Object Oriented Programming With A Limited Instruction Set

SSBC Version 5 Revision 0 (Abstract RTN)

Processor State:
```
PC<15..0>:        Program Counter
SP<15..0>:        Stack Pointer
MR<15..0>:        Memory Register
Ri<7..0>:       Register i (0 <= i <= 3)
IR<7..0>:       Instruction Register
Z:              1-bit Zero Flag
N:              1-bit Negative Flag
fault:          1-bit Fault Indicator
halt:             1-bit Halt Indicator
Reset:          Rest Signal
```

Main Memory:
```
MEM[0..2^16-1]<7..0>
```

Instruction Format:
```
opcode<3..0> := IR<3..0>
```

Operands/Address:
```
ii<7..0> := MEM[PC]
s1<7..0> := MEM[SP+1]
s2<7..0> := MEM[SP+2]
ext<15..0> := MEM[PC]#MEM[PC+1]
```

Program Status Word:
```
PSW<7..0> := Z#N#6@0:
```

Instruction Interpretation:
```
Ins_interpretation := (Reset -> (PC <- 0x0: SP <- 0xFFFA: halt <- 0x0:
   Fault <- 0x0: ins_interpretation): (NOT Reset) and (NOT Fault) ->
   (IR <- MEM[PC]; set_fault; (NOT Fault) -> (PC <- PC+1; ins_exe));
```

Fault Detection:
```
Set_fault := fault <- NOT (0 <= opcode <= 0xA)
```

Instruction Execution:
```
Ins_exe := (
   Noop (:= opcode=0) -> PC <- PC+1:
   Halt (:= opcode=1) -> halt <- 0x1:
   Pushimm (:= opcode=2) -> MEM[SP] <- ii; (SP <- SP-1;:PC <- PC+1):
   Pushext (:= opcode=3) -> MEM[SP] <- MEM[ext];
      (SP <- SP-1: PC <- PC+2):
   Popinh (:= opcode=4) -> SP <- SP+1:
   Popext (:= opcode=5) -> MEM[ext] <- s1; (SP <- SP+1: PC <- PC+2):
   Jnz (:= opcode=6) -> (NOT Z) -> PC <- ext:
   Jnn (:= opcode=7) -> (NOT N) -> PC <- ext:
   Add (:= opcode=8) -> MEM[SP+2] <- s1+s2;
      (Z <- NOT (R2<7> OR R2<6> .. OR R2<0>): N <- R2<7>: SP <- SP+1:
   Sub (:= opcode=9) -> MEM[SP+2] <- s1-s2;
      (Z <- NOT (R2<7> OR R2<6> .. OR R2<0>): N <- R2<7>: SP <- SP+1:
   Nor (:= opcode=10) -> MEM[SP+2] <- s1 NOR s2; SP <-SP+1:
); ins_interpretaion):
```

Memory Map:
```
PSW mapped to 0xFFFB
Port A (read only) mapped to 0xFFFC
Port B (read only) mapped to 0xFFFD
Port C (read only) mapped to 0xFFFE
Port D (read only) mapped to 0xFFFF
```

Notation:
```
->          if-then: true condition on left yields value and/or
              Action on right
<-          Register transfer: register on LHS stores value from RHS
:           Parallel Seperator: actions or evaluations carried out
              simultaneously
;           Sequential Seperator: RHS evaluated and/or performed
              after LHS
:=          Definition: text substitution
```

Description

What is SSBC?

The SSBC is a simple stack based computer. It is an 8 bit computer with a 16 bit address space and an 11 opcode instruction set.

What is the SSBC Compiler?

The SSBC Compiler is a compiler that takes a high level language with object oriented features such as C++ and compiles it into SSBC machine code.

Why is this significant?

