Informatics 2C — Computer Systems
Coursework 2
Deadline: Mon 29 November 2010, 12:00 noon

1 Introduction

This practical is based on material in the Computer Systems thread of the course. Its aim 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 essentially a hardware description language. In reality it is a library of methods based on C++, an object-oriented enhancement of the C programming language.

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 and will not use any of the SystemC features. 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 noon on Monday 29th 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 2 course guide.

Before attempting this coursework, you **have to go through the lab script called "SystemC basics"** available through 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 gtkwave.

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 and its datapath is illustrated in figure 1. All wires shown to be unconnected in the figure are actually connected to the control unit. It is 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 one further instruction, `jump`, are already implemented for you in the supplied code.

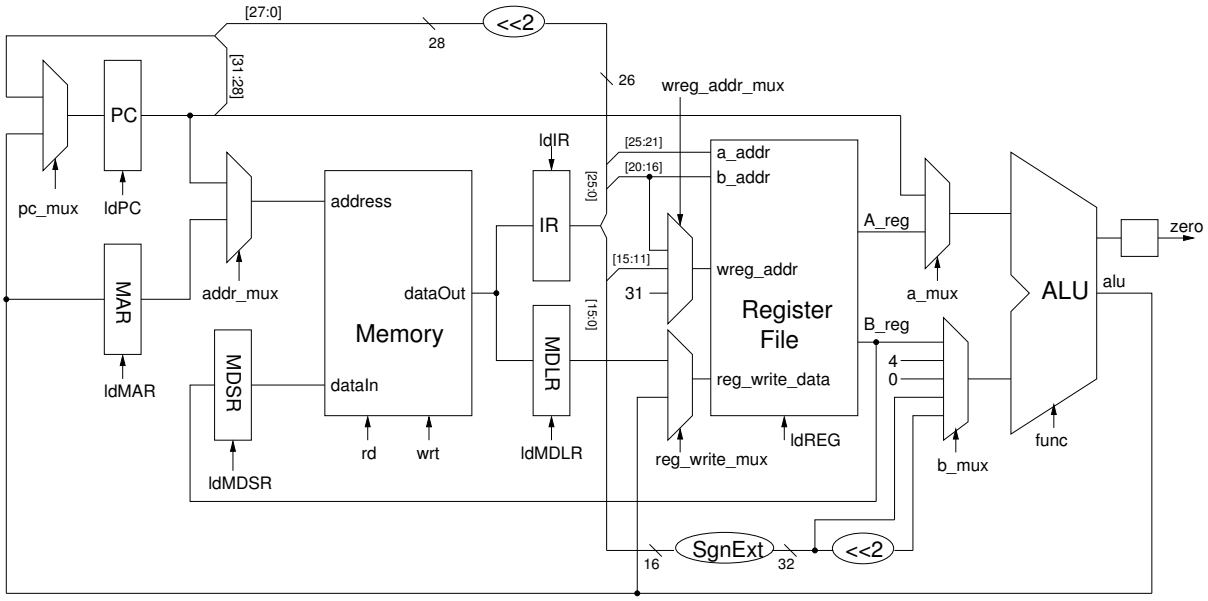


Figure 1: The datapath of the processor used in the practical.

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 instructions from a small subset of the MIPS instruction set. 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 a number of SystemC source files, a Makefile, and two other files, `mem1` and `mem2`, which contain memory maps (small programs in machine code) for loading into the simulator.

When in the `proc` directory, the SystemC model can be compiled with the command `make`, and the compiled application can be run with the command `./proc` *memoryfile*, where *memoryfile* is the name of a memory file to be loaded into the simulated memory.

3 Control unit

As explained in the lectures, the control unit of a multicycle processor is an FSM. Your task is to complete the code for the control unit of the processor, which is contained in the

function `controlUnit::ctrl_comb()`, in the file, `controlUnit.cpp`. You can ignore function `controlUnit::ctrl_regs()` in the same file which deals with initialisation, halting the simulation, and updating the flip-flops which hold the current state of the machine.

As you can see in the source code, function `controlUnit::ctrl_comb()` is essentially a large switch statement which should provide the right values for the control signals and the next state of the machine, based on the current state and a few input signals from the datapath. It models the combinational logic part of an FSM, but it is described using C code rather than circuit schematics.

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 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.

In Appendix A of this handout you will find a complete list of the control signals, 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 before the switch statement, 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). Default values are used here to save typing, as you do not have to write the value for all control signals for each case. However, you are encouraged to look into signals that can be set to a specific value for all instructions as this could lead to circuit improvements. Moreover some of the default values of the signals provided are conservative. Feel free to change these default values, as long as the processor still works correctly and with the fewest possible cycles per instruction.

4 Trying it out

Take a look at the file `mem1`. 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 `mem1` contains a small program consisting of three jump instructions and a halt instruction.

Compile the code provided for you, and try `./proc mem1`. 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 `mem1` 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 mem1`. Look at the resulting waveforms, making sure you understand what the simulation is doing and why.

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, describe for each instruction what you intend to happen on each cycle, and write down corresponding lines for the FSM truth table. Include this design documentation in your `controlUnit.cpp` file. The provided `controlUnit.cpp` file has a suggestion of what this documentation might look like. Examples of FSM truth tables can be found in the course slides and notes on logic design and in the Henessey and Patterson textbook. The 2nd and 3rd editions of this textbook also have some discussion of designing the FSM for a similar multi-cycle processor design.

