The Hex 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. Further, its instruction set requires a very small compiler, but is powerful enough to implement substantial programs including a self-compiling (bootstrapping) compiler for 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 is word addressed; however the instructions are all single byte so instruction addresses refer to a specific byte position within a word.
- The same instruction set can be used for processors with different wordlengths; the only requirement is that the wordlength is a number of bytes.
- 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. 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 with the 4-bit immediate of the next 8-bit instruction.
- NFIX which complements its its 4-bit immediate and then concatenates the result 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
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.

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
SVC		system call

```
#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_opr     0xD
#define i_pfix    0xE
#define i_nfix    0xF

#define o_brb     0x0
#define o_add     0x1
#define o_sub     0x2
#define o_svc     0x3
```

```
FILE *codefile;

FILE *simio[8];

char connected[] = {0, 0, 0, 0, 0, 0, 0, 0};

unsigned int mem[200000];
unsigned char *pmem = (unsigned char *) mem;

unsigned int pc;
unsigned int sp;
unsigned int areg;
unsigned int breg;
unsigned int oreg;

unsigned int 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:  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_pfix: oreg = oreg << 4; break;
      case i_nfix: oreg = 0xFFFFFF00 | (oreg << 4); break;
      case i_opr:
         switch (oreg)
         { case o_brb:    pc = breg; oreg = 0; break;
           case o_add:    areg = areg + breg; oreg = 0; break;
           case o_sub:    areg = areg - breg; oreg = 0; break;
           case o_svc:  svc(); break;
          };
          oreg = 0; break;
    };
  }
}
```

```
load()
{ int low;
  int length;
  int n;
  codefile = fopen("a.bin", "rb");
  low = inbin();
  length = ((inbin() << 16) | low) << 2;
  for (n = 0; n < length; n++)
    pmem[n] = fgetc(codefile);
};

inbin(d)
{ int lowbits;
  int highbits;
  lowbits = fgetc(codefile);
  highbits = fgetc(codefile);
  return (highbits << 8) | lowbits;
};

svc()
{ sp = mem[1];
  switch (areg)
  { case 0: running = false; break;
    case 1: simout(mem[sp + 2], mem[sp + 3]); break;
    case 2: mem[sp + 1] = simin(mem[sp + 2]) & 0xFF; break;
  }
}
```

```
simout(b, s)
{ char fname[] = {’s’, ’i’, ’m’, ’ ’, 0};
  int f;
  if (s < 256)
    putchar(b);
  else
  { f = (s >> 8) & 7;
    if (! connected[f])
    { fname[3] = f + ’0’;
       simio[f] = fopen(fname, "w");
       connected[f] = true;
      };
      fputc(b, simio[f]);
  };
};

simin(s)
{ char fname[] = {’s’, ’i’, ’m’, ’ ’, 0};
  int f;
  if (s < 256)
    return getchar();
  else
  { f = (s >> 8) & 7;
    fname[3] = f + ’0’;
    if (! connected[f])
    { simio[f] = fopen(fname, "r");
      connected[f] = true;
    };
    return fgetc(simio[f]);
  };
};
```