Stack machine

By Suwat Taechaphetchapaiboon (43410007)
Supachai Budsaratij (43410011)

The stack machine organized to simplify the implementation of block structured languages. It provides dynamic storage allocation through a stack of activation records. The activation records are linked to provide support for static scooping and they contain the context information to support procedures.

Motivation of design

The Stack Machine consists of a code segment, a data segment, a stack segment, an Arithmetic Logic unit (ALU), three address registers and a flag register. The code segment has the program to be executed. The stack segment holds the intermediate data and addresses generated during program execution. The three address registers are pointers into the code and stack segments:

- Instruction register (IR) kept opcode.
- Stack pointer (SP) points to the valid items on top of the stack.
- Program counter (PC) points to the next instruction to be executed
- Flag shows status of operation.(C carry, O : overflow , S : sign, Z : zero)
- ALU have 2 input latch (A,B register) and 1 output latch (C register)
- 256 I/O port

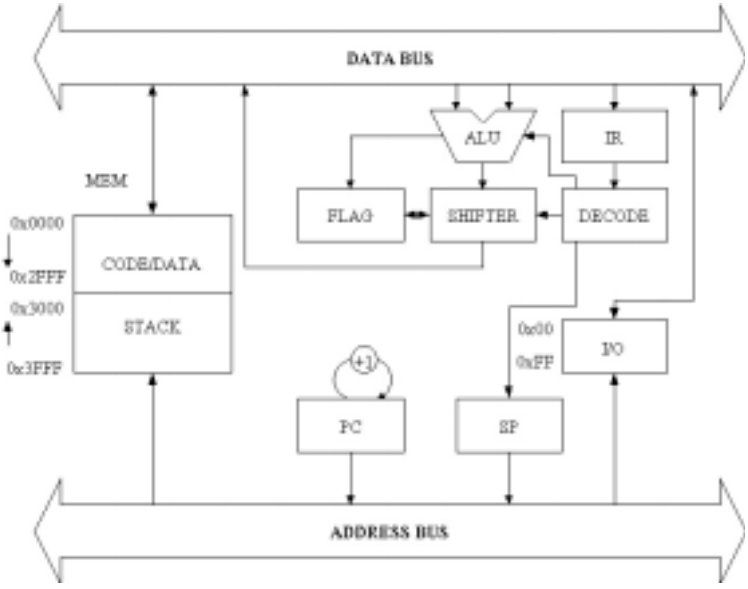
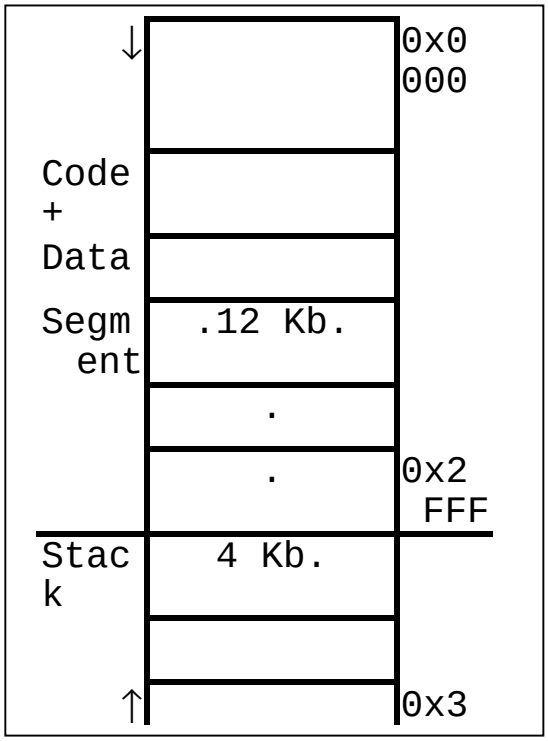


Figure 1 Architecture of Stack Machine

Code segment and data segment is 12 Kb., grow up from 0x0000 to 0x2FFF g, Stack segment is 4 Kb. Start at 0x3FFF down to 0x2FFF. The capability of memory is 16 Kb. as figure 2



Instruction set of the Stack Machine

An assembly language program for a stack machine is comprised of arithmetic, logical, input/output, control, stack altering, and assembler directive instructions. Table A summarizes the complete instruction set of the designed Stack Machine.

