ELEC50010 Instr. Arch + Comp. : CPU Coursework

This is the coursework for the 2020-21 year of the IA+C coursework.

The submission timings are:

- Mon Nov 23rd : Coursework "officially" starts (it's a 1 month coursework).
- Mon Dec 7th 22:00 : Optional formative feedback point. If you submit your current work in progress, then it will get manually examined, and receive oral formative feedback.
- Wed 16th 22:00 : Optional sanity check point. Some simple scripts will be run on current submissions to check for things like file-names, whether scripts can be executed, and ability to test a CPU that is not your own.
- Mon 21st Dec 22:00 : Final deliverables due.

Revision log

- 2020/08/13 : v0 - Initial draft
- 2020/10/20 : v1.0 - Updated with harvard and bus to provide simpler learning curve.
- 2020/11/16 : v1.1 - Minor tweaks based on lab results.
- 2020/11/20 : v1.2 - Added missing environment/standards part.

Overall goals

Your overall goals are to develop a working synthesisable MIPS-compatible CPU. This CPU will interface with the world using a memory-mapped bus, which gives it access to memory and other peripherals.

The goal of this coursework is not to get a single circuit working in a single piece of hardware. Instead it is to develop a piece of IP which could be sold and distributed to many clients, allowing them to integrate you CPU into any number of products. As a consequence the emphasis is on producing a production quality CPU with a robust testing process - you should deliver something that you expect to work on any FPGA or ASIC, rather than something that just works on a single device.

The emphasis on creating a "real" CPU makes this a more complex task than implementing a toy CPU with lots of extra debug hooks. In particular, the emphasis on memory-based input/output is very realistic, but means you need to be very methodical and analytical in the way you develop both your CPU *and* your test-bench and test-cases.

Coursework deliverables

Your coursework deliverables consist of the following:

1. `rtl/mips_cpu_bus.v` or `rtl/mips_cpu_harvard.v` : An implementation of a MIPS CPU which meets the pre-specified template for signal names and interface timings. You may also include other verilog modules in files of the form `rtl/mips_cpu/*.v` and/or `rtl/mips_cpu_*.v`. If you include both a `bus`

and a `harvard` verilog file it will be assumed that you want the `bus` version to be assessed. Any files not matching these patterns will be ignored.

2. `test/test_mips_cpu_bus.sh` or `rtl/test_mips_cpu_harvard.sh` : A test-bench for any CPU meeting the given interface. This will act as a test-bench for your own CPU, but should also aim to check whether any other CPU works as well. You can include both scripts, but only the one corresponding to your submitted CPU (bus or harvard) will be evaluated.

3. `docs/mips_data_sheet.pdf` : A data-sheet for your CPU, consisting of at most 4 A4 pages. This data-sheet should cover:

    - The overall architecture of your CPU.
    - At least one diagram of your CPU's architecture.
    - Design decisions taken when implementing the CPU.
    - The approach taken to testing CPUs.
    - At least one diagram or flow-chart describing your testing flow or approach.
    - Area and timing summary for the "Cyclone IV E 'Auto'" variant in Quartus (same as used in the EE1 "CPU" project).

4. Peer feedback : individual submission by each group member to provide peer feedback on your team members, submitted via Microsoft Forms.

Assessment

The coursework mark comes from the following components:

- Functionality (40%) : does the CPU work?

    - This is assessed purely based on whether instructions are functionally correct.
    - The only method used to assess correctness is to look at the changes to RAM that the CPU performs, and/or the final value of register `v0`.
    - The same set of instructions are tested for both the bus and harvard interfaces, but if the harvard interface is used, then this component is scaled by `0.8`.

- Testbench (30%) : can the test-bench detect whether other CPUs work?

    - This is assessed by telling your test-bench to test other CPUs.
    - The variant of your test-bench (bus versus harvard) assessed will match your CPU.
    - You should expect it to be tested on a "perfect" CPU, as well as selectively broken CPU
    - Your test-bench should not say the perfect CPU fails (false-negative), nor should it say the broken CPU passes (false-positive).

- Data-sheet (30%) : is the architectural and testing approach adequately described?

    - Have the required components been covered?
    - Is it a client-oriented document, rather than oriented at the people who developed the CPU?
    - Does it provide useful information specific to your solution?
    - Does it highlight any clever or important features/decisions?

- Peer-feedback (+-5%) : allocated according to peer feed-back within the group. This will affect the individual mark by up to 5% compared to the group mark.

Submission

Submission will be through a `.tar.gz` submitted via blackboard.

