

The Hex 8 Architecture

David May: February 21, 2014

Background

The architecture described here is specifically designed as a very simple processor suitable for explaining how a computer works. It can be programmed using the X language. The main features of the instruction set are:

- Short instructions are provided to allow efficient access to the stack and other data regions allocated by compilers; these also provide efficient branching and subroutine calling.
- The memory consists of 256 bytes and the processor registers are one byte long and each instruction is one byte long.
- 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 immediate data. A special instruction, OPR causes its operand to be interpreted as an inter-register operation. An instruction prefix is used to extend the range of immediate operands.

The prefix instruction is:

- PFIX which concatenates its 4-bit immediate with the 4-bit immediate 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 sixteen 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

pc represents the program counter

oreg represents the operand register

areg represents the left-hand operand register

breg represents the right-hand operand register

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

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 $pc \leftarrow breg$ branch absolute

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.

Expression evaluation

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

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	$pc \leftarrow breg$	branch absolute
ADD	$areg \leftarrow areg + breg$	add
SUB	$areg \leftarrow areg - breg$	subtract

```
#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_add     0xD
#define i_sub     0xE

#define i_pfix    0xF

FILE *codefile;

unsigned char mem[256];
unsigned char *pmem = (unsigned char *) mem;

unsigned char pc;
unsigned char areg;
unsigned char breg;
unsigned char oreg;

unsigned char inst;

unsigned int running;
```

```
main()
{
    load();

    running = true;
    oreg = 0;
    pc = 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:   stop(); pc = pc + oreg; oreg = 0; break;
          case i_brz:  if (areg == 0) pc = pc + oreg; oreg = 0; break;
          case i_brn:  if (areg > 127) pc = pc + oreg; oreg = 0; break;

          case i_brb:  pc = breg; oreg = 0; break;
          case i_add:  areg = areg + breg; oreg = 0; break;
          case i_sub:  areg = areg - breg; oreg = 0; break;

          case i_pfix: oreg = oreg << 4; break;
      };
    }
}
```

```
load()
{ int n; int i;
  codefile = fopen("a.bin", "rb");
  n = getbyte();
  i = 0;
  while (n != EOF)
  { pmem[i] = n;
    n = getbyte();
    i = i + 1;
  };
}

getbyte()
{ int high;
  high = gethex();
  if (high == EOF)
    return EOF;
  else
    return (high << 4) | gethex();
}

gethex()
{ int h;
  h = fgetc(codefile);
  while ((h == ' ') || (h == '\n'))
    h = fgetc(codefile);
  if (h == EOF)
    return EOF;
  else
    if ((h >= 'A') && (h <= 'F'))
      return (h - 'A') + 10;
    else
      return h - '0';
}

stop()
{ if (oreg == 0xFE)
  { printf("\nareg = %d\n", areg);
    running = false;
  }
}
```