1. Introduction

This document describes the Simple 42. This is the machine currently being implemented.
# CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2. Processor</td>
<td>2</td>
</tr>
<tr>
<td>2.1. Registers</td>
<td>2</td>
</tr>
<tr>
<td>2.2. Workspace</td>
<td>2</td>
</tr>
<tr>
<td>2.3. Scheduling</td>
<td>3</td>
</tr>
<tr>
<td>2.4. Channels</td>
<td>3</td>
</tr>
<tr>
<td>2.5. Synchronisation</td>
<td>3</td>
</tr>
<tr>
<td>2.6. Input and Output</td>
<td>4</td>
</tr>
<tr>
<td>2.7. Addressing</td>
<td>4</td>
</tr>
<tr>
<td>2.8. Initialisation</td>
<td>5</td>
</tr>
<tr>
<td>2.9. Logical operations</td>
<td>5</td>
</tr>
<tr>
<td>2.10. Arithmetic operations</td>
<td>5</td>
</tr>
<tr>
<td>2.10.1. Overflow</td>
<td>5</td>
</tr>
<tr>
<td>2.10.2. Addition and Subtraction</td>
<td>5</td>
</tr>
<tr>
<td>2.10.3. Multiplication</td>
<td>6</td>
</tr>
<tr>
<td>2.10.4. Division</td>
<td>6</td>
</tr>
<tr>
<td>2.11. Instruction formats</td>
<td>6</td>
</tr>
<tr>
<td>2.12. Primary Instructions</td>
<td>7</td>
</tr>
<tr>
<td>2.13. Secondary Instructions - Group one</td>
<td>15</td>
</tr>
<tr>
<td>2.15. Instruction summary</td>
<td>26</td>
</tr>
<tr>
<td>2.15.1. Primary instructions</td>
<td>26</td>
</tr>
<tr>
<td>2.15.2. Secondary Instructions</td>
<td>27</td>
</tr>
</tbody>
</table>
2. Processor

2.1. Registers

There are seven registers, each of length 1 word. They are:
- The pointer to the next instruction to be executed
- The pointer to the current process workspace
- The pointer to the end of the list of active processes
- The primary accumulator
- The secondary accumulator
- The carry accumulator
- The operand accumulator

The instruction pointer always points to the next instruction to be executed.

The workspace pointer is used to access all data used by the process.

The pw pointer is used only to manipulate the active process list.

The a and b registers are the sources for most arithmetic and logical operations. The c register is used to extend the length of a for operations manipulating double word values. In particular, it is used to hold the carry word for multiple length arithmetic.

The a and b registers are organised as a two word stack; loading a value pushes a into b and loads a, storing a value stores a and pops b into a. The c register is loaded and stored explicitly.

The o register is used in the formation of instruction operands.

2.2. Workspace

A process workspace consists of a vector of words in memory. It is used to hold the local variables and temporary values manipulated by the process. The first word (link) of each workspace is used to hold the link to the next process on the active process list; the second word (iptr) is used to hold the current instruction pointer of the process.
2.3. Scheduling

A queue of active processes is maintained; this is a linked list in which the link word of each workspace points to the next workspace. The pw register always points to the workspace behind the current one; this facilitates adding new processes to the list.

The processor executes the processes on the list in sequence, advancing to the next process whenever a pause instruction is executed, and whenever a process deschedules itself by executing a wait or synchronise instruction.

Processes may be added to the end of the list by a run instruction. A process which is descheduled when it performs a synchronise instruction on a channel is added to the end of the list when another process performs a synchronise instruction on the same channel.

(In a multiprocessor system, the implementation of run must cause a process to be scheduled on the appropriate processor; this is assumed to be the processor in whose local memory the process workspace resides.)

2.4. Channels

A channel is used to allow two processes to synchronise and communicate. It consists of one (or more) consecutive words in memory. The first of these is either set to 0, indicating that neither process is waiting to synchronise, or it points to the workspace of the waiting process. The second (and subsequent) words are used to hold data being communicated.

