

6.004 Computation Structures
Lab #5B

Note: This lab has two parts, 5A and 5B. Lab #5A continues the development of your Beta design using JSim. Lab #5B (this document) implements two new Beta instructions in software using BSim.

Project Description

The goal of this design project 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:	<table style="border-collapse: collapse; width: 100%;"> <tr> <td style="width: 25%; text-align: center; padding: 2px;">010000</td> <td style="width: 15%; text-align: center; padding: 2px;">Rc</td> <td style="width: 15%; text-align: center; padding: 2px;">Ra</td> <td style="width: 45%; text-align: center; padding: 2px;">literal</td> </tr> </table>	010000	Rc	Ra	literal
010000	Rc	Ra	literal		
Operation:	$PC \leftarrow PC+4$ $EA \leftarrow \text{Reg}[Ra] + \text{SEXT}(\text{literal})$ $MDATA \leftarrow \text{Mem}[EA]$ $\text{Reg}[Rc]_{7:0} \leftarrow \begin{cases} \text{MDATA}_{7:0} & \text{if } EA_{1:0} = 0b00 \\ \text{MDATA}_{15:8} & \text{else if } EA_{1:0} = 0b01 \\ \text{MDATA}_{23:16} & \text{else if } EA_{1:0} = 0b10 \\ \text{MDATA}_{31:24} & \text{else if } EA_{1:0} = 0b11 \end{cases}$ $\text{Reg}[Rc]_{31:8} \leftarrow 0x000000$				

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:	<table style="border-collapse: collapse; width: 100%;"> <tr> <td style="width: 25%; text-align: center; padding: 2px;">010001</td> <td style="width: 15%; text-align: center; padding: 2px;">Rc</td> <td style="width: 15%; text-align: center; padding: 2px;">Ra</td> <td style="width: 45%; text-align: center; padding: 2px;">literal</td> </tr> </table>	010001	Rc	Ra	literal
010001	Rc	Ra	literal		
Operation:	$PC \leftarrow PC+4$ $EA \leftarrow \text{Reg}[Ra] + \text{SEXT}(\text{literal})$ $MDATA \leftarrow \text{Mem}[EA]$ $\text{if } EA_{1:0} = 0b00 \text{ then } MDATA_{7:0} \leftarrow \text{Reg}[Rc]_{7:0}$ $\text{else if } EA_{1:0} = 0b01 \text{ then } MDATA_{15:8} \leftarrow \text{Reg}[Rc]_{7:0}$ $\text{else if } EA_{1:0} = 0b10 \text{ then } MDATA_{23:16} \leftarrow \text{Reg}[Rc]_{7:0}$ $\text{else if } EA_{1:0} = 0b11 \text{ then } MDATA_{31:24} \leftarrow \text{Reg}[Rc]_{7:0}$ $\text{Mem}[EA] \leftarrow MDATA$				

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)" which 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 location at 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 described later in this document. In order to interface properly with the checkoff code, your assembly language program should follow the template below:

```
.include beta.uasm
.include lab5Bcheckoff.uasm

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

Beta.uasm contains assembly language macro and symbol definitions for all the Beta instructions and register names. Lab5Bcheckoff.uasm contains the checkoff code for this project. 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.

You'll find breakpoints and single-stepping to be useful techniques in debugging your program. See below for details.

To help you get started here's an example illegal instruction handler that emulates an new instruction `swapreg(RA,RC)` which interchanges the values in registers RA and RC. This example can found on-line in `swapregs.uasm` and on the Related Resources page. The example includes `lab5Bmacros.uasm`, a file containing some useful macros for saving/restoring registers and extracting bit fields from a 32-bit word.

