6.004 Computation Structures
Spring 2009

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.004 Computation Structures
Lab #7

The goal of this lab is to add support for two new instructions to the Beta. But instead of adding hardware, we'll support the instructions in software (!) by writing the appropriate emulation code in the handler for "illegal instruction" exceptions.

The new instructions implement load and store operations for byte (8-bit) data:

LDB

Usage: LDB(Ra,literal,Rc)

Opcode:

010000	Rc	Ra	literal

Operation:

$PC \leftarrow PC+4$
$EA \leftarrow Reg[Ra] + SEXT(literal)$
$MDATA \leftarrow Mem[EA]$
$Reg[Rc]_{7:0} \leftarrow$ if $EA_{1:0} = 0b00$ then $MDATA_{7:0}$
else if $EA_{1:0} = 0b01$ then $MDATA_{15:8}$
else if $EA_{1:0} = 0b10$ then $MDATA_{23:16}$
else if $EA_{1:0} = 0b11$ then $MDATA_{31:24}$
$Reg[Rc]_{31:8} \leftarrow 0x000000$

The effective address EA is computed by adding the contents of register Ra to the sign-extended 16-bit displacement *literal*. The byte location in memory specified by EA is read into the low-order 8 bits of register Rc; bits 31:8 of Rc are cleared.

STB

Usage: STB(Rc,literal,Ra)

Opcode:

010001	Rc	Ra	literal

Operation:

$PC \leftarrow PC+4$
$EA \leftarrow Reg[Ra] + SEXT(literal)$
$MDATA \leftarrow Mem[EA]$
if $EA_{1:0} = 0b00$ then $MDATA_{7:0} \leftarrow Reg[Rc]_{7:0}$
else if $EA_{1:0} = 0b01$ then $MDATA_{15:8} \leftarrow Reg[Rc]_{7:0}$
else if $EA_{1:0} = 0b10$ then $MDATA_{23:16} \leftarrow Reg[Rc]_{7:0}$
else if $EA_{1:0} = 0b11$ then $MDATA_{31:24} \leftarrow Reg[Rc]_{7:0}$
$Mem[EA] \leftarrow MDATA$

The effective address EA is computed by adding the contents of register Ra to the sign-extended 16-bit displacement *literal*. The low-order 8-bits of register Rc are written into the byte location in memory specified by EA. **The other bytes of the memory word remain unchanged**.

When the Beta hardware (which doesn't know about these instructions) detects either of the two opcodes above, it will cause an "illegal instruction" exception (see section 6.4 of the Beta documentation) and set the PC to 4.

The checkoff code has loaded location 4 with "BR(UI)" that branches to an assembly language routine labeled UI which handles illegal instructions – this is the routine that you need to write. It should do the following:

1. Determine if the opcode for the illegal instruction is for LDB or STB. The address of the instruction *after* the illegal instruction has been loaded into register XP by the hardware (i.e., the illegal instruction is at memory address Reg[XP]-4).

2. If the illegal instruction is *not* LDB or STB, your routine should branch to the label _IllegalInstruction – note the leading underscore. Before branching, the contents of all the registers should be the same as they were when your routine was entered. So you should save and restore any registers you use in Step 1.

3. If the illegal instruction is LDB or STB, your routine should perform the appropriate memory and register accesses to emulate the operation of these instructions. Your routine will have to decode the instruction at Reg[XP]-4 to determine what registers and memory locations to use.

4. When your emulation is complete, return control to the interrupted program at the instruction following the LDB or STB. The contents of all the registers should be the same as they were when your routine was entered, except for the register changed by LDB. So you need to save and restore any registers you use in steps 1 and 3.

To test your code, we'll be using the BSim beta simulator. In order to interface properly with the checkoff code, your assembly language program should follow the template below:

```
.include /mit/6.004/bsim/beta.uasm
.include /mit/6.004/bsim/lab7checkoff.uasm

UI:
      … your assembly language code here …
```

Lab7checkoff.uasm contains the checkoff code for this lab. When execution begins, it does the appropriate initialization (setting SP to point to an area of memory used for the stack, etc.) and then executes a small test program that includes LDB and STB instructions that test your emulation routine**.** The program will type out messages as it executes, reporting any errors it detects. When it types "Checkoff tests completed successfully!", you're ready to submit your code to the on-line checkoff system.

To help you get started here's an example illegal instruction handler that emulates a new instruction swapreg(RA,RC) which interchanges the values in registers RA and RC. This example can found on-line in /mit/6.004/bsim/swapregs.uasm and on the Courseware webpage. The example includes lab7macros.uasm, a file containing some useful macros for saving/restoring registers and extracting bit fields from a 32-bit word.

```
.include /mit/6.004/bsim/beta.uasm
.include /mit/6.004/bsim/lab7macros.uasm
```

```
||| Handler for opcode 1 extension:
||| swapreg(RA,RC) swaps the contents of the two named registers.
||| UASM defn = .macro swapreg(RA,RC) betaopc(0x01,RA,0,RC)

regs:   RESERVE(32)            | Array used to store register contents

UI:
        save_all_regs(regs)

        LD(xp,-4,r0)           | illegal instruction
        extract_field(r0, 31, 26, r1)   | extract opcode, bits 31:26
        CMPEQC(r1,0x1,r2)      | OPCODE=1?
        BT(r2, swapreg)        | yes, handle the swapreg instruction.

        LD(r31,regs,r0)        | It's something else.  Restore regs
        LD(r31,regs+4,r1)      | we've used, and go to the system's
        LD(r31,regs+8,r2)      | Illegal Instruction handler.
        BR(_IllegalInstruction)

swapreg:
        extract_field(r0, 25, 21, r1)   | extract rc field
        MULC(r1, 4, r1)        | convert to byte offset into regs array
        extract_field(r0, 20, 16, r2)   | extract ra
        MULC(r2, 4, r2)        | convert to byte offset into regs array
        LD(r1, regs, r3)       | r3 <- regs[rc]
        LD(r2, regs, r4)       | r4 <- regs[ra]
        ST(r4, regs, r1)       | regs[rc] <- old regs[ra]
        ST(r3, regs, r2)       | regs[ra] <- old regs[rc]

        restore_all_regs(regs)
        JMP(xp)
```