Inf2C Computer Systems

Coursework 2

MIPS Multi-cycle Processor Design

Deadline: Fri 22 Nov 2013, 16:00

Paul Jackson

1 Introduction

The aim of this practical is to increase your familiarity with the structure and operation of a simple computer processor. It asks you to write and submit part of a SystemC program which models the operation of a simple multicycle processor and memory system.

SystemC is a language for describing and simulating hardware designs. It is realised as a library of C++ classes: hardware is described by C++ programs that use these library classes, and simulations are run by compiling and executing these programs.

This course does not teach you SystemC, nor are you advised to learn it by yourselves at this stage. We are merely using it as a simulation engine because it can effectively model concurrent operations, closely mimicking the operation of real hardware. The code that you are required to write will be plain C. The provided code will obviously use SystemC features but you are not required to learn how it works. What you need to understand is how simple computer hardware works, not the details of how to model it using SystemC.

Your answers should be submitted electronically **before 4pm on Friday 22nd November** (see later in this handout for details of how to do this). This is the second and last practical for Inf2C Computer Systems course. It is worth 50% of the coursework mark for Inf2C-CS and 12.5% of the overall course mark. **Please bear in mind the guidelines on plagiarism which are linked to from the Informatics 2nd Year Guide.**

Before attempting this coursework, you **have to go through the lab script called "SystemC Basics"** available from the course's Schedule web page. It provides a tutorial on how to use SystemC and how to view the results of the simulation using the gtkwave waveform viewer.

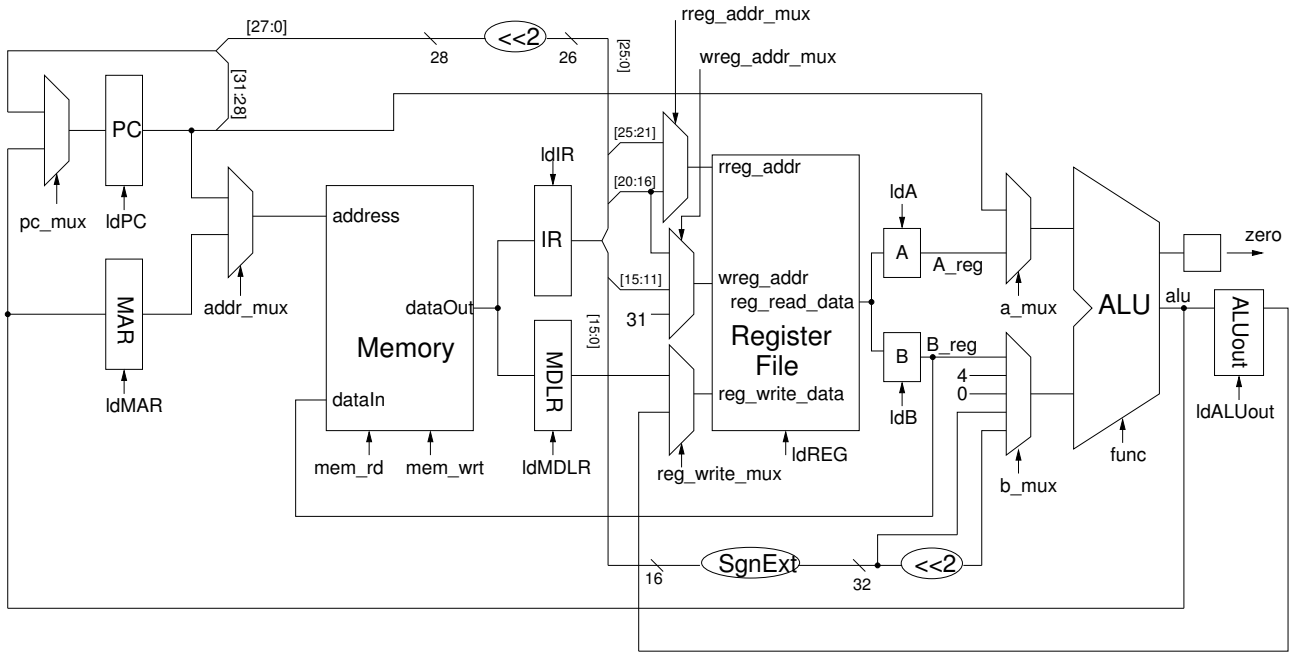


Figure 1: The datapath of the processor used in the practical.

