S2 Instruction Set (part2)

There are 4 groups of instructions:

1 **alu&logic**
```
add sub mul div
and or xor not shl shr
eq ne lt le gt ge
```

2 **control flow**
```
jmp jt jf jal ret
```

3 **data**
```
ld st mv push pop
```

4 **other**
```
trap
```

Form of instruction (number of arguments)

1 **alu&logic**
Most instructions are binary operators except "not" (unary).

```
dest = src1 op src2
```

It operates with registers and src2 may be a constant (number, called immediate value), for example,

```
add r1 r2 r3     is   R[r1] = R[r2] + R[r3]
add r1 r2 #10    is   R[r1] = R[r2] + 10
```

The name of register are r0..r31. r0 is always 0. Writing any value to r0 does not change it. The constant is an integer (17 bits) prefixed with "#". It may be a negative number, for example #-1.

"div" is an integer division (no fractional part). and, or, xor, shl, shr are bit-wise instructions.

```
shl r1 r2 #2     is  R[r1] = R[r2] shift left by 2 bits
```

The logical instructions: eq, ne... produce boolean result, the value true/false where false is 0, and true is not 0.

2 **control flow**
These instructions affect the sequence of execution of instructions by changing the program counter (PC).

```
jmp ads          goto ads
jt r1 ads        if r1 != 0 goto ads
jf r1 ads        if r1 == 0 goto ads

jal r1 ads       R[r1] = PC; goto ads
```

jal is a call to a subroutine. It saves the current program counter (which is now pointed to the place to continue when the subroutine is returned) in R[r1] which is called "link register".

```
  ret r1
```
jump back to the address stored in the link register (R[r1]).

3 **data**

Move the content between registers and memory. The calculation of the desired address of the memory (effective address) is called "addressing mode".

```
      reg   <-- ld --    memory
            --- st -->

      reg   <-- mv -->   reg
```

3.1 We will discuss ld/st first

1 absolute addressing mode -- the effective address is taken directly from the constant in the instruction (size 22 bits).

```
  ld r1 ads         R[r1] = M[ads]
  st ads r1         M[ads] = R[r1]
```

2 indirect addressing mode -- the effective address is calculated from a value in one register (that is why it is called "indirect") and a constant in the instruction (from the field "disp").

```
  ld r1 @d r2       R[r1] = M[ d + R[r2] ]
  st @d r2 r1       M[ d + R[r2] ] = R[r1]
```

3 index addressing mode -- the effective address is calculated from two registers (one is base address, another is an index).

```
  ld r1 +r2 r3      R[r1] = M[ R[r2] + R[r3] ]
  st +r2 r3 r1      M[ R[r2] + R[r3] ] = R[r1]
```

3.2 mv transfer data between registers, or set a value of a register (a constant is 22 bits).

```
  mv r1 r2          R[r1] = R[r2]
  mv r1 #n          R[r1] = n
```

3.3 push/pop transfter data between a stack data structure resided in a memory and a register.

```
  push r1 r2        R[r1]++; M[ R[r1] ] = R[r2]
```

Take a value in R[r2] and store it in a stack pointed to by R[r1]. R[r1] is called a stack pointer.

```
  pop r1 r2         R[r2] = M[ R[r1] ]; R[r1]--
```
Take a value from a stack (let R[r1] be the stack pointer) and store it to R[r2].

4 **other**

trap is an instruction that interface to the operating system functions such as: stopping the execution, print a number onto the screen, etc.

```
  trap 0            stop the execution
  trap 1            print R[30] to screen as an integer
  trap 2            print R[30] to screen as a character (ASCII)
```

Encoding of instructions

All instructions are 32 bits. There are 3 types of encoding: L, D, X.

```
L  op:5 | r1:5 |          ads:22        |
D  op:5 | r1:5 | r2:5  |    disp:17     |
X  op:5 | r1:5 | r2:5  | r3:5 | xop:12  |
```

The format L is used with the control flow instructions and ld/st in absolute addressing mode where they need 22-bit argument. The format D is used with the instruction which require an immediate value. Lastly, the format X is used for the instructions that require two register arguments. The format X's opcode has two parts, the first part, op, is 31, the second part is in the field xop (extended opcode).

1 **alu&logic**

```
   reg-imm      reg-reg
   fmt op      fmt op xop
add D 10    add X 31 0
sub D 11    sub X 31 1
mul D 12    mul X 31 2
div D 13    div X 31 3
and D 14    and X 31 4
or  D 15    or  X 31 5
xor D 16    xor X 31 6
shl D 17    shl X 31 7
shr D 18    shr X 31 8
eq  D 19    eq  X 31 9
ne  D 20    ne  X 31 10
lt  D 21    lt  X 31 11
le  D 22    le  X 31 12
gt  D 23    gt  X 31 13
ge  D 24    ge  X 31 14
            not X 31 22
```

2 **control flow**

```
   fmt op xop
jmp L 6
jal L 7
jt  L 8
jf  L 9
ret X 31 18
```

3 **data**

```
        format op            fmt op           fmt op xop

ld absolute L  1    ld indirect D 2    ld index X  31 16
st absolute L  3    st indirect D 4    st index X  31 17

   reg-imm    reg-reg
  fmt op     fmt op xop
mv L  5     mv X 31  15

push X 31 20
pop  X 31 21
trap X 31 19
```

Example

```
add r1 r2 r3            | 31 | 1  | 2  |  3 |   0     |
add r1 r2 #8            | 10 | 1  | 2  |     8        |
jt r4 100               | 8  | 4  |      100          |
ld r1 2000              | 1  | 1  |      2000         |
ld r1 @30 r2            | 2  | 1  | 2  |     30       |
ld r1 +r2 r3            | 31 | 1  | 2  |  3 |   16    |
mv r1 r2                | 31 | 1  | 2  |  0 |   15    |
mv r1 #40               | 5  | 1  |      40           |
push r1 r2              | 31 | 2  | 3  |  0 |   20    |
```

You can see the example of instruction encoding from the listing file.

Prabhas Chongstitvatana
5 February 2007