2.5. Synchronisation

When a process executes a synchronise instruction on a channel, the first word of the channel is tested to determine whether another process has executed a synchronise instruction on the channel (and is therefore halted). If so, the waiting process is released (by adding it to the end of the active process list); otherwise a pointer to the current process workspace is written to the channel and the current process halted (by removing it from the active process list).

A process may test whether a channel is ready to synchronise by testing whether the first word of the channel is 0.

When two processes share the same address space, the synchronise instruction must be implemented as an 'indivisible operation'.
2.6. Input and Output

Output is performed by storing the data to be output in the second and subsequent words of the channel, and executing two synchronise instructions on the channel; the first of these serves to indicate that the channel contains data; the second that the data has been taken (effectively a null input).

Input is performed by executing a synchronise instruction on the channel, taking the data from the channel and executing a further synchronise instruction on the channel (effectively a null output).

Output followed by input may be optimised by returning data instead of null on the second synchronisation.

Physical input and output is provided by channels which are connected directly to physical devices; the data words of the channel being connected directly to pins, or shift registers. The 'handshake' pins (if there are any) may be used to synchronise using the channel in the same way as the synchronise instruction.

It is expected that all other input and output mechanisms (such as block transfer devices) are regarded as degenerate processors; and communicate via channels (to receive pointers to blocks to be output etc.) in the manner described above.

2.7. Addressing

The memory may be addressed as either bytes or words. Incrementing a word address gives the address of the next word, and incrementing a byte address gives the address of the next byte. Consequently, the word address of a location is not the same as the byte address of the location.

If \( x \) is the word address of a location, the byte address of the location is \( \text{bytesperword} \times x \) (for a 16 bit machine, \( \text{bytesperword} = 2 \)).

The arrangement of bytes within a word is such that the byte at byte address \( (\text{bytesperword} \times x) \) occupies the least significant 8 bits of the word at word address \( x \).

The \( \text{w} \) and \( \text{pw} \) registers always hold word addresses; the \( \text{i} \) register always holds a byte address; instructions can therefore be executed only from the bottom (256 ** \( \text{bytesperword} \)) byte addresses in memory.
2.8. Initialisation

Initially the $w$, $pw$ and $i$ registers are undefined. The processor executes instructions from an input channel; these instructions serve to bootstrap (and test) the processor. An instruction (switch) is used to load the $w$, $pw$ and $i$ registers and to switch the processor between the run state (in which instructions are executed using the instruction pointer) and the initial state.

2.9. Logical operations

Logical operators perform bitwise operations on single word operands. The representation of false is assumed to be 0; the representation of true is assumed to be -1 (a word consisting entirely of '1' bits).

2.10. Arithmetic operations

The arithmetic operations are designed to implement single length 2's complement arithmetic. They also provide for unsigned and multiple precision arithmetic.

2.10.1. Overflow

It is possible to generate numbers which are too large to fit into a single word. This condition is known as overflow. An analysis of how this can occur in the arithmetic instructions follows.

2.10.2. Addition and Subtraction

These operations work both on signed and unsigned numbers. The only difference in usage is the overflow/carry condition. In the case of unsigned arithmetic overflow occurs when the result is too large to fit into one word. This is indicated by the carry being set. In the case of signed arithmetic there is never overflow when adding numbers of different sign or subtracting numbers of the same sign. However, overflow can occur when adding numbers of the same sign or subtracting numbers of differing sign. Overflow is detected by checking the sign of the result against the sign that would be expected on performing the operation. For example, when adding positive numbers the result is expected to be positive. If a negative result is obtained overflow has occurred. For example, in 16 bit 2's complement

$$\begin{align*}
(2^{15} - 1) + (2^{15} - 1) &= (2^{16} - 2)
\end{align*}$$

which is -2 when interpreted as a 2's complement number.
2.10.3. Multiplication

In both signed and unsigned multiplication two single length numbers are used to produce a double length result. The result can always be expressed correctly as a double length number, so no overflow can occur. However, it is also possible that the result can be correctly expressed as a single length number. For this to happen using signed arithmetic, the more significant word of the product must be a sign extension of the least significant word (ie all bits of the top word must equal the sign bit of the bottom word). This is the overflow condition set by the signed multiplication instruction. For unsigned arithmetic the result can be correctly expressed as a single length number if the more significant word of the result is zero.