2 The practical

In this practical you complete a SystemC program which models the operation of a simple processor and memory. The processor is based on the MIPS processor, but with a much reduced instruction set. Its datapath is illustrated in figure 1. All wires shown to be unconnected in the figure are actually connected to the control unit. They are not shown to keep the diagram clear. The processor instruction set is described in Appendix B and includes one non-MIPS instruction specially added for this practical, the `halt` instruction, which simply ends the simulation when executed. The `halt` instruction and three further instructions – `addi` (*add immediate*), `lw` (*load word*) and `j` (*jump*) – are already implemented for you in the supplied code.

Your task is to complete the model of the processor's control unit, so that it provides the correct sequence of control signals to the processor datapath to implement fetching and executing the required instructions. You must try to make the processor work as fast as possible, i.e. try to minimise the number of cycles each instruction needs to complete.

Before starting, you will need to make your own copy of the SystemC files. Download the file `cw2-code.tgz` at

```
http://www.inf.ed.ac.uk/teaching/courses/inf2c-cs/coursework/cw2-code.tgz
```

Unpack this tar-ball with

```
tar xzf cw2-code.tgz
```

It unpacks to two top-level directories, `proc` containing the code for the processor, and `systemc-2.2.0` containing the SystemC library code, and a symbolic link `systemc` to the `systemc-2.2.0` directory. Within the `proc` directory, you will find

- `controlUnit.cpp`, `controlUnit.h`, `datapath.cpp`, `datapath.h`, `defines.h`, `main.cpp`, `memory.cpp`, `memory.h`: C++ files describing the processor.

- `control-unit-table.ods`: An Open Office spreadsheet, describing the combinational logic of the control unit.
- `Makefile`: A *make* file containing instructions on how to compile the C++ code. If you are unfamiliar with the `make` utility, enter `man make` at a command prompt to read the first part of the manual page for `make`.
- `test_and`, `test_or`, `test_xor`, `test_addi`, `test_lw`, `test_sw`, `test_beq`, `test_bne`: `test_j`: *Memory test files* containing programs and data for loading into memory at the start of simulations and testing instruction implementations.

When in the `proc` directory, the SystemC model can be compiled with the command `make`, and the compiled simulation code can be run with the command `./proc` *memoryfile*, where *memoryfile* is the name of a memory test file.

3 Control unit

The code for the control unit is described in file `controlUnit.cpp`. As explained in the lectures, the control unit of a multicycle processor is an FSM. The current state of the FSM for this practical is held in variable `cur_state` (actually a field of an instance of the `controlUnit` class defined in `controlUnit.h`).

The control unit code is divided between two main functions

- `controlUnit::ctrl_regs()` which initialises and updates the current state of the machine, and deals with halting the simulation.
- `controlUnit::ctrl_comb()` which simulates the combinational logic of the state machine, computing the next state and outputs, given the current state and inputs.

Your job is to complete the implementation of `controlUnit::ctrl_comb()`. Do not touch `controlUnit::ctrl_regs()`.

The inputs available to the control unit are:

`ir` The contents of the Instruction Register in the datapath, which holds the current instruction being executed. In order to save you the trouble of finding out how to extract various bit fields from `ir` using SystemC, the variables `opcode` and `subfunct` are defined for you at the top of the `ctrl_comb` function. They contain the corresponding bit fields from `ir`, so you will not need to refer to `ir` in your code at all.

`zero` A flag which is `true` if the datapath ALU output at the end of the *previous* cycle was zero, and `false` otherwise. Note in the datapath that the ALU `zero` signal is stored into a flip-flop before it is given to the control unit.

Whereas the next state of the control unit can depend on both the current state and these inputs, the outputs of the control unit must depend only on the current state. This restriction on the derivation of the outputs is also applied to the control unit for the multicycle processor design in the Patterson and Hennessy text-book and the vending machine example presented in class. It is a common restriction, as it ensures the outputs are always valid early in each clock cycle and it can simplify the combinational logic design.