It is up to you to choose/manage source code control through whatever tool or technology you want. You can get access to github pro through the github education programme, but you can use any other service your team prefers - if you want to work out of a shared DropBox then that is up to you.

Note that any git repo should not be public while the assessment is ongoing, in order to avoid any plagiarism concerns. Once the assessment is finished you can make the code available publically.

CPU interface

You have a choice of two different interface styles for your CPU to support:

- **Bus** : A true memory bus based interface, which is directly compatible with industrial IP blocks. This requires instructions and data to be fetched over the same interface, and also allows memory to have variable latency.
- **Harvard** : A simpler interface which provides seperate instruction and data memory interfaces. These interfaces also support combinatorial read paths, and single-cycle write paths.

You need to choose one of these methods for the final submission, but might find it useful to start with harvard and then migrate to bus. Most of the internal control and arithmetic logic can be directly shared between the two approaches, as long as you are taking a disciplined approach to decomposing your design. It is also (intentionally) possible to take a Harvard CPU and wrap it in a module which will transparently adapt it to the Bus interface, which can be another route to a working bus-based CPU.

If you include both a bus and a harvard variant in your submission, then it is assumed that you intend the bus version to be the submitted version.

Because the harvard version simplifies away a number of more real-world constraints, the functionality mark is scaled by 0.8 compared to the same functionality in a bus CPU. The other components (testing and documentation) are unaffected by whether harvard or bus is used.

Shared interface aspects

Both interfaces share the following common signals:

```
module mips_cpu_...(
    input logic clk,
    input logic reset,
    output logic active,
    output logic[31:0] register_v0,
```

All signals are synchronous to `clk`, including `reset`.

The `reset` signal must be held high for at least 1 cycle to reset the CPU. This is a level-sensitive reset, which is synchronous to the clock.

The `active` signal should be driven high when `reset` is asserted, and remain high until the CPU halts. Once the CPU has halted (for any reason) the `active` signal should be sent low.

If the CPU has completed execution (i.e. it has been reset and then `active` has been sent low), then `register_v0` should contain the final value of register `$v0` (register index 2) from the register file. This is purely to make your test-benches easier, and is not something typically included in a CPU IP core.

The CPU does not have any support for interrupts or other input/output signals. The only way of communicating is via memory bus transactions, the `active` signal, and the `register_v0` signal.

Bus based interface

The CPU uses a single Avalon compatible memory-mapped interface to interact with memory. Your CPU acts as a bus controller, and issues read and write transactions in order to change memory contents. However, it is important to remember that your CPU should be completely independent of the memory itself. The memory may be a genuine hardware RAM implemented using BRAM or DDR, or it could be a completely virtual memory provided by a test-bench.

The bus-based CPU interface has the following signals:

```
module mips_cpu_bus(
    /* Standard signals */
    input logic clk,
    input logic reset,
    output logic active,
    output logic[31:0] register_v0,

    /* Avalon memory mapped bus controller (master) */
    output logic[31:0] address,
    output logic write,
    output logic read,
    input logic waitrequest,
    output logic[31:0] writedata,
    output logic[3:0] byteenable,
    input logic[31:0] readdata
);
```

Avalon is a clock synchronous protocol, so `readdata` will not become available until the cycle following the read request. The signal `waitrequest` is used to indicate a stall cycle, which means that the read or write request cannot complete in the current cycle, and so must be continued in the next cycle. See section 3.5.1 and Figure 7 of the Avalon spec for more info.

Harvard interface

Everything is easier if there are two seperate instruction and data memory buses, *and* the memory interfaces support combinatorial (zero-cycle) reads. Taken together, these allow you to build the simple single-cycle data-path developed during the first week of lectures. However, this is also very unrealistic, as most CPUs (ignoring embedded micro-controllers) only have access to a single memory bus, and have to deal with variable memory stall cycles. Unfortunately, such a single memory bus design is complex, and represents a difficult starting point, as there are two main ways of implementing it - either you need to effectively implement an instruction and data cache plus appropriate stall logic, or you need to implement a more complex multi-cycle finite-state machine to execute the instructions.