Once you have this design documentation, it should be fairly straightforward to write the code itself. Be sure though to add comments to the code to explain any non-obvious aspects of the code.

You can use the memory file `mem2` to test your simulator, but you could also produce your own memory files and use these for testing.

Submit your `controlUnit.cpp` file using the following command:

```
submit inf2 inf2c-cs cw2 controlUnit.cpp
```

6 Marking

The instructions in the `controlUnit.cpp` file will be checked for correctness using an *automated script*. We will then look at your code to see how clear your design documentation is, how many cycles each instruction takes, and how appropriately your code is commented.

Note that submitted files that do not compile *will be awarded 0 marks*. So, make sure you compile and test your implementation of the instructions.

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 ldMDSR`

`true` The Memory Data Store Register is loaded (from reg. file `B_reg` output) at the end of this cycle.

`false` Do nothing.

`boolean ldMDLR`

`true` The Memory Data Load Register is loaded (from memory) 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 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` Register file output `A_reg` is selected by the A Multiplexer.

`byte b_mux`

`B_REG` Register file 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` The ALU output 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

EOR 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:

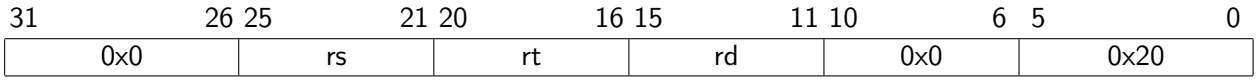
`int next_cycle` The next cycle number that the control unit should enter when the next rising clock edge arrives. Essentially this is the next state of the control unit.

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

Appendix B: Instruction set description

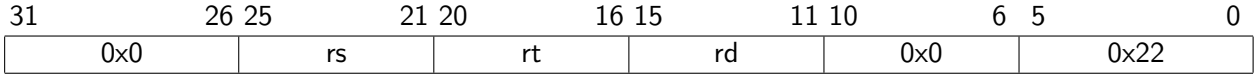
Add

Symbolic representation: `add rd, rs, rt`.
Adds the contents of registers `rs` and `rt`, and stores the result in register `rd`.



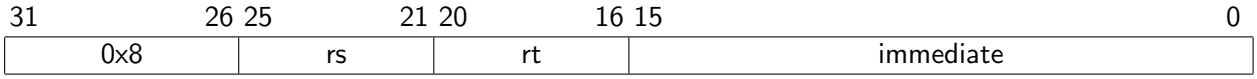
Subtract

Symbolic representation: `sub rd, rs, rt`.
Subtracts the contents of registers `rt` from `rs`, and stores the result in register `rd`.



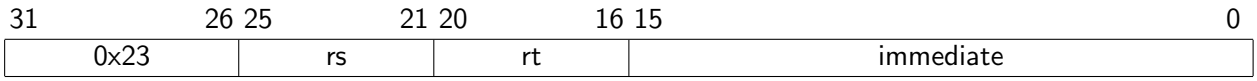
Add immediate

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`.



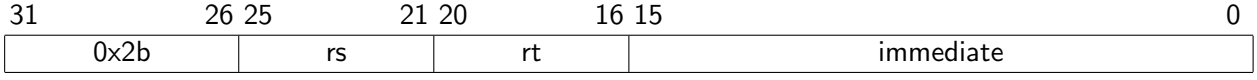
Load word

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`.



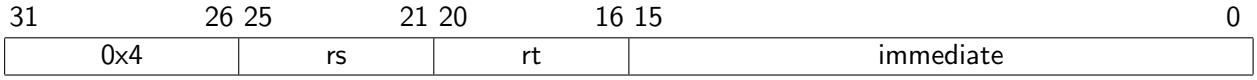
Store word

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.



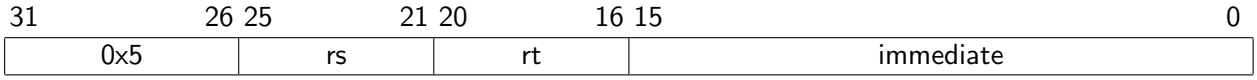
Branch on equal

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.



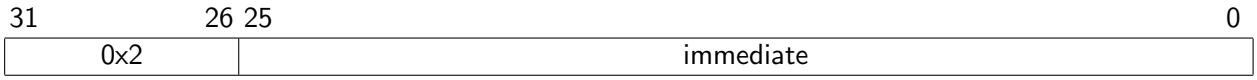
Branch on not equal

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`.



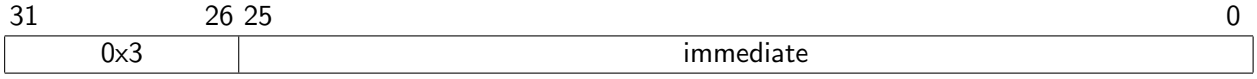
Jump

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.



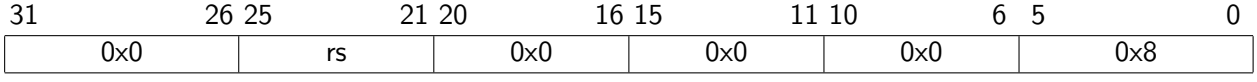
Jump and link

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.