In Appendix A of this handout you will find a complete list of the control signals output by the control unit, together with the valid values each can take, and the effect that each value has on the operation of the datapath or memory. You will need to refer to this list to complete the practical.

As you can see from the provided code, some of these control signals are assigned default values at the start of `controlUnit::ctrl_comb()`, which may then be overwritten. This is not a problem as function `ctrl_comb()` runs uninterrupted and only the final values of the signals are seen by the other blocks.

4 Trying it out

Take a look at the file `test_j`. It contains memory contents to be loaded into memory when the simulation starts. Lines beginning with a `%` contain hexadecimal representations of 32-bit values, which are loaded into memory in consecutive words starting at address 0. All other lines are comments. The file `test_j` contains a small program consisting of three jump instructions and a halt instruction.

Compile the code provided for you, and try `./proc test_j`. A waveform file, called `waves.vcd`, should be created in the same directory, showing the values for all interesting registers and other signals in the system. You can view it using an application called `gtkwave`. With the files `test_j` and `controlUnit.cpp` in front of you, together with the control signal definitions at the end of this handout and their encodings defined in `defines.h`, try out `./proc test_j`. Look at the resulting waveforms, making sure you understand what the simulation is doing and why.

You can also run the `test_addi` and `test_lw` tests for the other already implemented instructions, and inspect the resulting waveforms.

5 What you have to do

Your task is to extend the implementation of the function `ctrl_comb()`, so that it implements all the remaining MIPS instructions described in Appendix B. To do this you will need first to work out a suitable sequence of operations for each instruction, and then add code to implement the instruction.

Do not make changes to any other functions, or your submitted code may not work when we compile and test it with the original remaining functions.

To help you design your code, you should first extend the table provided in the spreadsheet `control-unit-table.ods` with descriptions of further states so that the table fully describes a state machine that controls the execution of each of the instructions you are asked to support. On the left are columns where values for the current state and inputs to the FSM are described, and on the right are columns for describing the corresponding next state and outputs. An `X` for an input value or output value indicates that the value does not matter. A row near the top of the table shows default values for the various outputs. When an actual value entered is the same as the default value for that output, the actual value is enclosed in parentheses to signal this to the reader.

The rest of the rows of the table are organised into groups, 1 group per current state. Working across the table, for each group of rows describing a state, the table specifies

- The state number.
- The instructions and their execution steps for which the state is relevant.
- What should happen on the data-path, both in English and using program-like notation.
- The possible next state transitions, sometimes dependent on the state machine inputs. When there are dependencies, the table indicates the instruction for which each transition might be relevant.
- The outputs to be generated.

Similar examples of FSM truth tables can be found in the course slides and notes on logic design and in the Patterson and Hennessy textbook. The 2nd and 3rd editions of this textbook also have some discussion of designing the FSM for a similar multi-cycle processor design. It is important to realise that the design here *is different* from that presented in the book and in lectures. For example, the Register File here has a single read port, not two as in the book and in the multi-cycle processor lecture. Also, as mentioned earlier, with the design here the `zero` signal fed as input to the control logic is acted on by the control logic in the cycle following the cycle in which it is generated. In the book and in the course slides, it can acted on in the *same* cycle it is generated.

When completing the table, you might find you want to modify the behaviour in states 0 or 1, as these states are common to all instructions, and in the supplied code, only sufficient operations are provided to support the provided example instructions.

Once you have completed the table entries for a given instruction, the coding of that instruction in the `ctrl_comb()` function should be reasonably straightforward – you just are coding the truth table in C. Please add comments to the code to describe what is going on, so the code reader doesn't have to refer back to the truth table spreadsheet to follow what is going on. Writing such comments ought to help you check that you have implemented the each segment of code correctly.

Feel free to optimise your design and your code. For example, if you find that two states of your state machine are identical or near identical, it might be possible to combine them. Also feel free to change default values of outputs. If you do so, update your spreadsheet, so your spreadsheet and code correspond. When deciding on what optimisations to make and when commenting your optimisations, remember that the overriding concern is clarity. For example, you might decide to explicitly set multiplexer values everywhere, never relying on default values. Or, if you do rely on default multiplexer settings, you might still add a comment at each place where you rely on some setting. Either approach would make clear where it matters how the processor datapaths are being routed.