```
.include beta.uasm
.include lab5Bmacros.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)           | Its 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)
```

Introduction to assembly language

Bsim incorporates an *assembler*: a program that converts text files into binary memory data. The simplest assembly language program is a sequence of numerical values which are converted to binary and placed in successive *byte* locations in memory:

```
| Comments begin with vertical bar and end at a newline

37 3 255      | decimal (the default radix)
0b100101     | binary (note the 0b prefix)
0x25         | hexadecimal (note the 0x prefix)
'a'          | character constants
```

Values can also be expressions; e.g., the source file

```
37+0b10-0x10    24 - 0x1  4*0b110-1    0xF7 % 0x20
```

generates 4 bytes of binary output, each with the value 23. Note the operators have no precedence – you have to use parentheses to avoid simple left-to-right evaluation. The available operators are

- unary minus
- ~ bit-wise complement
- + addition
- subtraction
- * multiplication
- / division
- % modulo (result is always positive!)
- >> right shift
- << left shift

We can also define *symbols* for use in expressions:

```
x = 0x1000      | address in memory of variable X
y = 0x10004     | another address

| Symbolic names for registers
R0 = 0
R1 = 1
...
R31 = 31
```

Note that symbols are case-sensitive: “Foo” and “foo” are different symbols. A special symbol named “.” (period) means the address of the next byte to be filled by the assembler:

```
. = 0x100      | assemble into location 0x100
 1  2  3  4
five = .      | symbol five has the value 0x104
 5  6  7  8
. = . + 16    | skip 16 bytes
 9 10 11 12
```

Labels are symbols that represent memory address. They can be set with the following special syntax:

```
X:          | this is an abbreviation for X = .
```

For example the table on the left shows what main memory will contain after assembling the program on the right.

```
----- MAIN MEMORY -----
byte:  3  2  1  0

1000: 09 04 01 00
1004: 31 24 19 10
1008: 79 64 51 40
100C: E1 C4 A9 90
1010: 00 00 00 10

. = 0x1000
sqrs: 0 1 4 9
      16 25 36 49
      64 81 100 121
      144 169 196 225
slen: LONG(. - sqrs)
```

Macros are parameterized abbreviations:

```
| macro to generate 4 consecutive bytes
.macro consec(n) n n+1 n+2 n+3

| invocation of above macro
consec(37)
```

The macro invocation above has the same effect as

```
37 38 39 40
```

Note that macros evaluate their arguments and substitute the resulting value for occurrences of the corresponding formal parameter in the body of the macro. Here are some macros for breaking multi-byte data into byte-size chunks

```
| assemble into bytes, little-endian format
.macro WORD(x) x%256 (x/256)%256
.macro LONG(x) WORD(x) WORD(x>>16)

LONG(0xdeadbeef)
```

Has the same effect as

```
0xef 0xbe 0xad 0xde
```

The body of the macro includes the remainder of the line on which the `.macro` directive appears. Multi-line macros can be defined by enclosing the body in “{“ and “}”.

Beta.uasm contains macro definitions for all the Beta instructions as well as convenience macros for PUSH, POP, loading constants, etc. Browse through that file to learn what’s available.

The following is an complete example assembly language program:

```
.include beta.uasm

. = 0                | start assembling at location 0
LD(input,r0)        | put argument in r0
BR(bitrev,r28)      | call bit reverse procedure
HALT()

| reverse the bits in r0, leave result in r1
bitrev:
  CMOVE(32,r2)       | loop counter
  CMOVE(0,r1)        | clear output register
loop:
  ANDC(r0,1,r3)      | get low-order bit
  SHLC(r1,1,r1)      | shift output word by 1
  OR(r3,r1,r1)       | OR in new low-order bit
  SHRC(r0,1,r0)      | done with this input bit
  SUBC(r2,1,r2)      | decrement loop counter
  BNE(r2,loop)       | repeat until done
  JMP(r28)           | return to caller