It is important to note that the result of some binary operation, such as subtraction and division, are sensitive to the order of operands on the stack. The description of such instructions in Table A explicitly names the operands to state the expect order. For example, in case of a division operation the divisor is at the top of the stack whereas the location below it has the dividend. Similarly, in case of a subtraction operation, the lower address has the subtrahend and the higher address has the minuend.

Instruction Encoding

The instructions of the Stack Machine are one byte, two bytes or three bytes wide. The two-byte and three-byte wide instructions are those that require an address, offset, or an immediate value. Table A shows the machine code for each instruction. Note that the opcode of an instruction always appears as the first byte. For two-byte and three-byte instruction, the second-byte and third-byte are contains the address, offset, or an immediate value.

Table A Instruction set of the Stack machine.

Instruction	Description
Arithmetic :	
ADD	Pop the top two locations, add and push the result (byte)
ADDW	Pop the top two locations, add and push the result (word)
SUB	Pop subtrahend and minuend ,subtract, and push the result (byte)
SUBW	Pop subtrahend and minuend ,subtract, and push the result (word)
MUL	Pop the multiplicand and multiplier ,multiply, and push the result (byte)
MULW	Pop the multiplicand and multiplier ,multiply, and push the result (word)
DIV	Pop the dividend and divisor, divide, and push the quotient and then the remainder (byte)
DIVW	Pop the dividend and divisor, divide, and push the quotient and then the remainder (word)
INC	Pop the top of stack , increment , and push the result to stack
INCW	Pop the top / next of stack , increment , and push the result to stack (word)
DEC	Pop the top of stack , decrement , and push the result to stack
DECW	Pop the top /next of stack , decrement , and push the result to stack (word)
Logical :	
AND	Pop the top two locations, perform bitwise AND , and push the result (byte)
ANDW	Pop the top two locations, perform bitwise AND , and push the result (word)
OR	Pop the top two locations, perform bitwise OR , and push the result (byte)
ORW	Pop the top two locations, perform bitwise OR , and push the result (word)
NOT	Pop the top location, perform bitwise NOT , and push the result (byte)
NOTW	Pop the top location, perform bitwise NOT , and push the result (word)
XOR	Pop the top two locations, perform bitwise XOR , and push the result (byte)
XORW	Pop the top two locations, perform bitwise XOR , and push the result (word)
SHL	Pop the top location, perform shift bit left 1 bit, and push the result
SHR	Pop the top location, perform shift bit right 1 bit, and push the result
ROL	Pop the top location, perform rotate left 1 bit , and push the result
ROR	Pop the top location, perform rotate right 1 bit, and push the result
Control Flow :	
CMP	Pop the top two locations, perform compare value, and set flag. (S , Z)
CMPW	Pop the top two locations, perform compare value, and set flag. (S , Z)
JMP Label	Unconditionally jump to the instruction at address label
JMR Label	Jump relate (8 bits from -128 to 127)
JS Label	Jump if sign flag is 1
JNS Label	Jump if sign flag is 0
JZ Label	Jump if zero flag is 1
JNZ Label	Jump if zero flag is 0
JC Label	Jump if carry flag is 1
JNC Label	Jump if carry flag is 0
Stack operations :	
LDI address	Pop immediately the top of stack (byte) into address.
LDIW address	Pop immediately the top of stack (Word) into address.
STI <var>	Push the value of <var> (byte)
STIW <var>	Push the value of <var> (word)

Instruction	Description
LDD address	Pop directly the top of stack (byte)
LDDW address	Pop directly the top of stack (word)
STD <var>	Push directly the value of <var> (byte)
STDW <var>	Push directly the value of <var> (word)
LDAW	
STAW	
Input / Output :	
IN port_no	Input port address 0-255.
OUT port_no	Ouput port address 0-255.
Subroutine :	
CALL address	Push the address to stack, and run subroutine.
RET	Pop the top location, and assign to PC.
Others :	
HLT	Halt program.
NOP	No operate.