For each of the instructions you are to implement, except `jal` and `jr`, a memory test file is provided. Test your implementations by running simulations with these files. The simulation code is configured to print out the values of the Program Counter and Registers 1-12 when the simulation halts. Read the comments at the start of the test files for the output expected.

You are expected to write memory test files `test_jal` and `test_jr` for the `jal` and `jr` instructions, and to submit these files with the other files you work on. Include appropriate comments in your test files, including a description of the output expected when the test is run.

6 Submitting your work

Generate a PDF version `control-unit-table.pdf` of your `control-unit-table.ods` table. You can do this using the *Export as PDF...* option on the `File` menu in the Libre Office *Calc* tool available on DICE machines. The page formatting for the provided spreadsheet is configured so the table should export in landscape mode on one or two sheets.

Create a tar and gzipped file `cw2-soln.tgz` of your files `controlUnit.cpp`, `test_jal`, `test_jr`, `control-unit-table.ods` and `control-unit-table.pdf` using the command

```
make submission
```

.

Submit your `cw2-soln.tgz` file using the command

```
submit inf2c-cs cw2 cw2-soln.tgz
```

.

7 Marking

Marking will take into account the following:

- *Whether your instruction implementations execute correctly.* We will check this with the help of an *automated script*, running tests similar to those provided. For this reason it is vital that you make sure your submitted code compiles and runs without crashing.

  As well as correctness, we will be checking here the performance of your code, whether your instructions execute in the numbers of cycles we expect them to take or whether they take longer.

- *The quality of your spreadsheet.* Are the instruction steps well described? Has appropriate use been made of `X` values for inputs and outputs? Do sample entries in the table look correct?

- *The quality of your code.* How easy is it to read? How well formatted and commented is it? How well optimised is the code? (Remember, as noted above, we will be looking for *appropriate* optimisations. We do not want to see the shortest possible code. But nor do we ideally want to see near-identical code segments repeated several times.)

- *The quality of your tests.* Do the tests of the `jal` and `jr` instructions well exercise the instructions? Are the tests well commented?

Half the marks will be awarded for the first item, half for the second three items.

Appendix A: Interface between the control unit and the datapath/memory

Listed below is each of the control signals driven by the control unit, controlling the operation of the datapath and memory in each clock cycle. Also given below are the valid values each field can be set to, and the corresponding effects on the operation of the datapath and memory.

Memory controls

`boolean mem_rd`

`true` The value `n` on the Memory Address input is used to address memory, and a memory read operation is started. The 32-bit word at address `n` of memory is output.

`false` Do nothing.

`boolean mem_wrt`

`true` The value `n` on the Memory Address input is used to address memory, and the 32-bit word on the Memory DataIn input is written into memory at address `n`.

`false` Do nothing.

Register controls

`boolean ldPC`

`true` The Program Counter is loaded at the end of this cycle.

`false` Do nothing.

`boolean ldMAR`

`true` The Memory Address Register is loaded at the end of this cycle.

`false` Do nothing.

`boolean ldIR`

`true` The Instruction Register is loaded at the end of this cycle.

`false` Do nothing.

`boolean ldMDLR`

`true` The Memory Data Load Register is loaded from the memory data output at the end of this cycle.

`false` Do nothing.

`boolean ldA`

`true` The A Register is loaded from the register file data output at the end of this cycle.

`false` Do nothing.

`boolean ldB`

`true` The B Register is loaded from the register file data output at the end of this cycle.

`false` Do nothing.

`boolean ldALUout`

`true` The ALU Output Register is loaded from the main ALU output at the end of this cycle.

`false` Do nothing.

`boolean ldReg`

`true` The value on the register file data input (`reg_write_data`) is written into the register selected by `wreg_addr` (see below), at the end of this cycle.

`false` Do nothing.

Multiplexer controls

`byte rreg_addr_mux`

Selects the source of the address of the general register that is read to the register file data output.

`RA_A` The IR bit field 25:21 (Rs) provides the address of the register to be read.

`RA_B` The IR bit field 20:16 (Rt) provides the address of the register to be read.

`byte wreg_addr_mux`

Selects which general register is written from the register file data input (`reg_write_data`) at the end of this cycle, if `ldREG` is `true`.

