edX

MITx 6.004.2x
Computation Structures 2: Computer Architecture

Help

Course / Overview / BSim Documentation

Previous

Next

BSim Documentation

🔖 Bookmarked

Calculator

Introduction to BSim

BSim is a simulator for the 6.004 Beta architecture. BSim provides 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.

BSim incorporates CodeMirror for editing your .uasm files. CodeMirror supports many common edit operations — please see the table at the end of this document for the mapping from keystrokes to edit operations. Whenever you click Check & Save the contents of your edit buffers will be saved on the server. Use the tabs in the editor window to select a particular buffer to edit. Tabs marked with ⊘ are read-only -- you can view their contents but not make any modifications.

To test your code, click on Assemble in the editor's toolbar. This will 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 pane showing the Beta datapath is displayed from which you can start execution of the program.

The Simulation pane 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 and memory are updated at the end of the cycle.

⏸	Pause execution.
⏮	Reset the contents of the PC and registers to 0, and memory locations to the values they had just after assembly was complete.
▶	Start simulation and run until a HALT() instruction is executed or a breakpoint is reached. You can stop a running simulation using the pause control described above. The display will be updated every cycle.
⏩	The same as the play button described above, but only updates the display once every 25,000 cycles for maximum simulation speed.
⏭	Execute the program for a single cycle and then update the display. Very useful for following your program's operation instruction-by-instruction.
⏮	Undo the last cycle and update the display. Very useful after clicking the single cycle button more times than intended. This button can only go back a limited number of steps.
Schematic View	Display the animated datapath in place of the programmer's panel.
Programmer's View	Display the programmer's panel in place of the animated datapath.

To switch between the editor and simulation panes, use the "Editor" and "Simulation" buttons at the top left of the window. "Split" will divide the screen evenly between the two. It is also possible to drag the dividers between panes to resize them individually. When both the editor and simulation panes are visible, clicking "Assemble" will not alter the layout of the window.

If `.options tty` is specified by the program, the small typeout window at the bottom of the simulation pane may be accessed by the running program. 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 a double slash and end at a newline
/* Multi-line comments are also available, and continue until
    reaching the termination sequence, */

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

Operator	Operation
−	unary minus
~	bit-wise complement
&	bit-wise AND
|	bit-wise OR
+	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)
```

which 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) Opcodes:	Reg[Rc] ← Reg[Ra] op Reg[Rb] ADD, SUB, MUL, DIV, AND, OR, XOR CMPEQ, CMPLT, CMPLE, SHL, SHR, SRA
OPC(Ra,literal,Rc) Opcodes:	Reg[Rc] ← Reg[Ra] op SEXT(literal15:0) 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(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 "/shared/bsim/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 "buffer_name"`

Process the text found in the specified buffer at this point in the assembly. The buffer name must be given as a string.

`.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 and the next byte assembled will be word-aligned.

`.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

📊 Calculator

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.

CodeMirror Editor Key Bindings

Move	PC keystroke	Mac keystroke
Left one character	Left	Left
Right one character	Right	Right
Up one line	Up	Up
Down one line	Down	Down
Beginning of line	Home, Alt-Left	Home, Cmd-Left
End of line	End, Alt-Right	End, Cmd-Right
Up one page	Page up	Page up
Down one page	Page down	Page down
Beginning of document	Ctrl-Home, Alt-Up	Cmd-Up
End of document	Ctrl-End, Ctrl-Down	Cmd-End, Cmd-Down
Left one group	Ctrl-Left	Alt-Left
Right one group	Ctrl-Right	Alt-Right
Selection and delete		
Select all	Ctrl-A	Cmd-A
Delete character before cursor	Backspace	Backspace
Delete character after cursor	Delete	Delete
Delete group before cursor	Ctrl-Backspace	Alt-Backspace
Delete group after cursor	Ctrl-Delete	Alt-Delete, Ctrl-Alt-Backspace
Delete line	Ctrl-D	Cmd-D
Search and replace		
Find	Ctrl-F	Cmd-F
Find next	Ctrl-G	Cmd-G

Find previous	Shift-Ctrl-G	Shift-Cmd-G
Replace	Shift-Ctrl-F	Cmd-Alt-F
Replace all	Shift-Ctrl-R	Shift-Cmd-Alt-F
Miscellaneous		
Undo	Ctrl-Z	Cmd-Z
Redo	Ctrl-Y, Shift-Ctrl-Z	Cmd-Y, Shift-Cmd-Z
Indent more	Ctrl-]	Cmd-]
Indent less	Ctrl-[	Cmd-[
Toggle overwrite	Insert	Insert

Discussion

Hide Discussion

Topic: Overview / BSim Documentation

Add a Post

| Show all posts ⌄ | by recent activity ⌄ |

💬 typo - Programmer's View button explanation

The explanation of the Programmer's View button is reversed.

2

💬 [STAFF] LDR macro

There is a convenience macro for LD(label, Rc) which corresponds to LD(R31, label, Rc). LDR is listed as LDR(Ra, label, Rc) but trans...

5

| ‹ Previous | Next › |

edX

edX

About

Affiliates

edX for Business

Open edX

Careers

News

Legal

Terms of Service & Honor Code

Privacy Policy

Accessibility Policy

Trademark Policy

Sitemap

Cookie Policy

Your Privacy Choices

Connect

Idea Hub

Contact Us

Help Center

Calculator

Security

Media Kit

深圳市恒宇博科技有限公司 粤ICP备17044299号-2

Calculator