S3.0 : A multicore 32-bit Processor

S3.0 is a multicore version of S2.1 simple processor. This is a typical simple 32-bit processor. It has three-address instructions and 32 registers. Most operations are register to register. The `ld/st` (load/store) instructions are used to move data between registers and memory. This document presents only S30 assembly language view. It does not give details about microarchitecture (such as pipeline).

Format

A general format of an instruction (register to register operations) using the convention "op dest source" is as follow:

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

such as

```
add r1 r2 r3   means  R[r1] = R[r2] + R[r3]
```

Addressing

To move values between memory and registers, `ld/st` instructions are used. There are three addressing mode: absolute, indirect and index. (`ld` is mem to reg, `st` is reg to mem).

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

similarly for store instruction

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

Instruction type

```
arithmetic and logic:
     add sub mul div
     and or not xor shl shr
     eq ne lt le gt ge
control flow:
     jmp jt jf jal ret
data:
     ld st mv push pop
```

Instruction meaning

```
      false == 0
      true  != 0
```

Data

```
ld r1 ads        is    R[r1] = M[ads]             load absolute
ld r1 @d r2      is    R[r1] = M[d+R[r2]]         load indirect
ld r1 +r2 r3     is    R[r1] = M[R[r2]+R[r3]]     load index
st r1 ads        is    M[ads] = R[r1]             store absolute
st r1 @d r2      is    M[d+R[r2]] = R[r1]         store indirect
st r1 +r2 r3     is    M[R[r2]+R[r3]] = R[r1]     store 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
ret r1           is    PC = R[r1]                 return
mv r1 r2         is    R[r1] = R[r2]
mv r1 #n         is    R[r1] = #n                 move immediate
```

Arithmetic

two-complement integer arithmetic

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


add r1 r2 r3      R[r1] = R[r2] + R[r3]                add
add r1 r2 #n      R[r1] = R[r2] + sign extended n      add immediate
...
```

logic (bitwise)

```
and r1 r2 r3      R[r1] = R[r2] bitand R[r3]                and
and r1 r2 #n      R[r1] = R[r2] bitand sign extended n    and immediate
...
eq r1 r2 r3       R[r1] = R[r2] == R[r3]          equal
eq r1 r2 #n       R[r1] = R[r2] == #n             equal immediate
...
shl r1 r2 r3      R[r1] = R[r2] << R[r3]          shift left
shl r1 r2 #n      R[r1] = R[r2] << #n             shift left immediate
. . .
not r1 r2         R[r1] = ~R[r2]                  logical not

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 is used as a stack pointer.

```
push r1 r2           R[r1]++; M[R[r1]]=R[r2]
pop  r1 r2           R[r2] = M[R[r1]]; R[r1]—
pushm r1            push R[0]..R[15] to stack
popm r1             pop R[0]..R[15] from stack
```

Interrupt

There is one level hardware interrupt and one software interrupt. R[31] stores the PC when an interrupt occurs. It is used for returning from an Interrupt Service Routine. The interrupt vector is designated at the location 1000+core_id (where core_id is 0…NC-1, where NC is the number of core).

```
ei                  enable interrupt
di                  disable interrupt
int 0               R[31] = PC, PC = M[1000+cid], software interrupt
reti                PC = R[31]
wfi                 wait for interrupt
intx r1             send interrupt signal to core R[r1]
```

Multicore support

```
cid r1              return core_id in R[r1]
sync                sync all cores
```

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

(r1 dest, r2,r3 source, ads and disp are sign extended)
```

Instructions are fixed length at 32 bits. There are 32 registers. The address space is 32-bit (4G) with 22-bit direct addressable (4M). The addressing unit is word (32-bit).

Opcode encoding

opcode op   format

```
0   nop                 L
1   ld    r1 ads        L     (ads 22 bits)
2   ld    r1 @d r2      D     (d 17 bits)
3   st    r1 ads        L
4   st    r1 @d r2      D
5   mv    r1 #n         L     (n 22 bits)
6   jmp   ads           L     (ads 22 bits)
```

```
7  jal  r1 ads        L    (ads 22 bits)
8  jt   r1 ads        L
9  jf   r1 ads        L
10 add  r1 r2 #n      D    (n 17 bits)
11 sub  r1 r2 #n      D
12 mul  r1 r2 #n      D
13 div  r1 r2 #n      D
14 and  r1 r2 #n      D
15 or   r1 r2 #n      D
16 xor  r1 r2 #n      D
17 eq   r1 r2 #n      D
18 ne   r1 r2 #n      D
19 lt   r1 r2 #n      D
20 le   r1 r2 #n      D
21 gt   r1 r2 #n      D
22 ge   r1 r2 #n      D
23 shl  r1 r2 #n      D
24 shr  r1 r2 #n      D
25..30 undefined
31 extended op        X

xop
0 add    r1 r2 r3     X
1 sub    r1 r2 r3     X
2 mul    r1 r2 r3     X
3 div    r1 r2 r3     X
4 and    r1 r2 r3     X
5 or     r1 r2 r3     X
6 xor    r1 r2 r3     X
7 eq     r1 r2 r3     X
8 ne     r1 r2 r3     X
9 lt     r1 r2 r3     X
10 le    r1 r2 r3     X
11 gt    r1 r2 r3     X
12 ge    r1 r2 r3     X
13 shl   r1 r2 r3     X
14 shr   r1 r2 r3     X
15 mv    r1 r2        X
16 ld    r1 +r2 r3    X
17 st    r1 +r2 r3    X
18 ret   r1           X
19 trap  n            X      use r1 as trap code (n 0..31)
20 push  r1 r2        X      use r1 as stack pointer
21 pop   r1 r2        X      use r1 as stack pointer
22 not   r1 r2        X
23 int   0            X
24 reti               X      no argument
25 ei                 X      use r1 as sp
26 di                 X      use r1 as sp
27 pushm r1           X      push multiple, r1 as sp
28 popm  r1           X      pop multiple, r1 as sp
29 cid r1             X
```

```
30 wfi                   X
31 intx r1               X
32 sync                  X
33..4095 undefined
```

Historical fact

S30 is based on S21 instruction set. S21 is an extension of S2 (S2, 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 processor) which was created in 1997.

To improve understandability of S2 assembly language, flags are not used. Instead, new logical instructions that have 3-address are introduced. 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. To avoid the confusion between absolute addressing and moving between registers, a new instruction "mv" is introduced. (and "ld r1 #n" is eliminated.)

The opcode format and assembly language format for S2 follow the tradition “dest = source1 op source2” from well-known historical computers: PDP, VAX and IBM S360.

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

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

The difference between S3.0 and S2.1 is as follows:

R[0] is free in S30, versus it is always zero in S2.1. Interrupt keeps return address in R[31], versus S2.1 keeps it in special `RetAds` register. All these changes made the instruction set simpler. S30 has additional instructions to control multicore: `cid, wfi, initx, sync`.

last update 1 May 2016