Laboratory 2

Objective

The objective of this laboratory is to implement an instruction interpreter for the instruction set of the KURM (KU RISC Machine) CPU that will be the result of your project. This device is a simple 9-instruction RISC-style CPU. As in traditional RISC architectures, the instruction set is composed of a small number of fixed-sized instructions with simple addressing modes. All arithmetic and logic operations operate on and store operands within the register file. The only memory addressing mode used by data transfer and branching instructions is register indirect.

The word length for this device is 16 bits. Although KURM reads and writes words, memory addresses index bytes. The following table shows the word in memory associated with each address:

Byte Address	0	1	2	3	4	5
0	**Data**					
1		**Data**				
2			**Data**			
3				**Data**		
4					**Data**	

The low byte of each word is designated as the most significant byte, and the high byte of each word the least significant. Because instructions are 16 bits in length, the program counter must be incremented by 2 to move to the next instruction.

There are 16 orthogonal, general-purpose registers available, as well as a 16-bit program counter. Register R_0 contains 0. Instructions specify register IDs using 4-bit values. The program counter cannot be directly addressed, but is affected by branch and jump instructions.

Your KURM will implement the following instruction set:

Instruction	Pseudo description	Op_code 4 bits	Rs 4 bits	Rt 4 bits	Rd 4 bits
ADD R_d, R_s, R_t	$R_d := R_s + R_t$	2	0–15	0–15	0–15
SUB R_d, R_s, R_t	$R_d := R_s - R_t$	6	0–15	0–15	0–15
AND R_d, R_s, R_t	$R_d := R_s$ "•" R_t	0	0–15	0–15	0–15
OR R_d, R_s, R_t	$R_d := R_s$ "+" R_t	1	0–15	0–15	0–15
SLT R_d, R_s, R_t	if ($R_s < R_t$) $R_d := 1$ else $R_d := 0$	7	0–15	0–15	0–15
LW R_d, off(R_s)	$R_d := M[off + R_s]$	8	0–15	0–15	offset
SW R_d, off(R_s)	$M[off + R_s] := R_r$	A	0–15	0–15	offset
BNE R_s, R_t<offset>	if (R_s != R_t) pc := pc + off +4	E	0–15	0–15	offset
JMP <addr_off>	pc := pc:addr:0	F	12 bit offset		

All instructions are one word in length, with the format of each instruction shown in the previous table. The high four bits always specify the operation, while the low 12 bits specify registers and offsets, depending on the instruction type. Mathematical operations (`ADD`, `SUB`, `AND`, `OR` and `SLT`) operate only on registers. Data transfer and branching operations (`LW`, `SW` and `BNE`) operate on two registers and an absolute offset value. The jump operation (`JMP`) operates on a single 12-bit offset.

Mathematical and logical operations treat the low 12 bits as register identifiers. The high four bits represent R_d, the middle four R_s and the low four R_t, as specified in the previous table. Addition, subtraction and set-greater-than treat the contents of R_s and R_t as 16-bit, 2s-compliment numbers. An overflow value should be generated by these instructions. Conjunction and disjunction treat R_s, R_t and R_d as unsigned, 16-bit values.

Load and store operations use the low 12 bits to specify a memory address, a source/destination register and an offset. R_s specifies the register containing a base address, R_d specifies the offset and R_t specifies the destination (or source) for data being read (or stored). Note that the only addressing mode is register indirect.

The branch-not-equal (`BNE`) operator uses the low 12 bits to specify the registers for comparison and a branch offset. R_s and R_t specify registers whose values are to be compared. If they are not equal, the program counter is incremented (or decremented) by the number of words specified by the offset value plus four bytes. The additional four may seem somewhat odd, but there are solid technical reasons for doing this that will become clear later.

Remember, offsets for loading, storing and branching are 4-bit, 2s-complement numbers that specify offsets as words. Be cautious as you add and subtract offsets to get new program counter values. Further realize that the length of the offset limits how far a program can branch using the `BNE` command.