The harvard interface here allows you to choose to use the simpler interface, which removes a lot of that complexity. The interface is as follows:

```
module mips_cpu_harvard(
    /* Standard signals */
    input logic     clk,
    input logic     reset,
    output logic    active,
    output logic [31:0] register_v0,

    /* New clock enable. See below. */
    input logic     clk_enable,

    /* Combinatorial read access to instructions */
    output logic[31:0]  instr_address,
    input logic[31:0]   instr_readdata,

    /* Combinatorial read and single-cycle write access to instructions */
    output logic[31:0]  data_address,
    output logic        data_write,
    output logic        data_read,
    output logic[31:0]  data_writedata,
    input logic[31:0]  data_readdata
);
```

The signals prefixed `instr_` implement the instruction bus, while those prefixed `data_` implement the data bus.

The new signal `clk_enable` supplies a clock enable, and should be used to determine whether to update your flips-flops in a given cycle. The general pattern for updating registers with a clock enable is:

```
always_ff @(posedge clk) begin
    if (reset) then begin
        /* Do reset logic */
        my_ff <= ... ;
    end
    else if(clk_enable) then
        /* Perform clock update */
        m_ff <= ... ;
```

```
    end
end
```

The interface semantics guarantee that if `clk_enable` is high then the following conditions all hold:

1. `instr_readdata == MEMORY[instr_address]`
2. `data_read==1 -> data_readdata == MEMORY[data_readdata]`
3. `data_write==1 -> MEMORY[data_address] == instr_writedata`

Note that `A -> B` means logical implication, so "if A then B".

You should still combinatorially drive all other output signals (e.g. `data_read`, `data_write`, `instr_addr`) during cycles where `clk_enable==0`, as the `clk_enable` signal is in part derived from those signals.

The Harvard interface does not provide access to byte enables, which means that partial store instructions (e.g. `sh`, `sb` and `swl`) are quite complicated. If you are getting to that level it is probably better to switch to the bus based interface.

Constraints on the interface are:

- `! (data_read & data_write)` : You cannot read and write in the same cycle.
- `data_write==1 -> instr_addr != data_addr` : You cannot modify the instruction currently begin read (note the comment later on self-modifying code).

Reset Behaviour

During reset (i.e. while the `reset` signal is high), the CPU should not initiate any memory transactions, as the memory may also be resetting at the same time.

The `reset` signal may be held high for more than one cycle, as other IP cores or devices could be driven by the same reset and need more than one cycle to reset.

It is not specified what the CPU should do during reset, but the *effect* of reset should be that:

- All ISA-visible MIPS data registers are set to zero.
- The next instruction to be executed post-reset should be at address `0xBFC00000`.

The address `0xBFC00000` is the reset vector of the CPU, and is the conventional reset vector for a "real" MIPS CPUs. The slightly odd address is to place it at the start of the 4MB region `[0xBFC00000,0xC0000000)`.

CPU Halt

Often CPUs do not "finish" in a meaningful way, and the expectation is that once a CPU powers on there will always be work for it to do. However, here we want a definitive end point for CPU execution, in order to make testing more tractable - we need to know when the CPU being tested has finished, so that we can look at how it has modified memory. To make things easier when learning, it is also very useful to have visibility on some internal CPU state, as doing everything via memory assumes you already have working memory instructions.

To make testing easier we include the `active` flag and the `register_v0` flag. The dual purpose of these signals is:

1. To detect when the CPU has finished executing instructions.
2. To allow a single 32-bit value to be passed from inside the CPU to the top-level module, without requiring any memory transactions.

The CPU is considered to halt when it executes the instruction at address 0. This behaviour is specific to this coursework specification, and not a general property of the MIPS ISA, ABI, or commercial IP cores.

The reason for this choice is intimately related to the reset conditions and MIPS O32 ABI; in particular, this choice exploits the following existing requirements:

- For the reset, we require that all registers (including the PC) are set to 0.
- The MIPS ABI also specifies that integer return values from functions are placed in register $v0, which is defined to be register 2.
- The MIPS ABI also specifies that the return address for a function is stored in register $ra, which is defined to be register 31.

This means that the following function:

```
int f(){
    return 23;
}
```

can be assembled into the following assembly:

```
f:  li $2, 23   # Load 23 into register $2
    jr $31      # Jump to the address in $31 (which will be zero)
    nop
```

Note that a compiler is likely to exploit the delay slot, and so will probably produce the following shorter code which exploits the delay slot:

```
f:  jr $31      # Jump to the address in $31 (which will be zero)
    li $2, 23   # Load 23 into register $2
```

If this rearranged code looks confusing, then look carefully at what the ISA says about advancing the PC and branches.

CPU Performance

The goal of the exercise is to deliver a functionally correct CPU, so performance is a secondary concern. However, your CPU should not exceed a worst-case CPI of 36 (ignoring memory stall cycles).

