Assignment 2: CPU Design and CPU Simulator

October 2006

In this exercise you will design a CPU and build a simulator for it. The end result is a C or C++ program that takes as input MIPS assembler code and that simulates the execution of this code (as if the code was executed as a normal executable on a MIPS processor).

1 Part 2.a CPU Design

A CPU consists of (1) a *datapath* (with registers, memory, ALU and other logic), that processes operations, and (2) a *control unit* that determines the order of these operations. It is your job to implement a micro-programmed control unit. Details about how to design this can be found in CH. 7 and 8 of the book "Logic and Computer Design Fundamentals" by M. Morris Mano en Charles R. Kime, which is the course book of "Digitale Technieken". It is recommended you study these chapters prior to making this assignment. In a previous edition of the Hennessy and Patterson book, there is also a chapter that deals with microporgrammed (also known as 'microcoded') control. This will be made available in the form of a handout. The references to Hennessy and Patterson (H&P) below refer to this handout.

1.1 Starting point

Design does not take place from scratch. As a starting point, take the MIPS implementation given in H&P. In Figure 5.1 of the handout you will find a so-called single-cycle datapath that you may use as the basis for your design. You will also find a first stab at a microcoded control for this datapath in the handout. It is *mandatory* that you build on this design. Convince yourself (*before* you hand in your solution) that whatever you want to remove from this design does no harm to the rest of the design.

One exception: you may remove *everything* that concerns exception handling. In other words, MOVI2S en MOVS2I-instructions need not be implemented. Furthermore, the IAR may be removed from the datapath and you do not need to take overflow into account.

The most important shortcoming of this design is that it implements the DLX instruction set and not the MIPS instruction set. Although the differences are minimal (as documented in the handouts), you will have to correct these. For example, some MIPS instructions (like NOR) are not implemented at all, wile others are not really executable with the given datapath and control. This concerns mainly the immediate-type instruction and the shifts.

I-type	
instr	opcode
BEQ	0x4
BNE	0x5
ADDI	0x8
ADDIU	0x9
SLTI	0x0a
SLTIU	0x0b
ANDI	0x0c
ORI	0x0d
XORI	0x0e
LUI	0x0f
LH	0x21
LB	0x20
LW	0x23
LBU	0x24
LHU	0x25
SB	0x28
SB	0x28
SH	0x29
SW	0x2b

R-type opcode = $00	
instr	funcode
SLL	0x0
SRL	0x2
SRA	0x3
SLLV	0x4
SRLV	0x6
SRAV	0x7
JR	0x8
JALR	0x9
BREAK	0x0d
ADD	0x20
ADDU	0x21
SUB	0x22
SUBU	0x23
AND	0x24
OR	0x25
XOR	0x26
NOR	0x27
SLT	0x2a
SLTU	0x2b

J-type	
instr	opcode
J	0x2
JAL	0x3

Table 1: Opcodes and Funcodes of the instructions to be implemented.

1.2 Assignments

It is your task to modify the datapath and the control in such as way that the MIPS instructions mentioned in the table below, are fully supported. The relevant specifications of MIPS can be found either in the handouts or on the assignments' webpage. Your implementation *must* comply with these specifications. The only exceptions concern the delay slots that are prescribed for some operations (e.g. jumps and branches). These may be ignored.

1.3 Datapath

Concerning the datapath: copy Figure 5.1 and draw all wires, gates and other logic that are needed to support all required MIPS instructions. Do this at "bit-level", i.e. describe exactly which bits are sent to the register file to read the right register, which bits go from the IR to the ALU to determine the ALU operation, etc.

The ALU may be implemented at "word-level"; indicate only the new functions that the ALU may require.

Memory is big-endian. Possible operations are reading (and writing) of bytes, halfwords and words. However, the address should be chosen thus, that all bytes to be read (or written) are in the same word (*word-aligned access*). In other words, valid addresses are (assuming A_1A_0 are the least significant bits):

type	$A_1 A_0$	byte 0	byte 1	byte 2	byte 3
word	0 0	X	X	X	X
halfword	0 0	X	X	–	–
halfword	1 0	–	–	X	X
byte	0 0	X	–	–	–
byte	0 1	–	X	–	–
byte	1 0	–	–	X	–
byte	1 1	–	–	–	X

Table 2: Valid addresses

Take care with load and store instructions. For instance, according to the specification, the MIPS load byte instruction

```
lb $reg, addr
```

should yield the full word of the 4-byte aligned address `addr` via the MDR (memory data register). You are responsible for extracting the appropriate byte out of this word and writing it to `$reg` by executing a few microinstructions.