Table A Instruction set of the Stack machine. (Continue)

The execution time of an instruction is evaluated on the basis of its use of ALU, registers, and the number of times, it need to access the code or stack segment. An ALU operation, code memory access, stack memory access, or data move costs one unit. The fetch and decode phases, being different from the execution phase, of an instruction are not reflected in the execution times given in Table B. The cumulative cost of fetching and decoding an instruction is considered to be one clock unit by the simulator.

Table B Instruction format, Encoding, and Execution time

Instruction	Code			Clock Units	Instruction	Code			Clock Units
	1'st byte	2'nd byte	3'rd byte			1'st byte	2'nd byte	3'rd byte	
LDI	00H	Data8		3	JO	34H	Lo adr	Hi adr	5
LDIW	01H	Lo Dat	Hi Dat	4	JNO	35H	Lo adr	Hi adr	5
LDD	02H	Lo adr	Hi adr	4	JC	36H	Lo adr	Hi adr	5
LDDW	03H	Lo adr	Hi adr	5	JNC	37H	Lo adr	Hi adr	5
LDA	04H	LO adr	HI adr	5	JS	38H	Lo adr	Hi adr	5
STA	05H	LO adr	HI adr	5	JNS	39H	Lo adr	Hi adr	5
STD	06H	Lo adr	Hi adr	4	NOP	40H			3
STDW	07H	Lo adr	Hi adr	5	HLT	41H			0
IN	08H	Adr8		3	SHL	50H			6
OUT	09H	Adr8		3	SHLW	51H			4
MUL	10H			5	SHR	52H			4
MULW	11H			8	SHRW	53H			6
DIV	12H			5	ROL	54H			4
DIVW	13H			8	ROLW	55H			6
SUB	14H			6	ROR	56H			4
SUBW	15H			8	RORW	57H			6
ADD	16H			6	AND	60H			5

Instruction	Code			Clock Units	Instruction	Code			Clock Units
	1'st byte	2'nd byte	3'rd byte			1'st byte	2'nd byte	3'rd byte	
ADDW	17H			9	ANDW	61H			9
INC	18H			4	OR	62H			6
INCW	19H			6	ORW	63H			9
DEC	1AH			4	NOT	64H			5
DECW	1BH			6	NOTW	65H			7
CALL	20H	Lo adr	Hi adr	4	XOR	66H			6
RET	21H			3	XORW	67H			8
JMP	30H	Lo adr	Hi adr	5	CMP	68H			5
JMR	31H	Data8		4	CMPW	69H			7
JZ	32H	Lo adr	Hi adr	5	LDAW	6AH	Lo adr	Hi adr	6
JNZ	33H	Lo adr	Hi adr	5	STAW	6BH	Lo adr	Hi adr	6

Table B Instruction format, Encoding, and Execution time (Continue)

As an example, consider the instruction ADD. It pops two operands (2 clock units), computes the result (1 clock unit), and pushes it on stack (1 clock unit). This costs 4 units. As another example, consider JZ Label instruction. The Zero flag is taken (1 clock unit), the ALU is used to check if Zero flag is 1 (1 clock unit), and the jump-address is fetched from the code segment in case the value is 1 (1 clock unit). If the jump is made, the cost is 3 units, otherwise, the cost is 2 unit.

Microarchitecture and microstep

LDI val : Load Immediate (Push immediate value)

```
Push  (value) ; SP - -;
```

LDIW val16 : Load Word Immediate (Push word immediate value)

```
Push  (Hi-byte) ; SP - -;
Push  (Lo-byte) ; SP - -;
```

LDD adr16 : Load Direct (Push data from memory)

```
Push  (MEM[adr16]) ; SP - -;
```

LDDW adr16 : Load Direct with 1 word data (Push data from memory)

```
Push  (MEM[adr16+1]) ; SP - -;
Push  (MEM[adr16+1]) ; SP - -;
```

