ISTD 50.002 Computation Structures

Zhang Yue
Oka Kurniawan
Wei Lu

Original creator: Lozano-Perez, Tomas, using materials originally developed by Christopher J. Terman and Stephen A Ward. Course materials for ISTD 103, Computation Structures. MIT-SUTD Collaboration, 2012.

Modified by: Oka Kurniawan, 2013.

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

ISTD 50.002 Computation Structures
BSim

Introduction to BSim

BSim is a simulator for the 50.002 Beta architecture. 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 in Linux or Mac OS X, type

```
$ 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 Display 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 display. Very useful for following your program's operation instruction-by-instruction.

Toggle visualization between the programmer's panel (the default) and the animated datapath.

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. Clicking the mouse will cause an interrupts to location 16; CLICK() can be used to fetch the coordinates of the last click into R0. The coordinates are encoded as (x<<16)+y, or -1 if there has been no mouse click since the last call to CLICK().

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.)

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
                                  . = 0x1000
  1000: 09 04 01 00               sqrs:  0 1 4 9
  1004: 31 24 19 10                      16 25 36 49
  1008: 79 64 51 40                      64 81 100 121
  100C: E1 C4 A9 90                      144 169 196 225
  1010: 00 00 00 10               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 symbol definitions for all the registers (R0, …, R31, BP, LP, SP, XP, r0, …, r31, bp, lp, sp, xp) and macro definitions for all the Beta instructions:

`OP(Ra,Rb,Rc)`	Reg[Rc] ← Reg[Ra] op Reg[Rb]
Opcodes:	**ADD**, **SUB**, **MUL**, **DIV**, **AND**, **OR**, **XOR**
	CMPEQ, **CMPLT**, **CMPLE**, **SHL**, **SHR**, **SRA**
`OPC(Ra,literal,Rc)`	Reg[Rc] ← Reg[Ra] op SEXT($literal_{15:0}$)
Opcodes:	**ADDC**, **SUBC**, **MULC**, **DIVC**, **ANDC**, **ORC**, **XORC**
	CMPEQC, **CMPLTC**, **CMPLEC**, **SHLC**, **SHRC**, **SRAC**
`LD(Ra,literal,Rc)`	Reg[Rc] ← Mem[Reg[Ra] + SEXT(literal)]
`ST(Rc,literal,Ra)`	Mem[Reg[Ra] + SEXT(literal)] ← Reg[Rc]
`JMP(Ra,Rc)`	Reg[Rc] ← PC + 4; PC ← Reg[Ra]
`BEQ/BF(Ra,label,Rc)`	Reg[Rc] ← PC + 4; if Reg[Ra] = 0 then PC ← PC + 4 + 4*SEXT(literal)
`BNE/BT(Ra,label,Rc)`	Reg[Rc] ← PC + 4; if Reg[Ra] ≠ 0 then PC ← PC + 4 + 4*SEXT(literal)
`LDR(Ra,label,Rc)`	Reg[Rc] ← Mem[PC + 4 + 4*SEXT(literal)]

Also included are some convenience macros:

```
LD(label,Rc)        expands to LD(R31,label,Rc)
ST(Ra,label)        expands to ST(Ra,label,R31)
BR(label)           expands to BEQ(R31,label,R31)
CALL(label)         expands to BEQ(R31,label,LP)
RTN()               expands to JMP(LP)
DEALLOCATE(n)       expands to SUBC(SP,n*4,SP)
MOVE(Ra,Rc)         expands to ADD(Ra,R31,Rc)
CMOVE(literal,Rc)   expands to ADDC(R31,literal,Rc)
PUSH(Ra)            expands to ADDC(SP,4,SP) ST(Ra,-4,SP)
POP(Rc)             expands to LD(SP,-4,Rc)  ADDC(SP,-4,SP)

HALT()              cause the simulator to stop execution
```

The following is a complete example assembly language program:

```
.include /50002/beta.uasm

. = 0                       | start assembling at location 0
    LD(input,r0)            | put argument in r0
    CALL(bitrev)            | call the procedure (= BR(bitrev,r28))
    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
    RTN()                   | return to caller  (= JMP(r28))

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

The BSim assembly language processor includes 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
noclk	disable clock interrupts (default)
div	simulate the DIV instruction (default)
nodiv	make the DIV opcode an illegal instruction
mul	simulate the MUL instruction (default)
nomul	make the MUL opcode an illegal instruction
kalways	don't let program enter user mode (ie, supervisor bit is always 1)
nokalways	allow program to enter user mode (default)
tty	enable RDCHAR(), WRCHAR(), CLICK() (see end of first section)
notty	RDCHAR(), WRCHAR(), CLICK() are disabled (default)
annotate	if BP is non-zero, label stack frames in the programmer's panel
noannotate	don't annotate stack frames (default)

`.pcheckoff …`
`.tcheckoff …`
`.verify …`

Supply checkoff information to the simulator.