2.10.4. Division

In division, a double length number is divided by a single length divisor giving a single length quotient and single length remainder. Overflow can occur in two ways; in dividing by zero, and in generating a quotient which is too large to fit into a single word.

2.11. Instruction formats

Each instruction is one byte long, and is divided into two 4 bit parts. The four least significant bits of the byte hold the instruction code, and the four most significant bits hold the operand. Instructions are executed by loading the 4 operand bits into the least significant four bits of the operand accumulator, which is then used as the actual operand of the instruction. An instruction (pfix) is provided to shift up the contents of the operand register by 4 places, thus allowing actual operands of any length up to one word to be represented. A further instruction (mpfix) is provided to allow negative operands up to one word long to be efficiently represented.
2.12. Primary Instructions

Load from workspace

ldw

code: 0

def: \( o := o + \text{opd} \)
\( b := a \)
\( a := w ! o \)
\( o := 0 \)

purpose: to load the value of a location in the current process workspace

Store to workspace

stw

code: 1

def: \( o := o + \text{opd} \)
\( w ! o := a \)
\( a := b \)
\( o := 0 \)

purpose: to store a value in a location in the current process workspace
load pointer into workspace

ldpw

code: 2

def: o := o + opd
    b := a
    a := w + o
    o := 0

purpose: to load a pointer to a location in the current process workspace

to load a pointer to the first location of a vector of locations in the current process workspace

load from workspace and increment

ldwi

code: 3

def: o := o + opd
    b := a
    a := w ! o
    w ! o := a + 1
    o := 0

purpose: to load the value of a location in the current process workspace, and increment the location

to facilitate the use of workspace locations as loop counters, incrementing towards zero

to facilitate the use of workspace locations as incrementing pointers to vectors of words or bytes
load from table

ldt

code: 4

def: o := o + opd
      a := a ! o
      o := 0

purpose: to load a word from an outer workspace
         to load a word from a table of values
         to load a word, using a word as a pointer
             (indirection) - in this case opd = 0

store to table

stt

code: 5

def: o := o + opd
      a ! o := b
      o := 0

purpose: to store a value in a location in an
         outer workspace
         to store a value in a table of values
         to store a word, using a word as a pointer
             (indirection) - in this case opd = 0
load pointer into table

ldpt

code: 6

def: o := o + opd
      a := a + o
      o := 0

purpose: to load a pointer to a location in an outer workspace

      to load a pointer to a location in a table of values

      to add a value to the accumulator

load from table and increment

ldti

code: 7

def: o := o + opd
      a, a!o := a!o, a!o + 1
      o := 0

purpose: to load the value of a location in an outer workspace, and increment the location

      to load the value of a location in a table of values, and increment the location
jump

j

code: 8

def: o := o + opd
i := i + o
o := 0

purpose: to transfer control forwards or backwards, providing loops, exits from loops, continuation after conditional sections of program

jump non zero

jnz

code: 9

def: o := o + opd
if a ^= 0 then i := i + o
a := b
o := 0

purpose: to transfer control forwards or backwards only if a non-zero value is loaded, providing conditional execution of sections of program and conditional loop exits

to facilitate comparison of a value against a set of values
load literal

ldl

code: 10

def: o := o + opd
      b := a
      a := o
      o := 0

purpose: to load a value

load pointer to instruction

ldpi

code: 11

def: o := o + opd
      b := a
      a := i + o
      o := 0

purpose: to load a pointer to a section of program
operate one

opr1
code: 12
def: (execute opd as a group one secondary instruction)
o := 0

purpose: perform a secondary instruction, using the operand as a group one secondary instruction code.

