Milestone 3

Objective

The objective of this milestone is to implement the controller and data path for the instruction set of the KURM (KU RISC Machine) CPU and integrate them together. 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. 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. 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).

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 two bytes. The additional two 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.

Lab 3 Exercises

Problem 1

Draw the data path of your processor with the control unit. The data path should include the register file and the ALU which was implemented in the previous milestone plus components such as memory, program counter, multiplexers etc.

Problem 2

Design and implement a behavioral model for the KURM controller that implements KURM instructions. Your controller should be implemented behaviorally using case statements and/or if statements in a Verilog module. Document (put comments) all inputs and outputs defined for the controller. Your controller should change states on each rising edge of the clock pulse. Write a testbed to test whether you controller gives correct control outputs to all instructions.

Project

Problem 3

Implement memory modules and other additional components required for your processor.

Problem 4

Design and implement a model for the KURM data path. Use your behavioral register file and ALU models from milestone 2 in conjunction with other components such as memory and multiplexers to implement your system.

Your data path should be implemented as a single module. Document (put comments) all control settings and outputs from the data path. Remember that you must include the system clock to trigger data storage elements.

Note that, in the design of the data path, you should use behavioral models for both the register file and the ALU. You may modify your behavioral models as necessary to suite your particular design. It is unlikely that your register file will require modification, but the control signals to the ALU may be reconfigured to more effectively interpret instructions.

Problem 5

Integrate your controller and data path to define a structural module for the KURM. Test your structural KURM implementation by a loading a sequence of instructions to the instruction memory.