1.4 Control

The datapath that you have designed is controlled using a microporgrammed control. The microprogram that you should build on is given in the handouts in Tables 5.23 and 5.25–5.29.

First scheck which modifications are needed in the microinstruction format (Figure 5.6). Leave out the microinstructions that handle MIPS instructions that are not part of the assignment and complete the microcode table. Also provide the definitive decode tables (Figures 5.24 and 5.26).

1.5 Getting started

The best way to inventorise everything that is missing in the initial design is the following. Take the design (datapath and microcode) and for each instruction in Table 1 try to execute the instruction on the datapath running the initial microcode. If this fails, then modify the datapath and/or control (microcode, decode tables, control signals, etc.

1.6 Submitting your solutions.

Submit your results *in writing* (this includes datapath, microcode, and decode tables).

datapath. Hand in the modified diagram of the original design in the book. Describe all logic that you added, what it does, and why it was needed.

control lines. Describe *all* control lines (both the old and the ones you modified). For each line, mention all possible values of the control signals that are sent by this line. For each value of these control signal, explain what will happen in the datapath if the signal has this value.

microinstructions. For each field of the microinstruction, describe all possible values and describe for each value which control lines are activated.

decode tables. For each decode table, describe the input (how many bits, coming from where?).

sequencing. Describe exactly how the address of the next microinstruction (microPC) is determined, depending on the current microPC, the current microinstruction, the values of the control lines (e.g. the 'MemReady?' line) and/or the outputs of the decode tables. Present this as short C pseudocode or a a block diagram.

2 Part 2b: CPU Simulator

For this part of the assignment, you will have to verify the paper design of the CPU that was developed in the first part. This will be done by means of simulation. You have to write the *simulator*, a C program which simulates the execution of a sequence of MIPS instructions on the CPU which was designed previously. If your design is good, then you will have no problems implementing it, so if you encounter problems then you should update your design (instead of using dirty tricks in your program). Your implementation should be a consistent reflection of your design. If your program is inconsistent with your design points will be deducted from your grade.

The choice of the programming environment is up to you. However, we require that the simulator can be compiled with gcc on *Linux* PCs and on *solaris* (beast) in LIACS without any changes. If your simulator does not compile it will not be accepted and graded. We also require that the C code is sufficiently commented and readable. The quality and readability of the code and comments will influence the grade.

Below we describe the general structure of the simulator and give some details on each part of it.

2.1 ALU

The simulator should be built in a modular fashion. Each module consists o 2 files: a `.c` file which contains the C code and a `.h` file which contains the definitions which will be used in the C files. The simulator should contain the following modules:

- A `datapath` module which implements the datapath. This module will make use of the `alu` and `mem` modules.
- A `control` module which implements the control. This module simulates the execution of a (generic) microprogram. It also contains the particular microprogram and decode tables developed in the 1st part of the assignment.
- an `alu` module which implements the ALU.
- a `memory` module which implements the memory.

- a `sim` module which combines the functions from `control` and `datapath` into a working simulator.

Use the header files for defining constants, data ctructures, function prototypes, etc., that are shared between multiple modules.

To get started, you can use the "skeletons" of a few header and C files that can be found in directory `~csca/edu/data/ca.opg3/`. Using the command `cp ~csca/edu/data/ca.opg3/* .` you can copy them all to your current directory. In this directory you will also find the Makefile which is used by the UNIX make command. In the Makefile you find the files which have to be compiled and the dependencies between different files. For example, in order to compile the file `alu.c`, you should type `make alu.o`. In order to compile the whole simulator, type `make sim`.

Now we desribe in more detail how each module will have to be implemented.

2.2 ALU

You have to write the function `alu_operation` which has 4 parameters: two 32-bit input integers, one 32 bit output number and the operation to be performed. This operation is determined by the corresponding control signal. Note that the generation of control signals `zero?` and `negative` can happen in this functions as well as in the datapath. The ALU can be implenented on the word level (bitwise approach is not necessary). See the example files in the `~csca/edu/data/ca.opg3/` directory.