LDA adr16 : Load indirect

```
Push (MEM[adr16]); SP - -;
```

STA adr16 : Store indirect

```
MEM[adr16] = Pop( ); SP + +;
```

STD adr16 : Store direct

```
MEM[adr16] = Pop( ); SP + +;
```

STDW adr16 : Store word direct

```
MEM[adr16] = Pop( ); SP + +;
MEM[adr16+1] = Pop( ); SP + +;
```

IN adr8 : Input port

```
Push ( IO[adr8]); SP - -;
```

OUT adr8 : Output port

```
IO[adr8] = Pop( );  SP + +;
```

MUL : Multiply

```
A = Pop( ); SP + +;
B = Pop( ); SP + +;
C = A * B;  UPDATE-FLAG;
Push( C ); SP - -;
```

MULW : Multiply (word)

```
A = Pop( ); SP + +;
B = Pop( ); SP + +;
C = A * B;  UPDATE-FLAG;
Push( HI(C) ); SP - -;
Push( LO(C) ); SP - -;
```

DIV : Divide

```
A = Pop( ); SP + +;
B = Pop( ); SP + +;
C = A / B ;  UPDATE-FLAG;
Push( C ); SP - -;
```

DIVW : Divide (Word)

```
A = Pop( ); SP + +;
A = Pop( ) << 8 | A;  SP + +;
B = Pop( ); SP + +;
B = Pop( ) << 8 | B;  SP + +;
C = A / B ;  UPDATE-FLAG;
Push( HI(C) ); SP - -;
Push( LO(C) ); SP - -;
```

SUB : Subtract

```
A = Pop( );  SP + +;
B = Pop( );  SP + +;
C = A – B ;  UPDATE-FLAG;
Push (C); SP - -;
```

SUBW : Subtract (word)

```
A = Pop( ); SP + +;
A = Pop( ) << 8 | A ; SP + +;
B = Pop( );  SP + +;
C = A – B;  UPDATE-FLAG;
Push ( HI(C) ); SP - -;
Push ( LO(C) ); SP - -;
```

ADD : Add

```
A = Pop( );  SP + +;
B = Pop( );  SP + +;
C = A + B;  UPDATE-FLAG;
Push( C ); SP - -;
```

ADDW : Add (word)

```
A = Pop( );  SP + +;
A = Pop( ) << 8 | A;  SP + +;
B = Pop( );  SP + +;
B = Pop( ) << 8 | B;  SP + +;
C = A + B;  UPDATE-FLAG;
Push( HI(C) ); SP - -;
Push( LO(C) ); SP - -;
```

INC : Increase

```
A = Pop( ); SP + +;
A = A + 1; UPDATE-FLAG;
Push( A );  SP - -;
```

INCW : Increase (Word)

```
A = Pop( ); SP + +;
A = Pop( ) << 8 | A; SP + +;
A = A + 1; UPDATE-FLAG;
Push( HI(A) );  SP - -;
Push( LO(A) ); SP - -;
```

DEC : Decrease

```
A = Pop( ); SP + +;
A = A – 1;  UPDATE-FLAG;
Push ( A ); SP - -;
```

DECW : Decrease

```
A = Pop( ); SP + +;
A = Pop( ) << 8 | A; SP + +;
A = A – 1;  UPDATE-FLAG;
Push ( HI(A) ); SP - -;
Push ( LO(A) ); SP - -;
```

CALL adr16 : Call subroutine

```
Push ( HI(PC + 3)); SP - -;
Push ( LO(PC + 3)); SP - -;
PC = adr16;
```

RET : Return from subroutine

```
A = Pop( ); SP + +;
PC = Pop( ) << 8 | A ; SP + +;
```

JMP adr16 : Unconditionally jump

```
PC = adr16;
```

JMR adr8 : Jump relate to address –128 to 127

PC = PC + adr8;