Instruction Set

The target instruction-set is 32-bit little-endian MIPS1, as defined by the MIPS ISA Specification (Revision 3.2).

The instructions to be tested are:

Code	Meaning
ADDIU	Add immediate unsigned (no overflow)
ADDU	Add unsigned (no overflow)
AND	Bitwise and
ANDI	Bitwise and immediate
BEQ	Branch on equal
BGEZ	Branch on greater than or equal to zero
BGEZAL	Branch on non-negative (>=0) and link
BGTZ	Branch on greater than zero
BLEZ	Branch on less than or equal to zero
BLTZ	Branch on less than zero
BLTZAL	Branch on less than zero and link
BNE	Branch on not equal
DIV	Divide
DIVU	Divide unsigned
J	Jump
JALR	Jump and link register
JAL	Jump and link
JR	Jump register
LB	Load byte
LBU	Load byte unsigned
LH	Load half-word
LHU	Load half-word unsigned
LUI	Load upper immediate
LW	Load word
LWL	Load word left
LWR	Load word right
MTHI	Move to HI
MTLO	Move to LO

Code	Meaning
MULT	Multiply
MULTU	Multiply unsigned
OR	Bitwise or
ORI	Bitwise or immediate
SB	Store byte
SH	Store half-word
SLL	Shift left logical
SLLV	Shift left logical variable
SLT	Set on less than (signed)
SLTI	Set on less than immediate (signed)
SLTIU	Set on less than immediate unsigned
SLTU	Set on less than unsigned
SRA	Shift right arithmetic
SRAV	Shift right arithmetic
SRL	Shift right logical
SRLV	Shift right logical variable
SUBU	Subtract unsigned
SW	Store word
XOR	Bitwise exclusive or
XORI	Bitwise exclusive or immediate

It is strongly suggested that you implement the following instructions first: `JR, ADDIU, LW, SW`. This will match the instructions considered in the formative assessment.

Memory Map

Your CPU should not make any explicit assumptions about the location of instructions, data, or peripherals within the address space. It should simply execute the instructions it is given, and perform reads and writes at the addresses implied by the instructions.

There are only two special memory locations:

- `0x00000000` : Attempting to execute address 0 causes the CPU to halt.
- `0xBFC00000` : This is the location at which execution should start after reset.

Whether a particular address maps to RAM, ROM, or something else is entirely down to the top-level circuit outside your CPU. It may be that the top-level is a test-bench which contains small simulated memories, and simply maps transactions to reads and writes of a verilog array. Or the test-bench could emulate only the specific addresses that it expects to be read or written, without tracking the actual memory contents. Alternatively your CPU may have been synthesised into an FPGA, in which case the memories may correspond to a large set of block RAMs, DDR, network adaptors, and anything else your customer decided to attach the CPU to.

Exceptions

Our memory bus has no mechanism for indicating that a particular read or write access failed, in order to keep the interface simple. This means that there is no portable way for you to test how a given processor responds to invalid addresses. The only thing you can do is give it test-cases which will result in it accessing a known sequence or range of addresses, and then check that it does indeed access those addresses. If a CPU-under-test ever accesses an address which is outside that set of known addresses, then you can legitimately claim that it failed the test-case, and halt the test-bench immediately (if you wish). Similarly, if the CPU-under-test does not access an address which you know must be accessed, then it must also have failed. *You are not required to validate the exact sequence of addresses, this is simply talking about what is valid or not to test.*

There is also no defined mechanism to allow CPUs to indicate that an arithmetic exception has occurred (e.g. overflow). As a consequence, the various overflow-checking instructions (`add`, `sub`) etc. are not included in the testable set of instructions. So while you can implement them in your CPU, you should not attempt to execute them in your general test-bench. Note that `gcc` will not generate such instructions by default, so you will not see them if compiling C code to MIPS. *This restriction is quite artificial and only for coursework purposes. There is a well-defined mechanism based on exception handlers that could have been used, and would require no changes to the Verilog interface.*

A CPU is not required to have any specific handling for undefined or out-of-spec instructions. So a correct CPU can take any reasonable default behaviour if it is asked to execute an instruction which is outside the defined set of testable instructions. Note that "reasonable" does not mean "any" - you shouldn't deliberately take destructive actions if an invalid instruction is encountered.