input:
```

LONG(0x12345) | 32-bit input (in HEX)

The BSim assembly language processor include a few helpful directives:

- `.include filename`
Process the text found in the specified file at this point in the assembly.
- `.align`
`.align expression`
Increment the value of “.” until it is 0 modulo the specified value, e.g., “.align 4” moves to the next word boundary in memory. A value of 4 is used if no expression is given.
- `.ascii "chars..."`
Assemble the characters enclosed in quotes into successive bytes of memory. C-like escapes can be used for non-printing characters.
- `.text "chars..."`
Like `.ascii` except an additional 0 byte is added to the end of the string in memory.
- `.breakpoint`
Stop the Beta simulator if it fetches an instruction from the current location (i.e., the value of “.” at the point the `.breakpoint` directive occurred). You can define as many breakpoints as you want.
- `.protect`
This directive indicates that subsequent bytes output by the assembler are “protected,” causing the simulator to halt if a ST instruction tries to overwrite their value. This directive is useful for protecting code (e.g., the checkoff program) from being overwritten by errant programs.
- `.unprotect`
The opposite of `.protect` – subsequent bytes output by the assembler are not protected and can be overwritten by the program.
- `.options ...`
Used to configure the simulator. Available options:
 - `clk` enable periodic clock interrupts to location 8
 - `nodiv` make the DIV opcode an illegal instruction
 - `nomul` make the MUL opcode an illegal instruction
 - `kalways` don't let program enter user mode (ie, supervisor bit is always 1)
 - `tty` enable RDCHAR(), WRCHAR() instructions (see end of next section)
- `.pcheckoff ...`
- `.verify ...`
Supply checkoff information to the simulator.

Introduction to BSim

In this lab, we'll be using the BSim beta simulator to execute the program you've written. The BSim user interface is very similar to JSim's: there's a simple editor for typing in your program and some tools for assembling the program into binary, loading the program into the simulated Beta's memory, executing the program and examining the results.

To run BSim on a Unix machine, type the following at the Unix prompt

```
% bsim &
```

It can take a few moments for the Java runtime system to start up, please be patient! BSim takes as input a *assembly language program* to be executed. The initial BSim window is a very simple editor that lets you enter and modify your netlist. If you use a separate editor to create your netlists, you can have BSim load your netlist files when it starts:

```
% bsim filename ... filename &
```

There are various handy buttons on the BSim toolbar:



Exit. Asks if you want to save any modified file buffers and then exits BSim.



New file. Create a new edit buffer called "untitled". Any attempts to save this buffer will prompt the user for a filename.



Open file. Prompts the user for a filename and then opens that file in its own edit buffer. If the file has already been read into a buffer, the buffer will be reloaded from the file (after asking permission if the buffer has been modified).



Close file. Closes the current edit buffer after asking permission if the buffer has been modified.



Reload file. Reload the current buffer from its source file after asking permission if the buffer has been modified. This button is useful if you are using an external editor to modify the netlist and simply want to reload a new version for simulation.



Save file. If any changes have been made, write the current buffer back to its source file (prompting for a file name if this is an untitled buffer created with the "new file" command). If the save was successful, the old version of the file is saved with a ".bak" extension.



Save file, specifying new file name. Like "Save file" but prompts for a new file name to use.



Save all files. Like "save file" but applied to all edit buffers.



Assemble the current buffer, i.e., convert it into binary and load it into the simulated Beta's memory. Any errors detected will be flagged in the editor window and described in the message area at the bottom of the window. If the assembly completes successfully, a window showing the Beta datapath is created from which you can start execution of the program.



Assemble the current buffer and output the resulting binary to a file whose name is the same as source file for the current buffer with ".bin" appended.



Using information supplied in the checkoff file, check for specified memory values. If all the checks are successful, submit the program to the on-line assignment system.

The datapath window has some additional toolbar buttons that are used to control the simulation. The values shown in the window reflect the values on Beta signals after the current instruction has been fetched and executed but just before the register file is updated at the end of the cycle.



Stop execution and update the datapath display.



Reset the contents of the PC and registers to 0, and memory locations to the values they had just after assembly was complete. You have to stop a running simulation before a reset.



Start simulation and run until a HALT() instruction is executed or a breakpoint is reached. You can stop a running simulation using the stop control described above. For maximum simulation speed, the datapath display is not updated until the simulation is stopped.



Execute the program for a single cycle and then update the datapath display. Very useful for following your program's operation instruction-by-instruction.



Bring up a window that let's you configure the cache parameters for main memory.

If ".options tty" is specified by the program, a small 5-line typeout window appears at the bottom of the datapath window. You can output characters to this window by executing a WRCHAR() instruction after placing the character value in R0. The tty option also allows for type-in: any character typed by the user causes an interrupt to location 12; RDCHAR() can be used to fetch the character value into R0.

If “.options clock” is specified by the program, an interrupt to location 8 is generated every 10,000 cycles. (Remember though that interrupts are disabled until the program enters user mode – see section 6.3 of the Beta documentation.)