operate two

opr2
code: 13
def: (execute o as a group two secondary instruction)
o := 0

purpose: perform a secondary instruction, using the operand as a group two secondary instruction code.
prefix

prefix

code: 14

def: o := o + opd
      o := o << 4

purpose: to allow instruction operands which are not in the range 0 - 15 to be represented using one or more prefix instructions.

negative prefix

nprefix

code: 15

def: o := (-opd) << 4

purpose: to allow negative operands to be efficiently represented
2.13. Secondary Instructions - Group one

reverse

rev

code: 0

def: a, b := b, a

purpose: to reverse operands of asymmetric operators, where this cannot conveniently be done in a compiler

equal to zero
eqz

code: 1

def: a := a = 0

purpose: to test that a holds a non-zero value
to implement logical (but not bitwise) negation
to implement
a = 0 as eqz
a ~= 0 as eqz, eqz
if a = 0 ... as jnz
if a ~= 0 ... as eqz, jnz

greater
gt

code: 2

def: a := b > a (signed)
c := b > a (unsigned)

purpose: to compare a and b (treating them as two's complement integers), loading -1 (true) if b is greater than a, 0 (false) otherwise
to implement b > a (unsigned) as gt, ldac
to facilitate multiple precision comparisons
to implement b < a by reversing operands
to implement b <= a as (gt, eqz), and b >= a by reversing operands, and (gt, eqz)
and

and

code:  3

def:   a := b \ a

purpose: to load the logical and of a and b, setting each bit to 1 if the corresponding bits in both a and b are set to 1, 0 otherwise

or

or

code:  4

def:   a := b \ a

purpose: to load the logical or of a and b, setting each bit to 1 if either of the corresponding bits of a and b is set, 0 otherwise

xor

xor

code:  5

def:   a := b xor a

purpose: to load the logical exclusive or of a and b, setting each bit to 1 if the corresponding bits of a and b are different, 0 otherwise

to implement bitwise not as (ldl -1, xor)
clear carry

clc

code: 6

def: c := 0

purpose: to clear c before multiple precision arithmetic

load from carry

ldac

code: 7

def: b := a
     a := c

purpose: to load c after multiple precision arithmetic
to facilitate unsigned comparisons

store to carry

stac

code: 8

def: c := a
     a := b

purpose: to set c before multiple precision arithmetic
add

code:  9

def:  a := b + a

purpose:  to load the sum of b and a
to compute addresses of words or bytes in vectors

add with carry

addc

code:  10

def:  a := b + a + c<0>
b := overflow(b + a + c<0>)
c := carry(b + a + c<0>)

purpose:  to load the sum of a, b and the least significant bit of c, setting b to indicate arithmetic overflow, c to indicate carry

to facilitate multiple precision addition

subtract

sub

code:  11

def:  a := b - a

purpose:  to subtract a from b, loading the result.
to implement
    a = b  as sub, eqz
    a := b  as sub, eqz, eqz
    if a = b .. as sub, jnz, ..
    if a := b .. as sub, eqz, jnz, ..

subtract with borrow

subc

code:  12

def:  a := b - a - c<0>
b := overflow(b - a - c<0>)
c := carry(b - a - c<0>)

purpose:  to subtract a and the least significant bit of c from b, loading the result, setting b to indicate arithmetic overflow, c to indicate carry (borrow)
to facilitate multiple precision subtraction
signed multiply

mul

code: 13

def: c, a := b * a (signed)
b := overflow(b * a)
purpose: to multiply a and b, loading the least significant part of the result into a, the more significant into c. b is set if the double length result cannot be reduced to single length without loss of accuracy.

unsigned multiply and add

umul

code: 14

def: c, a := a*b + c (unsigned)
b := b
purpose: to multiply a and b, adding c into the result, loading the least significant part of the result into a and setting c to the most significant part.

to facilitate multiple precision arithmetic

signed divide

div

code: 15

def: a := c.a / b (signed - a takes sign of c.a/b)
b := overflow(c.a / b)
c := c.a REM b (signed - c takes sign of c.a)
purpose: to divide c and a by b, loading the result and setting c to the remainder.

