The BigHex Machine

David May: October 20, 2016

Background

The BigHex machine is specifically designed as a very simple computer suitable for explaining how a computer works. Further, its instruction set requires a very small compiler, but is powerful enough to implement useful programs. The main features of the instruction set are:

- Short instructions are provided to allow efficient access to regions in memory allocated by compilers including a procedure calling stack; these also provide efficient branching and subroutine calling.
- The memory is addressed as a collection of 16-bit words; however the instructions are all 8-bit so instruction addresses refer to a specific byte position within a word.
- The processor has a small number of registers. Some registers are used for specific purposes such as accessing the program or building large constants.
- Instructions are easy to decode.

All instructions are 8-bit; each instruction contains 4 bits representing an operation and 4 bits of an immediate operand. A special instruction, OPR causes its operand to be interpreted as an inter-register operation. Instruction prefixes are used to extend the range of immediate operands and to provide more inter-register operations.

The prefixes are:

- PFIX which concatenates its 4-bit immediate operand with the 4-bit immediate operand of the next 8-bit instruction.
- NFIX which complements its its 4-bit immediate operand and then concatenates the result with the 4-bit immediate operand of the next 8-bit instruction.

The prefixes are inserted automatically by compilers and assemblers.

The normal state of a processor is represented by 4 registers. Two of the registers are used to hold the sources and destination of arithmetic and logic operations. Another (the operand register) is used to accumulate the operands of the prefixes.

register	**use**
pc	the program counter
$oreg$	the operand register
$areg$	left-hand operand and result of arithmetic
$breg$	right-hand operand of arithmetic

Instruction Issue and Execution

The instruction set has only twenty instructions and allows a very simple design.

The main components are:

- The registers.
- The A multiplexor, which selects one of $areg$, pc, $oreg$ and zero.
- The B mutiplexor, which selects one of $breg$, $oreg$ and zero.
- The arithmetic unit, which combines the operands selected by the A and B multiplexors.
- The memory, which takes addresses from the arithmetic unit output and data from $areg$.
- The result multiplexor, which selects either the memory data output or the arithmetic unit output; this multiplexor output is supplied to the registers.
- The instruction register, decoder and control matrix.
- The clock and timing generator.

Each instruction is executed in three stages: the instruction is fetched; the pc is incremented; the instruction is executed.

Instruction set Notation and Definitions

In the following description

mem	represents the memory
$port$	represents a set of input-output ports
$display$	represents a display consisting of 16 rows each of 16 bits
pc	represents the program counter
$oreg$	represents the operand register
$areg$	represents the left-hand operand register
$breg$	represents the right-hand operand register
row	represents the row of the diplay next to be updated
$u4$	is a 4-bit unsigned source operand in the range $[0 : 15]$

Data access

The data access instructions fall into several groups. One of these provides access via the stack pointer.

LDAM	$areg \leftarrow mem[oreg]$	load from memory
LDBM	$breg \leftarrow mem[oreg]$	load from memory
STAM	$mem[oreg] \leftarrow areg$	store to memory

Access to constants and program addresses is provided by instructions which either load values directly or enable them to be loaded from a location in the program:

LDAC	$areg \leftarrow oreg$	load constant
LDBC	$breg \leftarrow oreg$	load constant
LDAP	$areg \leftarrow pc + oreg$	load address in program

Access to data structures is provided by instructions which combine an address with an offset:

LDAI	$areg \leftarrow mem[areg + oreg]$	load from memory
LDBI	$breg \leftarrow mem[breg + oreg]$	load from memory
STAI	$mem[breg + oreg] \leftarrow areg$	store to memory

Expression Evaluation

Expressions are evaluated using the ADD and SUB instructions.

ADD	$areg \leftarrow areg + breg$	add
SUB	$areg \leftarrow areg - breg$	subtract

Branching, jumping and calling

The branch instructions include conditional and unconditional relative branches. A branch using an offset in the stack is provided to support jump tables.

BR	$pc \leftarrow pc + oreg$	branch relative unconditional
BRZ	if $areg = 0$ then $pc \leftarrow pc + oreg$	branch relative zero
BRN	if $areg < 0$ then $pc \leftarrow pc + oreg$	branch relative negative
BRB	$pc \leftarrow breg + oreg$	branch absolute
SVC	system call	

To call a procedure, the return address can be loaded using the LDAP instruction and the BR instruction can be used to branch to the procedure entrypoint. The procedure entry will store the return address; the exit will load this return address into $breg$ and use a BRB instruction to branch back to the calling procedure.

Input and Output

Data can be transferred to an from the input-output ports uing IN and OUT instructions.

IN	$areg \leftarrow port[areg]; oreg = 0;$	input from port
OUT	$port[breg] \leftarrow areg; oreg = 0;$	output to port

The BigHex machine has four illuminated buttons connected to port 0 and another four connected to port 1. Each button corresponds to a single bit of the word being input or output.

position	colour	port	bit
left	blue	0	0
left	green	0	1
left	red	0	2
left	yellow	0	3
right	blue	1	0
right	green	1	1
right	red	1	2
right	yellow	1	3

There is also a display consisting of 16 rows each of 16 pixels; the rows are illuminated in sequence by copying words from the top 16 locations in memory.

