Indian Institute of Technology Kharagpur

AUTUMN Semester, 2019
COMPUTER SCIENCE AND ENGINEERING

Computer Organization Laboratory

Assignment-10: Verilog Design and Implementation of a Single-cycle RISC ISA

Full Marks: 30

Time allowed: ϵ hours

INSTRUCTIONS: **Submission Instructions:** Make one submission per group in the form of a single zipped folder containing your Verilog source code files(s), Verilog testbench(es), and the documentation (in PDF format). Name your submitted zipped folder as `Assgn_10_Grp_<Group_no>.zip` and (e.g. `Assgn_10__Grp_25.zip`). Inside each submitted source and testbench files, there should be a clear header describing the assignment no., problem no., semester, group no., and names of group members. Liberally comment your code to improve its comprehensibility.
Problem statement starts from the next page.

Class	Instruction	Usage	Meaning
Arithmetic	Add	add rs,rt	$rs \leftarrow (rs) + (rt)$
	Multiply (unsigned)	multu rs,rt	$\{reg_{19}, reg_{20}\} \leftarrow (rs) \times_{unsigned} (rt)$
	Multiply (signed)	mult rs,rt	$\{reg_{19}, reg_{20}\} \leftarrow (rs) \times_{signed} (rt)$
	Comp	comp rs,rt	$rs \leftarrow$ 2's Complement (rt)
	Add immediate	addi rs,imm	$rs \leftarrow (rs) + imm$
	Complement Immediate	compi rs,imm	$rs \leftarrow$ 2's Complement (imm)
Logic	AND	and rs,rt	$rs \leftarrow (rs) \wedge (rt)$
	XOR	xor rs,rt	$rs \leftarrow (rs) \oplus (rt)$
Shift	Shift left logical	shll rs, sh	$rs \leftarrow (rs)$ left-shifted by sh
	Shift right logical	shrl rs, sh	$rs \leftarrow (rs)$ right-shifted by sh
	Shift left logical variable	shllv rs, rt	$rs \leftarrow (rs)$ left-shifted by (rt)
	Shift right logical variable	shrlv rs, rt	$rs \leftarrow (rs)$ right-shifted by (rt)
	Shift right arithmetic	shra rs, sh	$rs \leftarrow (rs)$ arithmetic right-shifted by sh
	Shift right arithmetic variable	shrav rs, rt	$rs \leftarrow (rs)$ right-shifted by (rt)
Memory	Load Word	lw rt,imm(rs)	$rt \leftarrow mem[(rs) + imm]$
	Store Word	sw rt,imm,(rs)	$mem[(rs) + imm] \leftarrow (rt)$
Branch	Unconditional branch	b L	goto L
	Branch Register	br rs	goto (rs)
	Branch on zero	bz L	if $(zflag == 1)$ then goto L
	Branch on not zero	bnz L	if$(zflag == 0)$ then goto L
	Branch on Carry	bcy L	if $(carryflag == 1)$ then goto L
	Branch on No Carry	bncy L	if $(carryflag == 0)$ then goto L
	Branch on Sign	bs	if $(signflag == 1)$ then goto L
	Branch on Not Sign	bns L	if $(signflag == 0)$ then goto L
	Branch on Overflow	bv L	if $(overflowflag == 1)$ then goto L
	Branch on No Overflow	bnv L	if $(overflowflag == 0)$ then goto L
	Call	Call L	$ra \leftarrow$ (PC)+4 ; goto L
	Return	Ret	goto (ra)

1. Our processor KGP-RISC has the Instruction Set Architecture (ISA) as shown above. Assume that the processor word size is 32 bits, and the processor uses 32-bit addresses. Also, assume the processor has total 32 registers, of which certain registers (e.g. `ra` which stores the return address) play similar roles as the corresponding registers in the MIPS-32 ISA. Out of the 32 registers, register no. 19 and reg. no. 20 together can be used to hold the product of a signed/unsigned multiply operation on two 32-bit numbers – reg. no. 19 is used to hold the most significant word and reg. no. 20 for the least significant word, while the opcode determines whether the multiplication is signed or unsigned. You are to develop first the opcode format for the above instruction set, identify the data path element(s), and design the data path along with the necessary control signals. Subsequently, we shall develop a single-cycle instruction execution unit for KGP-RISC.

Proceed step-by-step as follows (each major step should be accompanied by documentation in your design document):

(a) Decide the registers and the register usage convention.

(b) Design a suitable instruction format and instruction encoding. While deciding the opcode, you should keep provisions for adding more instructions to the ISA.

(c) Design and implement the *Instruction Fetch* phase, and identify whether you need to store the fetched instruction in a register or not.

(d) Design and implement the *Instruction Decoder*, based on the instruction encoding you have adopted.

(e) Design and implement the *Register Bank*, based on the instruction encoding you have adopted. It is recommended that you adopt a hierarchical design style for the register bank, and avoid extremely large multiplexors, etc.

(f) Design and implement the *Arithmetic Logic Unit* (ALU) in a hierarchical manner, with possibly different modules responsible for different operations. For each individual module, decide the coding style that

you will adopt. If possible, try to perform logic sharing among the different modules inside the ALU, to decrease the hardware footprint of the ALU. Remember the role of the different special flag registers that are part of the ISA.

(g) Design and implement the *Branching Logic*, in conjunction with the ALU you have designed.

(h) Design and implement the *Load-Store unit*. You would be taught the usage of *Block RAM* (**BRAM**) hard-macro modules on Xilinx FPGAs, which can be directly instantiated in your code, and which can be used to emulate memory. Your instruction fetch unit and load-store unit will interact with the BRAM module(s) for instruction fetch and data transfer.

(i) Design and implement the *Control Unit*. Consider the truth table for the controller signals as a function of the opcode and the function code in the instruction format. Note that for a single cycle implementation of the controller there is no state and the design is purely combinational.

(j) Finally, simulate the entire design (pre- and post-synthesis), with proper testbenches. It is desirable that you write the assembly language code for a simple algorithm (e.g. Bubblesort) targeting the processor you have designed, and demonstrate its working as a testcase on a FPGA.

Work Schedule (status updates would be conducted in lab)

- Up to Step (b): by 14/10/2019 EoD
- Up to Step (e): by 16/10/2019 EoD
- Up to Step (g): by 23/10/2019 EoD
- Up to Step (i): by 28/10/2019 EoD
- Complete design: by 30/10/2019 EoD