`WA_RD` The IR bit field 15:11 (Rd) provides the address of the register to be written.

`WA_RT` The IR bit field 20:16 (Rt) provides the address of the register to be written.

`WA_31` Register 31 ($ra) is to be written.

`byte pc_mux`

`PC_ALU` The ALU output is selected by the PC Multiplexer.

`PC_IMM` The concatenation of the 4 most significant bits of the PC (which has already been incremented by 4) with the 26 least significant bits of the IR, shifted left by 2 is selected by the PC Multiplexer.

`byte addr_mux`

`ADDR_PC` The Program Counter output is selected by the Address Multiplexer.

`ADDR_MAR` The Memory Address Register output is selected by the Address Multiplexer.

`byte a_mux`

`A_PC` The Program Counter output is selected by the A Multiplexer.

`A_REG` The A register output `A_reg` is selected by the A Multiplexer.

`byte b_mux`

`B_REG` The B register output `B_reg` is selected by the B Multiplexer.

`B_4` The constant value 4 is selected by the B Multiplexer.

`B_0` The constant value 0 is selected by the B Multiplexer.

`B_IR_16` The least significant 16 bits of the Instruction Register, sign-extended to 32 bits, are selected by the B Multiplexer.

`B_IR_16X4` The least significant 16 bits of the Instruction Register, multiplied by 4 (i.e. shifted left by 2) and sign-extended to 32 bits, are selected by the B Multiplexer.

`byte reg_write_mux`

`RW_ALU_OUT` The ALU Output Register is selected by the `reg_write` Multiplexer.

`RW_MEM` The Memory Data Load Register is selected by the `reg_write` Multiplexer.

ALU control

`byte func`

Controls the output the ALU as a function of the ALU inputs A and B. (Operator symbols used in the notation below have their usual C meanings)

```
ADD alu = A + B
```

```
SUB alu = A - B
```

```
AND alu = A & B
```

`OR alu = A | B`

`XOR alu = A ^ B`

Special fields

There are two special control fields, which do not control the operation of the datapath or memory, which should also be assigned. These are:

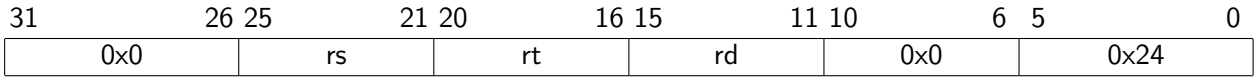
`int next_state` The number of the next state that the control unit should enter when the next rising clock edge arrives.

`boolean halt` If this is set to `true`, the simulation halts.

Appendix B: Instruction set description

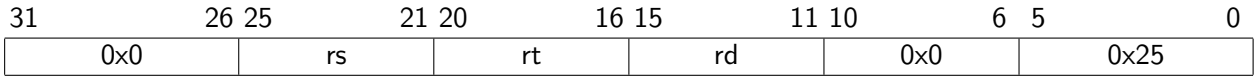
AND

Symbolic representation: `and rd, rs, rt`.
Computes the bit-wise logical AND of the contents of registers `rs` and `rt`, and stores the result in register `rd`.



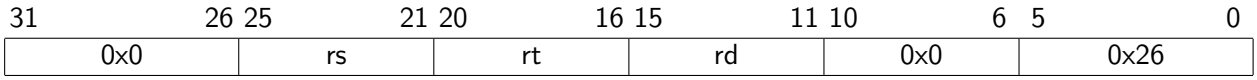
OR

Symbolic representation: `or rd, rs, rt`.
Computes the bit-wise logical OR of the contents of registers `rs` and `rt`, and stores the result in register `rd`.



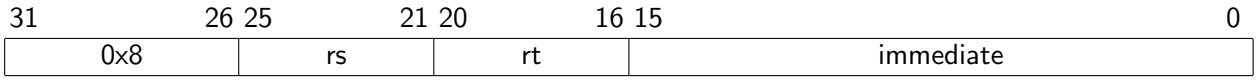
Exclusive OR

Symbolic representation: `xor rd, rs, rt`.
Computes the bit-wise logical XOR of the contents of registers `rs` and `rt`, and stores the result in register `rd`.