2.3 Memory

This module should implement a big-endian memory of 1024 words (one word is 4 bytes). The modules should contain the three following functions which will be used in other modules: `readmem`, `writemem` and `mem_operation`.

The `readmem` function will be called from the main simulator module. This function reads a sequence of 4-byte integer numbers in hexadecimal format from the standard input. The integers are separated by the newline character. Each number represents the contents of a word in memory. The numbers should be stored consecutively in the simulator's memory, starting at address 0. The sequence will consist of at most 1024 numbers. If there are less than 1024 numbers, the remaining memory should be filled with zeros.

The `writemem` function dumps the memory contents to standard output. If all the memory contents after a certain address are zero, it should *not* print this 'tail'.

The function `mem_operation` should implement execution of one read from, or one write to the memory. In other words, it *either* reads 1 full word (in MIPS the memory always returns the whole word, as extracting the right byte, or halfword is the responsibility of the CPU), *or* it writes the appropriate byte, halfword or word. Note that one of the parameters of this function is the set of control signals, because the operation that should be performend is determined by the `memory_operation` control signal (which is determined by the `misc` field of a microinstruction. This function also sets the `mem_ready` control signal. Finally this function should detect misaligned accesses and give an error message on `stderr`.

TIP. Have a look at the data structure `union` when thinking how to implement memory.

2.4 Control

This function implements the microcoded control unit in C. It should have at least the following 3 functions: `set_signals`, `find_new_mPC` and `control_init`. The first function is responsible for setting up the control signal appropriately (given a microinstruction). The 2nd function determines the address of the next microinstruction to executed (i.e. it determines the next value of the microprogram counter mPC). The new value of mPC depends on the current value of mPC, `Cond` and `JumpLabel` fields in the current microinstruction and the values of certain control signals that are generated by the datapath, ALU or memory. These 2 functions should be called every clock cycle. The 3rd function is responsible for initialising the control (and datapath, think of the PC). It should be called once per simulator execution.

This module should also contain the tables for microcode and the decode tables developed in the 1st part of this assignment. Think of using the `struct` data structures and arrays of `struct`s when implementing these tables. We require that the functions described above should be generic, i.e. they should not depend on particular microcode/decode tables. This will make your simulator flexible: simply change the microcode/decode tables but not any of the functions if you need to implement different microprogram or decode tables.

2.5 Datapath

Implement the datapath designed in the first part of the assignment. Choose the appropriate data structure for the datapath. Think of the difference between the 'state' elements (registers, latches) and the combinatorial logic that does not have its own state and, therefore, does not require a datastructure (see Ch.7 of "Logic and Computer Design Fundamentals" by Morris Mano). **TIP:** use the `struct` datastructure.

This modules should contain at least the function: `update_state`, which takes the control signals as the parameters and updates the state of the datapath according to the value of these signals. It will call the `alu_operation` function of the ALU module and, possibly, the `mem_operation` function of the memory module. The `update_state` should be called by the simulator every clock cycle.

2.6 The simulator

Combine the modules described above to make a simulator which reads the initial memory contents from standard input, executes the program starting from address 0 till the `break` is encountered and then writes the contents of the memory to standard output.

The simulator should also print the follwoing data on `stderr`: authors of the simulator and their student numbers, total number of dynamic instructions executed by the simulator (including `break`), and total number of clock cycles required for the simulation (including `break`).

In the directory `~csca/edu/data/ca.opg3.data/` one can find the following files to test the sim-

ulator:

- `rfac.i`: the example input file
- `rfac.s`: the corresponding MIPS assembly file

TIP: use UNIX redirection to redirect the input file to standard input: `sim < rfac.i`

2.7 Deliverable

You should hand in all the `.[ch]` files, the `Makefile`, the executable `sim` and the `README` files electronically as described later in this section. All the text files hould be printed and submitted as well. All the code files should be sufficiently commented, the quality of the comments and the readability of the code will influence your grade. The simulator should be compatible with gcc and should work on Linux and Solaris (beast) without any modifications. Solutions that do not compile will be rejected. In the `README` file you should describe the general structure of the simulator and how different modules interact. If you have introduced changes to your original design, describe them in this file as well.

Technical issues. Deliver files both electronically and as printouts. To deliver the files electronically, use the following:

```
tar cvzf yourname_opg2.tgz [list of file you want to send]

uuencode yourname_opg2.tgz yourname_opg2.tgz | elm -s "Opgave2" ca
```