The jump operation uses the low 12 bits to specify a single offset value in a substantially different way. Unlike the branch offset, the jump offset is not interpreted as a 2s-complement number. The jump address is calculated as follows:

15:13	12 ... 1	0
PC 15:13	addr_offset	0

The 16-bit absolute address is formed by first multiplying the offset value by 2, resulting in a 13-bit number. Then, the three most significant bits from the program counter are prepended to the result to produce a 16-bit number. The `JMP` operation is achieved by setting the program counter to this value. Note that the absolute address can be constructed extremely efficiently. Your implementations should reflect this. Further note that the jump operation cannot reach all memory locations.

Instruction Interpreter

An instruction interpreter is a behavioral model of a CPU that is frequently used early in a design cycle to prototype the device. In some cases, the instruction interpreter can be used by software developers to start coding activities, even before hardware prototypes or emulators are available. You will use your instruction interpreter to familiarize yourself with the KURM design and to develop a test entity for use throughout the design process.

We will use a single VHDL process, activated by the clock signal, to process instructions. The following VHDL skeleton gives a template for implementing the interpreter: (Note that items in *italics* represent things you must define or implement.)

```
entity cpu is
  port (datain: in word; dataout: out word;
        address: out word; rd,wr: out bit;
        reset,clk: in bit);
end entity cpu;
```

Your entity should support a clock input and communication with a memory device. The `datain` and `dataout` ports are used to input and output data, respectively. The `address` port is used to specify a source or destination address, depending on the operation performed. The `rd` and `wr` ports are strobes that instruct the memory to output (`rd`) or store (`wr`) data, respectively.

To write to memory:

1. Set `dataout` to the data value to be stored
2. Set `address` to the address where the data should be stored
3. Output a pulse on `wr`

To read from memory:

1. Set `address` to the address where the data should be read from
2. Output a pulse on `rd`
3. Read the data from `datain`

The `clk` input implements a standard clock and causes the machine to read and execute the next instruction. The `reset` input is used to set the program counter to 0, implementing a reset action.

A behavioral memory model is provided with this laboratory. There are two files you must include; the behavioral model of memory and a collection of memory utilities. See the memory source file for an example of how to use the memory model. It is extremely important to read the documentation in the memory model before you use it. You will find memory initialization and dumping capabilities to help you test your system.

Your architecture should use variables to represent registers and define each instruction over those registers. It should repeatedly fetch instructions from memory and execute them using a case statement that uses the instruction to determine what case to select.

```
architecture interpreter of cpu is
begin
  update: process
    Variables for registers and program counter
  begin
    Wait on a rising clock edge
    Fetch instruction from memory
    Decode the instruction
    Execute the instructions using a case statement
  end process update;
end architecture interpreter;
```

Problem 1

Develop an entity model for the interface of the KURM. This entity should define all inputs into the KURM and outputs generated. This must include the interface to memory, the system clock and the CPU reset. A template for this entity is provided above.

Problem 2

Develop behavioral instruction interpreter architecture for the KURM. The instruction interpreter should be structured and behave like the instruction interpreter defined above. It should execute one instruction per clock cycle.

Problem 3

Integrate the behavioral model with a memory module in a test entity. Create a test entity that includes your CPU model, a clock model and the memory model provided. Define the interconnections between these components.

Problem 4

Exercise the model to demonstrate correctness. Load one or more program into memory, reset the CPU and start the clock. When your program has terminated, dump the memory to assure correct execution of the program. Note that your test programs should exercise your CPU model thoroughly.

Optional Problem

Modify your memory and CPU interface so that, instead of simply waiting a fixed time for memory access to occur, a data ready signal is asserted when the memory access is complete. Use the data ready signal as an input to the CPU and use it to cause the CPU to wait on memory access instructions to complete.

Things to remember

- Instruction decode and execute are achieved with a case statement. See the VHDL Tutorial on this CD for a description of the case statement. Each instruction will have its own entry in the case statement associated

with behavioral VHDL implementing the instruction.

- You will find definitions for `word`, `byte` and `nybble` types included in the memory module definition. It is highly recommended that you define functions to extract the various instruction fields from a word.
- Implementing registers as an array of words may simplify implementing the instruction decode.
- Defining functions that convert back and forth between numbers and binary values could prove valuable. Remember that offsets are 4-bit, 2s-compliment values. This implies that you can have negative offsets.
- It will take at least five hours to implement and test the KURM behavioral model after completing the memory module. Do not put this off until the night before it is due.