructions in it. This can be useful to see what, if any, extra padding the assembler has added when assembling your file.

If you want to examine the contents of your flat binary file, you can use hexdump. For instance, running

```
hexdump -e '"%08_ax:" 4/1 "%02x" "\n"' fib.bin
```

will print out the contents of the binary file fib.bin in 4-byte chunks with offsets on the left, as shown below:

```
00000000:24080044
00000004:240d0074
00000008:8dad0000
0000000c:240a0001
…
```

Memory Abstraction (***)

We provide a memory abstraction for the ISA simulator you are to implement for Project 1 so that you do not need to implement memory on your own. You must use this abstraction to model memory in your simulator.

The memory abstraction is 64 KB (0x10000 bytes) large. It has three main functions (getMemValue, setMemValue, printMemory). Each of these functions returns 0 on success and a nonzero value on failure (an error message is also printed on failure).

The getMemValue and setMemValue functions take in a memory address, value, and size (which can be byte size (8 bits), half word size (16 bits), or word size (32 bits)). The getMemValue function returns a value from memory by reference through its "value" parameter, since the function's return value is the status of the operation (0 on success and nonzero on failure, as mentioned above). The printMemory function takes in a start and end address and prints the values of memory from the start address to the end address in 32-bit (4 byte) chunks, with five such chunks (20 bytes) per line. For example, the following three lines of printMemory(..) output each show the value of 20 bytes of memory, starting from addresses 0x0, 0x14, and 0x28 respectively.

```
0x00000000: 0x00000000 0x00000000 0x00000000 0x00000000 0xff000000
0x00000014: 0x00000000 0x0000c000 0x00000000 0x00000000 0x00000000
0x00000028: 0x00000000 0x00000000 0x00000000 0x0a000000 0x00000000
```

If the address range to print is not strictly divisible by 32, the value of a few bytes beyond the end address will also be printed at the end of the output.

Computers (***)

We have tested the files for this project on the Nobel cluster at Princeton OIT. That is where we recommend that you work on the project. If you choose to try to do this elsewhere (e.g. your own computer) then it is your responsibility to ensure that it will work on Nobel when we grade your submission after you hand it in. For access to Nobel start with this link: https://researchcomputing.princeton.edu/systems/nobel

A note on Endianness: (***)

Endianness refers to the order in which bytes are stored in memory with respect to reading and writing multiple-byte words from memory. Big-endian processors use the leftmost or "big end" byte as the word address, while little-endian processors use the rightmost or "little end" byte as the word address. Thus, for example, if storing the value 0xAABBCCDD at address 0, big-endian and little-endian processors would store the value as shown in the following table:

Address	Big-endian	Little-endian
0x0	0xAA	0xDD
0x1	0xBB	0xCC
0x2	0xCC	0xBB
0x3	0xDD	0xAA

MIPS is a big-endian architecture, while x86 processors (such as the ones on the Nobel cluster and most laptops) are little-endian. So, in this assignment you are simulating a big-endian machine by executing the simulator on a little-endian machine. Thus, if on an x86 machine, you read a 32-bit instruction from one of your binary test programs (which are generated in big-endian format), the bytes will be reversed because the underlying x86 machine is little-endian. To assist you with changing the read data back to big-endian format, we provide two functions (signatures in EndianHelpers.h) to convert 16-bit and 32-bit integers from little-endian to big-endian:

```
extern uint32_t ConvertWordToBigEndian(uint32_t value); extern
uint16_t ConvertHalfWordToBigEndian(uint16_t value);
```

You do not need to worry about endianness when reading data from or writing data to the memory abstraction. Any endianness issues that arise with the memory abstraction are handled internally by the provided code.

Integer Types (***)

We recommend you use the types for unsigned integers defined in inttypes.h (such as uint32_t, uint16_t, etc) to represent quantities so as to avoid some issues related to overflow and sign extension. The types in inttypes.h describe both the type and size of the integer in question, which is very useful when dealing with code that requires bit manipulation. For example, **u**int**32**_t is an **u**nsigned **32**-bit integer, and **u**int16_t is an **u**nsigned **16**-bit integer.