Test-bench

Your test-bench is a bash script called `test/test_mips_cpu_bus.sh` or `test/test_mips_cpu_harvard.sh` that takes a required argument specifying a directory containing an RTL CPU implementation, and an optional argument specifying which instruction to test:

```
test/test_mips_cpu_(bus|harvard).sh [source_directory] [instruction]?
```

Here `source_directory` is the relative or absolute path of a directory containing a verilog CPU, and `instruction` is the lower-case name of a MIPS instruction to test. If no instruction is specified, then all test-cases should be run. Your test-bench may choose to ignore the instruction filter, and just produce all outputs.

The test-bench should print one-line per test-case to stdout, with the each line containing the following components separated by whitespace:

1. Testcase-id : A unique name for the test-case, which can contain any of the characters `a-z`, `A-Z`, `0-9`, `_`, or `-`.
2. Instruction : the instruction being tested, given as the lower-case MIPS instruction name.
3. Status : Either the string "Pass" or "Fail".
4. Comments : The remainder of the line is available for free-from comments or descriptions.

If there are no comments then a trailing comma is not needed. Examples of possible output are:

```
addu_1 addu Pass
addu-2 addu Fail   Test return wrong value
MULTZ    mult    Pass    # Multiply by zero
```

Assuming you are in the root directory of your submission, you could test your CPU `rtl/mips_cpu_bus.v` as follows:

```
$ test/test_mips_cpu_bus.sh rtl
addu_1 addu Pass
addu_2 addu Pass
subu_1 subu Pass
subu_2 subu Pass
```

Restricting it to use the addu instruction:

```
$ test/test_mips_cpu_bus.sh rtl addu
addu_1 addu Pass
addu_2 addu Pass
```

If you were to replace `bus` with `harvard` then it should would instead test the `harvard` implementation.

Your test-bench does not need to implement the instruction filter argument, and can choose to just run all test-cases every time it is run. However, you should be aware that if your test-bench locks up or otherwise aborts on one instruction, then it will appear as if all following instructions were never tested.

The total simulation time for your entire test-bench should not exceed 10 minutes on a typical lap-top.

Your test-bench should never modify anything located in the mips source directory. So it should not create any files in the source directory (e.g. `rtl`), and it definitely should not modify any of the files.

Working and input directory

To keep things simple, you can assume that your test-script will always be called from the base directory of your submission. This just means that your script is always invoked as `test/test_mips_cpu_bus.sh`.

However, you should not assume anything about the directory containing the source MIPS. This could be a sub-directory of your project, or could be at some other relative or absolute path. For example, it might be invoked as:

```
test/test_mips_cpu_bus.sh ../../reference_mips_cpu
```

to get your testbench to execute against a reference CPU. Or it could be invoked as:

```
test/test_mips_cpu_bus.sh /home/dt10/elec50010/cw/marking/team-23/rtl
```

Either way, your test-bench just needs to compile the verilog files included in that

Auxiliary files

Your test-bench can make use of any number of auxiliary files and directories, for example things like testcase inputs, pre-compiled object files, or whatever you like. You should aim to keep the submission as small as possible (e.g. using `.gitignore` files), but there is no penalty for including more than is needed.

Environment and Standards

The verilog should be written to adhere to the sub-set of SystemVerilog 2012 supported by Icarus verilog 11.0. CPUs should be written to assume that verilog files are compiled with `-g 2012`, and test-benches should also provide that flag when compiling.

The test environment should be assumed to be Ubuntu 18.04. Version 11.0 of Icarus verilog is already compiled and installed. Standard base Ubuntu packages will be installed, along with the following packages:

- `build-essential` (g++, make)
- `git`
- `gcc-mips-linux-gnu`
- `qemu-system-mips`
- `python3`
- `cmake`
- `verilator`
- `libboost-dev`
- `parallel`

Clarifying notes

Self-modifying code

No distinction should be made between instruction and data addresses - it is legal to both read a memory address as data and to execute it. For almost all implementations this should happen naturally, and is a corner case that only comes into effect with seperate instruction and data caches.

However, we will require that no address that is executed as an instruction is every modified. This is because we lack any method to tell CPUs that their instruction caches (if they exist) may have been invalidated by data accesses.

How to choose between bus and harvard?

If you think about it, a large amount can be shared between the two as long as you create split things up logically. In terms of test-cases for MIPS instructions, they are going to be the same between the two approaches. It is only the test-bench which is going to have to implement a different interface for the CPU, but the instructions it loads can be the same.

Similarly, in the CPU you should find that all the instruction decode and execute logic is mostly the same. It is only the parts that deal with instruction timing and memory that are different. So you can have a single shared execution core that is used by two variants.