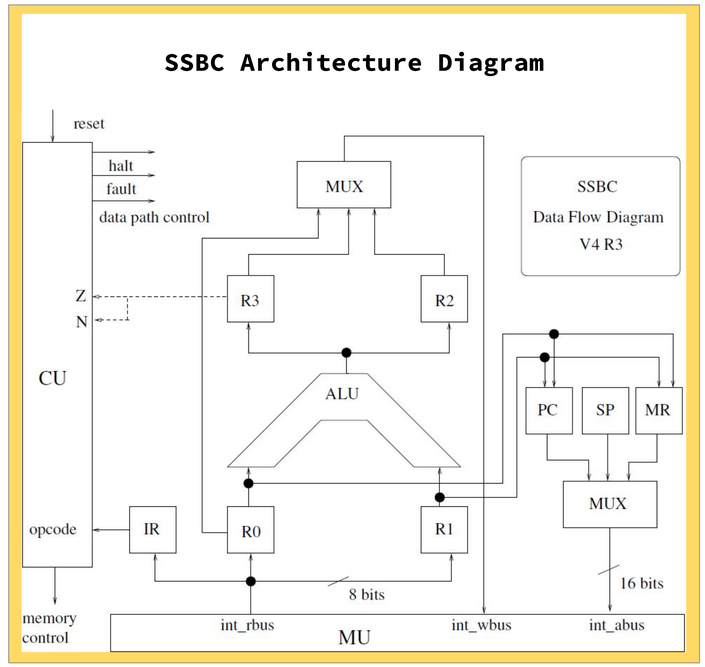
By designing a compiler for the SSBC we are able to show that object oriented programming can be achieved on a stack based computer with a limited instruction set.

An example using a function pointer can be seen on the right. Top Right (C++) Bottom Right (SSBC)

Function Pointer Example (C++)

```cpp
#include <iostream>

// Takes an integer and squares it
int square(int x) {
    return x*x;
}

int main() {
    // declare function pointer that
    // takes an integer as a parameter
    int (*calcSqrArea)(int);

    // point function pointer at
    // square function
    calcSqrArea = &square;

    // calcSqrArea(2) has same functionality
    // as square(2)
    int area = calcSqrArea(2);

    std::cout << area << std::endl;

    return 0;
}
```

SSBC Architecture Diagram



Function Pointer Example (SSBC)

```
//point function pointer at
square function
00000010 pushimm
00000000     H:square
00000101 popext
00000000     calcSqrArea_HIGH
00101110
00000010 pushimm
00101111     L:square
00000101 popext
00101101     calcSqrArea_LOW
00101111

//compute calcSquareArea(2)
00000010 pushimm
00100010     L:return
00000010 pushimm
00000000     H:return
00000010 pushimm
00000010     2
00000010 pushimm
00000000     0
00000101 popext
11111111     PSW
11111011
00000011 pushext
00000000     calcSqrArea_HIGH
00101110
00000101 popext
00000000     jump_HIGH
00100010
00000011 pushext
00101101     calcSqrArea_LOW
00101111
00000101 popext
00000000     jump_LOW
00100011
00000110 jnz
00000000     jump_HIGH: 0
00000000     jump_LOW: 0

//return:
//save result to variable area
00000101 popext
00000000     area
00110000

//print value in area to port A
00000011 pushext
00000000     area
00110000
00000101 popext
11111111     portA
11111100

//halt program
00000001 halt

//reserve memory space main's variables
00000000 calcSqrArea_HIGH: 0
00000000 calcSqrArea_LOW: 0
00000000 area: 0

//square:
00000101 popext
00000000     x
01101110
00000011 pushext
00000000     x
01101110
00000101 popext
00000000     i
01101111
00000010 pushimm
00000000     0
00000101 popext
00000000     result
01110000

//return x*x
00000011 pushext
00000000     i
01101111
00000010 pushimm
00000000     0
00001001 sub
00000111 jnn
00000000     end_while
01100011
00000011 pushext
00000000     x
01101110
00000011 pushext
00000000     result
01110000
00001000 add
00000101 popext
00000000     result
01110000
00000010 pushimm
11111111     -1
00000011 pushext
00000000     i
01101111
00001001 sub
00000101 popext
00000000     i
01101111
00000010 pushimm
00000000     0
00000101 popext
11111111     PSW
11111011
00000110 jnz
00000000     while_loop
00111111
00000101 popext
00000000     square_return_HIGH
01101100
00000101 popext
00000000     square_return_LOW
01101110
00000011 pushext
00000000     result
01110000
00000110 jnz
00000000     square_return_HIGH
00000000     square_return_LOW

//reserve memory for square's
variables
00000000 x: 0
00000000 i: 0
00000000 result: 0
```

By Eric Roberts and Justin Pye