S21 A Hypothetical 32-bit Processor

This is a typical simple 32-bit CPU. It has a three-address instruction architecture, with 32 registers and load/store instructions. This document presents only S21 assembly language view. It does not give details about microarchitecture (such as pipeline).

S21 has three-address instruction set. A general format of an instruction (register to register operations) is:

```
op r1 r2 r3     means  R[r1] = R[r2] op R[r3]
```

To pass values between memory and registers, load/store instructions are used. There are three addressing mode: absolute, indirect and index.

load means R <- mem
store means mem <- R

```
ld r1,ads          R[r1] = M[ads]           absolute
ld r1,d(r2)        R[r1] = M[d+R[r2]]       indirect
ld r1,(r2+r3)      R[r1] = M[R[r2]+R[r3]]   index
st r1,ads          M[ads] = R[r1]           absolute
st r1,d(r2)        M[d+R[r2]] = R[r1]       indirect
st r1,(r2+r3)      M[R[r2]+R[r3]] = R[r1]   index
```

In assembly language, these addressing modes are written as (using the convention `op dest <- source`):

```
ld r1,ads          ld r1 ads
ld r1,d(r2)        ld r1 @d r2
ld r1,(r2+r3)      ld r1 +r2 r3
st r1,ads          st ads r1
st r1,d(r2)        st @d r2 r1
st r1,(r2+r3)      st +r2 r3 r1
```

There are three flags: Zero, Carry, Overflow/underflow.

Instruction type

```
arith & logic:  add sub mul div  and or xor eq ne lt le gt ge shl shr
control flow:   jmp jt jf jal ret
data:           ld st mv push pop
```

Instruction meaning

false == 0
true != 0
R[0] always zero

```
op r1 r2 r3      is    R[r1] = R[r2] op R[r3]
op r1 r2 #n      is    R[r1] = R[r2] op n

ld r1,ads        is    R[r1] = M[ads]           absolute
ld r1,d(r2)      is    R[r1] = M[d+R[r2]]       indirect
ld r1,(r2+r3)    is    R[r1] = M[R[r2]+R[r3]]   index
st r1,ads        is    M[ads] = R[r1]           absolute
st r1,d(r2)      is    M[d+R[r2]] = R[r1]       indirect
st r1,(r2+r3)    is    M[R[r2]+R[r3]] = R[r1]   index
```

```
jmp ads          is    pc = ads
jt r1 ads        is    if R[r1] != 0  pc = ads
jf r1 ads        is    if R[r1] == 0  pc = ads
jal r1,ads       is    R[r1] = PC; PC = ads
                       // jump and link, to subroutine
ret r1           is    PC = R[r1]
                       // return from subroutine

mv r1 r2         is    R[r1] = R[r2]
mv r1 #n         is    R[r1] = #n
push r1 r2       is    R[r1]++; M[R[r1]] = R[r2]
                       // push r2, r1 as sp
pop  r1 r2       is    R[r2] = M[R[r1]]; R[r1]--
                       // pop to r2, r1 as sp
```

arithmetic

two-complement integer arithmetic

```
add r1,r2,r3      R[r1] = R[r2] + R[r3]
add r1,r2,#n      R[r1] = R[r2] + sign extended n (n is 17 bits)
```

add, sub affect Z,C -- C indicates carry (add) or borrow (sub)
mul, div affect Z,O -- O indicates overlow (mul) or underflow (div) and divide by zero

logic (bitwise)

```
and r1,r2,r3      R[r1] = R[r2] bitand R[r3]
and r1,r2,#n      R[r1] = R[r2] bitand sign extended n
. . .
eq r1,r2,r3       R[r1] = R[r2] == R[r3]
eq r1,r2,#n       R[r1] = R[r2] == #n
. . .
shl r1,r2,r3      R[r1] = R[r2] << R[r3]
shl r1,r2,#n      R[r1] = R[r2] << #n
. . .
```

affect Z,S bits

```
trap n            special instruction, n is in r1-field.
```

trap 0 stop simulation
trap 1 print integer in R[30]
trap 2 print character in R[30]

stack operation

To facilitate passing the parameters to a subroutine and also to save state (link register) for recursive call, two stack operations are defined: push, pop. r1-field is used as a stack pointer.

Instruction format

```
L-format    op:5 rd1:5 ads:22
D-format    op:5 rd1:5 rs2:5 disp:17
X-format    op:5 rd1:5 rs2:5 rs3:5 xop:12
```
(rd dest, rs source, ads and disp sign extended)

Instructions are fixed length at 32-bit. Register set is 32, with R[0] always return zero. The address space is 32-bit, addressing is word. Flags are: Z zero, S sign, C carry, O overflow/underflow.

ISA and opcode encoding

mode:
a - absolute
d - displacement
x - index
i - immediate
r - register
r2 - register 2 operands
s - special 1 operand

opcode op mode format

```
0 nop
1 ld      a      L     ld r1 ads
2 ld      d      D     ld r1 d(r2)
3 st      a      L     st ads r1
4 st      d      D     st d(r2) r1
5 mv      a      L     mv r1 #n (22 bits)
6 jmp     a      L         r1 = 0
7 jal     a      L     jal r1 ads
8 jt      a      L
9 jf      a      L
10 add    i      D     add r1 r2 #n
11 sub    i      D
12 mul    i      D
13 div    i      D
14 and    i      D
15 or     i      D
16 xor    i      D
17 eq     i      D
18 ne     i      D
19 lt     i      D
20 le     i      D
21 gt     i      D
22 ge     i      D
23 shl    i      D
24 shr    i      D
25..30 undefined
31 xop    -      X

xop
0 add     r      X     add r1 r2 r3
1 sub     r      X
2 mul     r      X
3 div     r      X
4 and     r      X
5 or      r      X
6 xor     r      X
```

```
7 eq     r     X
8 ne     r     X
9 lt     r     X
10 le    r     X
11 gt    r     X
12 ge    r     X
13 shl   r     X
14 shr   r     X
15 mv    r2    X    mv r1 r2
16 ld    x     X    ld r1 (r2+r3)
17 st    x     X    st (r2+r3) r1
18 ret   s     X      use r1
19 trap  s     X      use r1 as number of trap
20 push  r2    X        use r1 as stack pointer
21 pop   r2    X        use r1 as stack pointer
22 not   r2    X
23..4095 undefined
```

Historical fact

S21 is an extension of S2 in 2007, as a result of my experience in teaching assembly language. S2 has been used for teaching since 2001.
S2 itself is an "extended" version of S1 (a 16-bit cpu) which is used since 1997.

To improve understandability of S2 assembly language, flags are not used. Instead, new logical instructions that have 3-address are introduced. They are similar to existing arith instructions. The result (true/false) is stored in a register. Two new conditional jumps are introduced "jt", "jf" to make use of the result from logical instructions. Also to avoid the confusion between absolute addressing and moving between registers, a new instruction "mv" is introduced. (and to make ld/st symmetry, "ld r1 #n" is eliminated.)

The opcode format and assembly language format for S2 follow the tradition dest = source1 op source2 from PDP, VAX and IBM S360. As r0
always is zero, many instructions can be synthesis using r0.

```
or r1,r2,r0         move r1 <- r2
or r1,r0,r0         clear r1
sub r0,r1,r2        compare r1 r2  affects flags
```

To complement a register, xor with 0xFFFFFFFF (-1) can be used.

```
xor r1,r2,#-1       r1 = complement r2
```

last update 20 Nov 2010