JZ adr16 : Jump if Zero flag is 1

IF FZ = 1 , PC = adr16

JNZ adr16 : Jump if Zero flag is 0

IF FZ = 0 , PC = adr16

JO adr16 : Jump if Overflow flag is 1

IF FO = 1 , PC = adr16

JNO adr16 : Jump if Overflow flag is 0

IF FO = 0 , PC = adr16

JC adr16 : Jump if Carry flag is 1

IF FC = 1 , PC = adr16

JNC adr16 : Jump if Carry flag is 0

IF FC = 0 , PC = adr16

JS adr16 : Jump if Sign flag is 1

IF FS = 1 , PC = adr16

JNS adr16 : Jump if Sign flag is 0

IF FS = 0 , PC = adr16

SHL : Shift left 1 bit

```
A = Pop( ); SP + +;
A <<= 1;  UPDATE-FLAG;
Push (A); SP - -;
```

SHLW : Shift left 1 bit (word)

```
A = Pop( ); SP + +;
A = Pop( ) << 8 | A; SP + +;
A <<= 1;  UPDATE-FLAG;
Push ( HI(A)); SP - -;
Push ( LO(A)); SP - -;
```

SHR : Shift right 1 bit

```
A = Pop( ); SP + +;
A >>= 1;  UPDATE-FLAG;
Push (A); SP - -;
```

SHRW : Shift right 1 bit (word)

```
A = Pop( ); SP + +;
A = Pop( ) << 8 | A; SP + +;
A >>= 1;  UPDATE-FLAG;
Push ( HI(A)); SP - -;
Push ( LO(A)); SP - -;
```

ROL : Rotate left 1 bit

```
A = Pop( ); SP + +;
FC = A & 0x80
A <<= 1;
A += FC;
Push (A); SP - -;
```

ROR : Rotate right 1 bit

```
A = Pop( ); SP + +;
FC = A & 0x01
A >>= 1;
A += FC;
Push (A); SP - -;
```

ROLW : Rotate left 1 bit (word)

```
A = Pop( ); SP + +;
A = Pop( ) << 8 | A; SP + +;
FC = A & 0x8000
A <<= 1;
A += FC;
Push ( HI(A)); SP - -;
Push ( LO(A)); SP - -;
```

RORW : Rotate right 1 bit (word)

```
A = Pop( ); SP + +;
A = Pop( ) << 8 | A; SP + +;
FC = A & 0x0001
A >>= 1;
A += FC;
Push ( HI(A) ); SP - -;
Push ( LO(A) ); SP - -;
```

AND : And operator

```
A = Pop( );  SP + +;
B = Pop( ); SP + +;
C = A and B; UPDATE-FLAG;
Push (C);  SP - -;
```

ANDW : And operator (word)

```
A = Pop( );  SP + +;
A = Pop( ) << 8 | A;  SP + +;
B = Pop( ); SP + +;
B = Pop( ) << 8 | B;  SP + +;
C = A and B; UPDATE-FLAG;
Push ( HI(C) );  SP - -;
Push ( LO(C) ); SP - -;
```

OR : Or operator

```
A = Pop( ); SP + +;
B = Pop( ); SP + +;
C = A OR B;   UPDATE-FLAG;
Push ( C ); SP - -;
```

ORW : Or operator (word)

```
A = Pop( ); SP + +;
A = Pop( ) << 8 | A; SP + +;
B = Pop( ); SP + +;
B = Pop( ) << 8 | B;  SP + +;
C = A OR B;   UPDATE-FLAG;
Push ( HI(C) ); SP - -;
Push ( LO(C) ); SP - -;
```

NOT : Not operator

```
A = Pop( ); SP + +;
A = ! A;  UPDATE-FLAG;
Push( A );  SP - -;
```

NOTW : Not operator (word)

```
A = Pop( ); SP + +;
A = Pop( ) << 8 | A; SP + +;
A = ! A;  UPDATE-FLAG;
Push( HI(A) );  SP - -;
Push( LO(A) ); SP - -;
```

XOR : Xor operator

```
A = Pop( );  SP + +;
B = Pop( );  SP + +;
C = A XOR B;   UPDATE-FLAG;
Push ( C );  SP - -;
```

XORW : Xor operator

```
A = Pop( );  SP + +;
A = Pop( ) << 8 | A;  SP + +;
B = Pop( );  SP + +;
B = Pop( ) << 8 | B;  SP + +;
C = A XOR B;   UPDATE-FLAG;
Push ( HI(C) );  SP - -;
Push ( LO(C) );  SP - -;
```