Instruction summary

LDAM	$areg \leftarrow mem[oreg]$	load from memory
LDBM	$breg \leftarrow mem[oreg]$	load from memory
STAM	$mem[oreg] \leftarrow areg$	store to memory
LDAC	$areg \leftarrow oreg$	load constant
LDBC	$breg \leftarrow oreg$	load constant
LDAP	$areg \leftarrow pc + oreg$	load address in program
LDAI	$areg \leftarrow mem[areg + oreg]$	load from memory
LDBI	$breg \leftarrow mem[breg + oreg]$	load from memory
STAI	$mem[breg + oreg] \leftarrow areg$	store to memory
BR	$pc \leftarrow pc + oreg$	branch relative unconditional
BRZ	if $areg = 0$ then $pc \leftarrow pc + oreg$	branch relative zero
BRN	if $areg < 0$ then $pc \leftarrow pc + oreg$	branch relative negative
BRB	$pc \leftarrow breg + oreg$	branch absolute
ADD	$areg \leftarrow areg + breg$	add
SUB	$areg \leftarrow areg - breg$	subtract
IN	$areg \leftarrow port[areg]$	input from port
OUT	$port[breg] \leftarrow areg$	output to port

C Simulator

The following C program simulates the BigHex computer. It includes the instructions to perform input-output, representing the ports as an array. Similarly, it includes transfers to the display. It can easily be extended as appropriate to transfer the contents of the ports to and from a host operating system, and to transfer the contents of the display to a host operating system.

```
#include "stdio.h"

#define true     1
#define false    0

#define i_ldam   0x0
#define i_ldbm   0x1
#define i_stam   0x2

#define i_ldac   0x3
#define i_ldbc   0x4
#define i_ldap   0x5

#define i_ldai   0x6
#define i_ldbi   0x7
#define i_stai   0x8

#define i_br     0x9
#define i_brz    0xA
#define i_brn    0xB
#define i_brb    0xC

#define i_opr    0xD
#define i_pfix   0xE
#define i_nfix   0xF

#define o_add    0x0
#define o_sub    0x1
#define o_in     0x2
#define o_out    0x3
```

```
FILE *codefile;

unsigned short mem[32768];
unsigned char *pmem = (unsigned char *) mem;

unsigned char port[4];
unsigned short display[16];

unsigned short pc;
unsigned short areg;
unsigned short breg;
unsigned short row;
unsigned short oreg;

unsigned char inst;

unsigned int running;
```

```
main()
{ load();
  running = true; oreg = 0; pc = 0; row = 0;

  while (running)
  { inst = pmem[pc];
    pc = pc + 1;
    oreg = oreg | (inst & 0xf);

    switch ((inst >> 4) & 0xf)
    { case i_ldam: areg = mem[oreg]; oreg = 0; break;
      case i_ldbm: breg = mem[oreg]; oreg = 0; break;
      case i_stam: mem[oreg] = areg; oreg = 0; break;

      case i_ldac: areg = oreg; oreg = 0; break;
      case i_ldbc: breg = oreg; oreg = 0; break;
      case i_ldap: areg = pc + oreg; oreg = 0; break;

      case i_ldai: areg = mem[areg + oreg]; oreg = 0; break;
      case i_ldbi: breg = mem[breg + oreg]; oreg = 0; break;
      case i_stai: mem[breg + oreg] = areg; oreg = 0; break;

      case i_br:  pc = pc + oreg; oreg = 0; break;
      case i_brz: if (areg == 0) pc = pc + oreg; oreg = 0;break;
      case i_brn: if ((int)areg < 0) pc = pc + oreg; oreg = 0;break;
      case i_brb: pc = pc + oreg; break;

      case i_pfix: oreg = oreg << 4; break;
      case i_nfix: oreg = 0xFFFFFF00 | (oreg << 4); break;
      case i_opr:
         switch (oreg)
         { case o_add:    areg = areg + breg; break;
           case o_sub:    areg = areg - breg; break;
           case o_in:     areg = port[areg]; break;
           case o_out:    port[breg] = areg; break;
         };
         oreg = 0; break;
    };
    display[row] = mem[0x7ff0 + row]; row = (row + 1) & 0xf;
  }
}
```

```
load()
{ codefile = fopen("sim2", "rb");
  loadmem(0);
  codefile = fopen("sim3", "rb");
  loadmem(1);
}

loadmem(n)
{ char ch;
  int addr;
  addr = n;
  ch = fgetc(codefile);
  while ((('0' <= ch) && (ch <= '9')) ||
         (('A' <= ch) && (ch <= 'F')))
  { pmem[addr] = ((hexval(ch) << 4) | hexval(fgetc(codefile)));
    ch = fgetc(codefile); ch = fgetc(codefile);
    addr = addr + 2;
  }
};

hexval(ch)
{ int v;
  if (('0' <= ch) && (ch <= '9'))
    v = ch - '0';
  else
    v = (ch - 'A') + 10;
  return v;
}
```