b is set to overflow (can be caused by divide by 0 or by having c.a/b too large for a single word)

to facilitate single length division
2.14. Secondary Instructions - Group two

unsigned divide

udiv

code: 0

def: 
a := c.a / b  (unsigned)
b := overflow(c.a/b)
c := c.a REM b  (unsigned)
purpose: to divide c and a by b, loading the result and setting c to the remainder.
to facilitate multiple length division.

sign extend

sex

code: 1

def: if a < 0 then c := -1 else c := 0

purpose: to convert a single length signed integer into a double length signed integer

test long

tlng

code: 2

def: 
b := ~((c=0 \ a>0) \ (c=-1 \ a<0))
purpose: to test if a double length signed value can be reduced to a single length signed value
load byte

lb

code: 3

def: a := b % a

purpose: to load a byte from a string or vector of bytes

store byte

sb

code: 4

def: b % a := c

purpose: to store a byte in a string or vector of bytes
shift left

sl

code: 5

def: c.a := c.a << b

purpose: to shift c and a left by b places, filling unused bits with 0, and shifting bits from the most significant end of a into the least significant end of c

shift right

sr

code: 6

def: c.a := c.a >> b

purpose: to shift c and a right by b places, filling unused bits with 0, and shifting bits from the least significant end of c into the most significant end of a
to provide field extraction in conjunction with the and instruction

jump indexed

jx

code: 7

def: i := i + a

purpose: to transfer control to one of a number of sections of program, depending on the value in the accumulator
wait

wait

code: 8

def:  
w ! iptr := i
w := w ! link
pw ! link := w
i := w ! iptr

purpose: to remove the current process from the active process list and advance to the next process on the active process list

run

run

code: 9

def:  
pw ! link := a
pw := a
a ! link := w

purpose: to add a process to the end of the active process list

pause

pse

code: 10

def:  
w ! iptr := i
pw := w
w := w ! link
i := w ! iptr

purpose: to share the processor time between the processes currently on the active process list
call process

call

code: 11

def: w ! iptr := i
       pw ! link := a
       a ! link := w ! link
       w, a := a, w
       i := b

purpose: to replace the current process on the active
         process list with a (newly created) process

         to facilitate code sharing, where two identical
         processes are executed on the same processor

         (fails if only one active process)

return from process

ret

code: 12

def: pw ! link := a
         a ! link := w ! link
       w := a
       i := w ! iptr

purpose: to replace the current process with its caller

synchronise

sync

code: 13

def: if a!0 = 0
         then a!0 := w
         wait
         else run(a!0)
             a!0 := 0

purpose: to allow two processes to synchronise and
          communicate using a channel
switch

switch
code: 14
def:
running := (a ^= 0)
w := a
pw, a := a, pw
i := b

purpose: to switch the processor between the run state and the initial state

move block

move
code: 15
def:
do c times
!a := !b
unless iochan(a) do a := a + 1
unless iochan(b) do b := b + 1

purpose: to provide rapid transfer of blocks of data from
memory to memory
memory to channel
channel to memory
channels to channel
2.15. Instruction summary

2.15.1. Primary instructions

workspace operations

- load from workspace
- store to workspace
- load pointer into workspace
- load from workspace and increment

table operations

- load from table
- store to table
- load pointer into table
- load from table and increment

jumps

- jump
- jump non zero

literal and address loads

- load literal
- load pointer to instruction

operations on registers

- operate one
- operate two

long operands

- prefix
- negative prefix
2.15.2. Secondary Instructions

operand preparation
   reverse

relational
   equal to zero
   greater

logical
   and
   or
   exclusive or

carry accumulator operations
   clear carry
   load from carry
   store to carry

arithmetic
   add
   add with carry
   subtract
   subtract with borrow

signed multiply
unsigned multiply and add

signed divide
unsigned divide

sign extend
test long
byte operations
  load byte
  store byte
shifts
  shift left
  shift right
table jump
  jump indexed
scheduling
  wait
  run
  pause
code sharing and calling
  call process
  return from process
synchronisation
  synchronise
initialisation
  switch
block moves
  move block