CMP : Compare value with stack data

```
A = Pop( );  SP + +;
B = Pop( );  SP + +;
C = A – B;  UPDATE-FLAG;
```

CMPW : Compare value with stack data (word)

```
A = Pop( );  SP + +;
A = Pop( ) << 8 | A;  SP + +;
B = Pop( );  SP + +;
B = Pop( ) << 8 | B;  SP + +;
C = A – B;  UPDATE-FLAG;
```

The Stack machine Simulator

The Stack Machine simulator evaluated by test1 to test5, that show the source code below. All test program written in assembly language of simulator. To validate the architecture of processor and instruction set . finally, we show the performance of Stack machine simulator. (Cycle Per Instruction = CPI)

```
; -- Filename : Test8.asm
;    Bubble sort with 100 data.
;
        org     0               ; Start at 0x000
[main   ldiw    dFirst
        stdw    ai              ; ai point to first data
        ldiw    dNext
        stdw    aj              ; aj point to next data

[OuterLoop                      ; Outer loop
        ldiw    dLast           ; push dLast(last data ai)
        lddw    ai              ; Push ai to stack
        cmpw                    ; Compare ai with dLast
                                ;     If ai > dLast then not zero
                                ;     If ai = dLast then zero
                                ;     If ai < dLast then negative
        jz      eloop           ; If zero then goto eloop (ai = last time)

[chkaj
; --- If aj = dEnd (last time) must do outer loop
        ldiw    dEnd
        lddw    aj
        cmpw
        jnz     InnerLoop      ; else do inner loop

; --- ai point to next value
        lddw     ai
        incw
        stdw     ai              ; skip ai to next data

; --- skip aj to ai+1
        lddw     ai
        incw
        stdw     aj              ; aj point to ai+1

        jmp     OuterLoop       ; goto OuterLoop

[InnerLoop
; --- Compare [aj] with [ai] if [ai] > [aj] then swap [aj]<->[ai]
        lda     aj
        lda     ai
        cmp
        js      SkipSwap        ; If [ai] > [aj] , not swap. (Sign = 0)

; --- swap
        lda     ai
        lda     aj
        sta     ai
        sta     aj

[SkipSwap
; --- point aj to next data
        lddw    aj
        incw
        stdw    aj
        jmp     chkAj


[eloop  hlt             ; Terminated

; --------------- Data segment start here
        org     900
[ai     DW      0
[aj     DW      0
```

```
        org     1000

[dFirst DB      67              |       DB      116             |       DB      62
[dNext  DB      1               |       DB      115             |       DB      61
        DB      0               |       DB      114             |       DB      60
        DB      6               |       DB      113             |       DB      59
        DB      7               |       DB      112             |       DB      58
        DB      20              |       DB      111             |       DB      57
        DB      30              |       DB      110             |       DB      56
        DB      12              |       DB      109             |       DB      55
        DB      158             |       DB      108             |       DB      54
        DB      157             |       DB      107             |       DB      53
        DB      156             |       DB      106             |       DB      52
        DB      155             |       DB      105             |       DB      51
        DB      154             |       DB      104             |       DB      50
        DB      153             |       DB      103             |       DB      19
        DB      152             |       DB      102             |       DB      18
        DB      151             |       DB      101             |       DB      17
        DB      150             |       DB      100             |       DB      16
        DB      133             |       DB      49              |       DB      15
        DB      132             |       DB      48              |       DB      13
        DB      131             |       DB      47              |       DB      12
        DB      130             |       DB      46              |       DB      11
        DB      129             |       DB      45              |       DB      10
        DB      128             |       DB      44              |       DB      9
        DB      127             |       DB      43              |       DB      2
        DB      126             |       DB      42              |       DB      90
        DB      125             |       DB      41              |       DB      78
        DB      124             |       DB      40              |       DB      28
        DB      123             |       DB      69              |       DB      12
        DB      122             |       DB      68              |       DB      56
        DB      121             |       DB      67              |       DB      34
        DB      120             |       DB      66              |       DB      255
        DB      119             |       DB      65              |[dLast   DB      200
        DB      118             |       DB      64              |[dEnd    DB      0
        DB      117             |       DB      63              |

; -------------------------------------------------------------
        end             ; End of program
```

```
C:\work\stkmc\sample>stkmc
CPU reset ...  success ...
- Stack machine simulator -
- Object code file :test8.hx
\
-HALT-
No. instruction = 75906
No. clock = 389820,CPI = 5.135562
```

