S3.0 multicore processor

S3.0 is extended from S21. It 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. Load means transfer memory to register. Store means transfer register to memory. There are three addressing modes: absolute, index and displacement.

```
ld r1 ads          R[r1] = M[ads]          absolute
ld r1 +r2 r3       R[r1] = M[R[r2]+R[r3]]  index
ld r1 @d r2        R[r1] = M[d + R[r2]]    displacement
st r1 ads          M[ads] = R[r1]          absolute
st r1 +r2 r3       M[R[r2]+R[r3]] = R[r1]  index
st r1 @d r2        M[d + R[r2]] = R[r1]    displacement
```

In assembly language, these addressing modes are written using the convention op dest <- source.

Instruction type

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

Instruction meaning

false == 0
true != 0

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

Data

d is 17-bit sign extended
m is 22-bit sign extended

```
ld r1 ads         R[r1] = M[ads]          absolute
ld r1 +r2 r3      R[r1] = M[R[r2]+R[r3]]  index
ld r1 @d r2       R[r1] = M[d + R[r2]]    displacement
st r1 ads         M[ads] = R[r1]          absolute
st r1 +r2 r3      M[R[r2]+R[r3]] = R[r1]  index
st r1 @d r2       M[d + R[r2]] = R[r1]    displacement
```

```
mov r1 #m        R[r1] = m                move constant to register
mov r1 r2        R[r1] = R[r2]            move value between registers
```

Control

```
jmp ads          pc = ads
jt r1 ads        if R[r1] != 0  pc = ads    jump if r1 is true
jf r1 ads        if R[r1] == 0  pc = ads    jump if r1 is false
jal r1 ads       R[r1] = PC; PC = ads       jump and link
ret r1           PC = R[r1]                 return
```

Interrupt

```
int #n           R[31] = PC, PC = M[1000+4*n] sw interrupt
reti             PC = R[31]                   return from int
wfi              R[31] = PC, stop             wait for int
```

Arithmetic

two-complement integer arithmetic
n is 17-bit sign extended

```
add r1 r2 r3      R[r1] = R[r2] + R[r3]
add r1 r2 #n      R[r1] = R[r2] + n
sub r1 r2 ...     R[r1] = R[r2] - R[r3]
mul r1 r2 ...     R[r1] = R[r2] * R[r3]
div r1 r2 ...     R[r1] = R[r2] / R[r3]  integer division
mod r1 r2 ...     R[r1] = R[r2] % R[r3]  modulo
```

Logic (bitwise)

n is 17-bit sign extended

```
and r1 r2 r3      R[r1] = R[r2] bit-and R[r3]
and r1 r2 #n      R[r1] = R[r2] bit-and n
or r1 r2 r3       R[r1] = R[r2] bit-or  R[r3]
or r1 r2 #n       R[r1] = R[r2] bit-or  n
xor r1 r2 r3      R[r1] = R[r2] bit-xor R[r3]
xor r1 r2 #n      R[r1] = R[r2] bit-xor n
eq r1 r2 r3       R[r1] = R[r2] == R[r3]
eq r1 r2 #n       R[r1] = R[r2] == n
ne ...
lt ...
le ...
gt ...
ge ...
shl r1 r2 r3      R[r1] = R[r2] << R[r3]
shl r1 r2 #n      R[r1] = R[r2] << n
shr r1 r2 r3      R[r1] = R[r2] >> R[r3]
shr r1 r2 #n      R[r1] = R[r2] >> n
```

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.

```
push r1 r2     R[r1]++; M[R[r1]] = R[r2]      push r2, r1 as stack pointer
pop  r1 r2     R[r2] = M[R[r1]]; R[r1]--      pop to r2, r1 as stack pointer
pushm r1       push multiple R[0]..R[15], r1 as stack pointer
popm  r1       pop multiple R[0]..R[15], r1 as stack pointer
```

Multicore support

```
cid r1         return core number to R[r1]
intx #c        generate int0 to core #c
sync           global (all-core) synchronisation
```

Pseudo

Pseudo instructions are unlike other instructions, they are used mainly to control the simulator and to perform input/output. "trap" instruction have two arguments. The second argument is the trap number designated the operation.

```
trap r1 #n       special instruction, n is in r2-field.

trap r1 #0       stop simulation
trap r1 #1       print integer in R[r1]
trap r2 #2       print character in R[r2]
```

Instruction format

L-format op:5 r1:5 ads:22
D-format op:5 r1:5 r2:5 disp:17
X-format op:5 r1:5 r2:5 r3:5 xop:12

(ads 22-bit, disp 17-bit sign extended)

Instructions are fixed length at 32 bits. There are 32 registers. The address space is 32-bit. Access to memory is always on word boundary (no byte-access). Absolute address (L-format) is 22-bit or the first 4M words. Index and indirect access can reach the whole 32-bit address space. Immediate value (D-format) is 17-bit. It is sign extended. The jump instructions (jmp, jt, jf) have 22-bit address.

ISA and opcode encoding

opcode format
```
0   L    nop            no operation
1   L    ld r1 ads      (ads 22-bit)
2   D    ld r1 @d r2    (d 17-bit sign extended)
3   L    st r1 ads      (ads 22-bit)
```

```
4  D    st r1 @d r2   (d 17-bit sign extended)
5  L    mov r #m      (m 22-bit sign extended)
6  L    jmp ads
7  L    jal r1 ads
8  L    jt r1 ads
9  L    jf r1 ads
10 D    add r1 r2 #n  (n 17-bit sign extended)
11 D    sub r1 r2 #n  ...
12 D    mul r1 r2 #n
13 D    div r1 r2 #n
14 D    and r1 r2 #n
15 D    or r1 r2 #n
16 D    xor r1 r2 #n
17 D    eq r1 r2 #n
18 D    ne r1 r2 #n
19 D    lt r1 r2 #n
20 D    le r1 r2 #n
21 D    gt r1 r2 #n
22 D    ge r1 r2 #n
23 D    shl r1 r2 #n
24 D    shr r1 r2 #n
25 D    mod r1 r2 #n
26..30 undefined
31 xop    -     X

xop
0  X   add r1 r2 r3
1  X   sub r1 r2 r3
2  X   mul r1 r2 r3
3  X   div r1 r2 r3
4  X   and r1 r2 r3
5  X   or r1 r2 r3
6  X   xor r1 r2 r3
7  X   eq r1 r2 r3
8  X   ne r1 r2 r3
9  X   lt r1 r2 r3
10 X   le r1 r2 r3
11 X   gt r1 r2 r3
12 X   ge r1 r2 r3
13 X   shl r1 r2 r3
14 X   shr r1 r2 r3
15 X   mod r1 r2 r3
16 x   mov r1 r2
17 X   ld r1 +r2 r3
18 X   st r1 +r2 r3
19 X   ret r1
20 X   trap r1 #n    use n at r1 n = 0..31
21 X   push r1 r2    use r1 as stack pointer
22 X   pop r1 r2     use r1 as stack pointer
23 X   not r1 r2
24 X   int #n        sw interrupt 0..3
25 X   reti
26 X   pushm r1      use r1 as stack pointer
27 X   popm r1       use r1 as stack pointer
28 X   intx #n       use n at r1, n = 0..31
29 X   wfi
30 X   cid r1
```

```
31 X   sync

32..4095 undefined
```

when a field is not used, it is filled with 0.

last update 8 Mar 2017