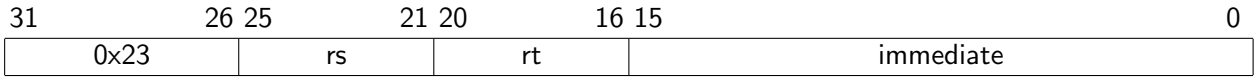
Add immediate

Symbolic representation: `addi rt, rs, n`.
Sign extend the 16-bit 2's complement integer `n`, add to the contents of register `rs`, and store the result in register `rt`.



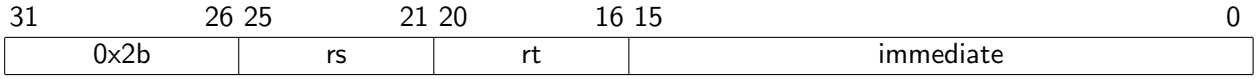
Load word

Symbolic representation: `lw rt, n(rs)`.
Sign extend the 16-bit 2's complement integer `n`, add to the contents of register `rs`, and use the resulting integer to address memory and read the word at that address, and store in register `rt`.



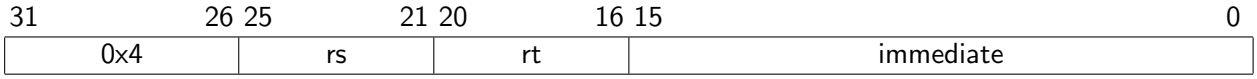
Store word

Symbolic representation: `sw rt, n(rs)`.
Sign extend the 16-bit 2's complement integer `n`, add to the contents of register `rs`, and uses the resulting integer to address memory, storing the word in register `rt` at that address.



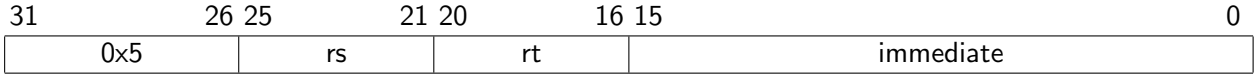
Branch on equal

Symbolic representation: `beq rt, rs, label`.
Compares the contents of registers `rs` and `rt`, and if they are equal branch to the address indicated by `label`. The address is calculated by the machine as follows: multiply the 2's complement immediate (held at bits 15:0 of the instruction) by 4, sign extend to 32-b and add to the address of the following instruction. This is the address written into the PC if the comparison is successful.



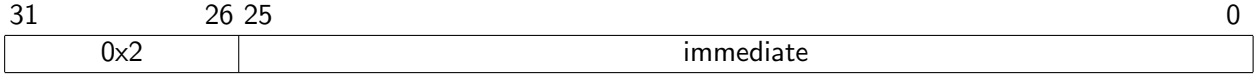
Branch on not equal

Symbolic representation: `bne rt, rs, label`.
Compares the contents of registers `rs` and `rt`, and if they are not equal branch to the address indicated by `label`. The address calculation is the same as for `beq`.



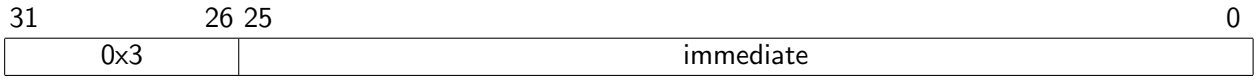
Jump

Symbolic representation: `j target`.
Unconditionally jump to the address indicated by `target`. The target address is calculated by concatenating the 4 most-significant bits of the address of the following instruction, with the immediate multiplied by 4.



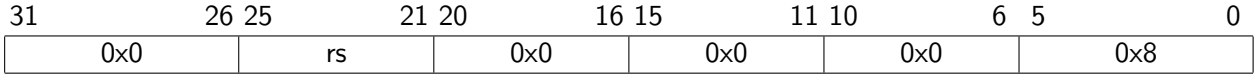
Jump and link

Symbolic representation: `jal target`.
Save the address of the following instruction in register `$ra` ($31), and unconditionally jump to the address indicated by target, which is calculated as in the jump instruction.



Jump register

Symbolic representation: `jr rs`.
Unconditionally jump to the address in register `rs`.



Halt

This final instruction is not a MIPS instruction, but is included for the purposes of the practical.
Symbolic representation: `halt`.
Causes the simulated processor to halt.