Data after sorted.

```
MEM[01000] = 0          |   MEM[01025] = 44         |   MEM[01050] = 67
MEM[01001] = 1          |   MEM[01026] = 45         |   MEM[01051] = 68
MEM[01002] = 2          |   MEM[01027] = 46         |   MEM[01052] = 69
MEM[01003] = 6          |   MEM[01028] = 47         |   MEM[01053] = 78
MEM[01004] = 7          |   MEM[01029] = 48         |   MEM[01054] = 90
MEM[01005] = 9          |   MEM[01030] = 49         |   MEM[01055] = 100
MEM[01006] = 10         |   MEM[01031] = 50         |   MEM[01056] = 101
MEM[01007] = 11         |   MEM[01032] = 51         |   MEM[01057] = 102
MEM[01008] = 12         |   MEM[01033] = 52         |   MEM[01058] = 103
MEM[01009] = 12         |   MEM[01034] = 53         |   MEM[01059] = 104
MEM[01010] = 12         |   MEM[01035] = 54         |   MEM[01060] = 105
MEM[01011] = 13         |   MEM[01036] = 55         |   MEM[01061] = 106
MEM[01012] = 15         |   MEM[01037] = 56         |   MEM[01062] = 107
MEM[01013] = 16         |   MEM[01038] = 56         |   MEM[01063] = 108
MEM[01014] = 17         |   MEM[01039] = 57         |   MEM[01064] = 109
MEM[01015] = 18         |   MEM[01040] = 58         |   MEM[01065] = 110
MEM[01016] = 19         |   MEM[01041] = 59         |   MEM[01066] = 111
MEM[01017] = 20         |   MEM[01042] = 60         |   MEM[01067] = 112
MEM[01018] = 28         |   MEM[01043] = 61         |   MEM[01068] = 113
MEM[01019] = 30         |   MEM[01044] = 62         |   MEM[01069] = 114
MEM[01020] = 34         |   MEM[01045] = 63         |   MEM[01070] = 115
MEM[01021] = 40         |   MEM[01046] = 64         |   MEM[01071] = 116
MEM[01022] = 41         |   MEM[01047] = 65         |   MEM[01072] = 117
MEM[01023] = 42         |   MEM[01048] = 66         |   MEM[01073] = 118
MEM[01024] = 43         |   MEM[01049] = 67         |   MEM[01074] = 119
```

```
MEM[01075] = 120
MEM[01076] = 121
MEM[01077] = 122
MEM[01078] = 123
MEM[01079] = 124
MEM[01080] = 125
MEM[01081] = 126
MEM[01082] = 127
MEM[01083] = 128
```

```
MEM[01084] = 129
MEM[01085] = 130
MEM[01086] = 131
MEM[01087] = 132
MEM[01088] = 133
MEM[01089] = 150
MEM[01090] = 151
MEM[01091] = 152
MEM[01092] = 153
```

```
MEM[01093] = 154
MEM[01094] = 155
MEM[01095] = 156
MEM[01096] = 157
MEM[01097] = 158
MEM[01098] = 200
MEM[01099] = 255
```

Conclusion

Stack Machine shows the others way to construct the stack-base storage, In LIFO (Last in first out) sequence, easy way to cope with data storing in the stack and compact size of program but the lack of parallelism, Stack Machine will be made more cycle clock per instruction. So that in ours test program show the large value of CPI because the push or pop command can not get value at the same time.

So, we accept the concept of stack machine that process in sequencing, use push and pop command for manage the input /output data. The advantage of this machine is simple in memory architecture, a few address registers, compact source code and no operand field. It effort to use in block-sequence language.

The way to improve stack machine‘s performance is

- pipe-line
- high speed memory
- high speed bus