1 Instruction Set of dVM IR

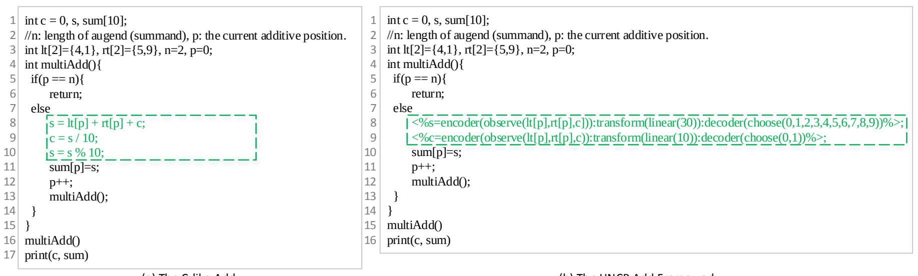
Table 1. Instruction Set.

Instruction	Description	Micro Instruction Action
CALL L	Save PC+1(return addr) to stack R, then jump to L.	$P_r \leftarrow P_r+1$; $R[P_r] \leftarrow P_c+1$; $P_c \leftarrow L$
RET	Match with CALL. Pop value ((return addr)) from top of R, then jump to this value.	$P_c \leftarrow R[P_r]$; $P_r \leftarrow P_r-1$
GOTO L	Jump to L	$P_c \leftarrow L$
GOTOF L	if-then-else. If condition is false, jump to L, else do PC++; The condition is the value of top of Stack S.	**if** $S[P_s]$==False **then** $P_c \leftarrow L$ **else** $P_c \leftarrow P_c+1$; $P_s \leftarrow P_s-1$
DO	Copy from S to R at the begin of loop. Top of S is loop counter; The secondary is loop limit.	$P_r \leftarrow P_r+2$; $R[P_r] \leftarrow S[P_s]$; $P_s \leftarrow P_s-1$; $R[P_r-1] \leftarrow S[P_s]$; $P_s \leftarrow P_s-1$; NEXT
LOOP L	Increase top of R. If top of R is less than secondary, jump to L, else exit the loop.	$R[P_r] \leftarrow R[P_r]+1$; **if** $R[P_r]<R[P_r-1]$ **then** $P_c \leftarrow L$ **else** $P_c \leftarrow P_c+1$
ENDLOOP	Clear the parameters for 'do..loop endloop' in R.	$P_r \leftarrow P_r-2$; NEXT
NEXT	Step forward explicitly.	$P_c \leftarrow P_c+1$
INC	Increase top of S by 1.	$S[P_s] \leftarrow S[P_s]+1$; NEXT
DEC	Decrease top of S by 1.	$S[P_s] \leftarrow S[P_s]-1$; NEXT
SWAP	Swap the top and secondary element of S.	$t \leftarrow S[P_s]$; $S[P_s] \leftarrow S[P_s-1]$; $S[P_s-1] \leftarrow t$
+	Add the top and secondary element of S.	$t1 \leftarrow S[P_s]$; $P_s \leftarrow P_s-1$; $t2 \leftarrow S[P_s]$; $S[P_s] \leftarrow ADD(t1,t2)$;
-	Sub the top by the secondary element of S.	$t1 \leftarrow S[P_s]$; $P_s \leftarrow P_s-1$; $t2 \leftarrow S[P_s]$; $S[P_s] \leftarrow SUB(t1,t2)$;
*	Multiply the top by the secondary element of S.	$t1 \leftarrow S[P_s]$; $P_s \leftarrow P_s-1$; $t2 \leftarrow S[P_s]$; $S[P_s] \leftarrow MUL(t1,t2)$;
/	Divide the top by the secondary element of S.	$t1 \leftarrow S[P_s]$; $P_s \leftarrow P_s-1$; $t2 \leftarrow S[P_s]$; $S[P_s] \leftarrow DIV(t1,t2)$;
==	Test if the top of S equals the secondary.	$t1 \leftarrow S[P_s]$; $P_s \leftarrow P_s-1$; $t2 \leftarrow S[P_s]$; $S[P_s] \leftarrow EQ(t1,t2)$;
>	Test if the top of S is greater than the secondary.	$t1 \leftarrow S[P_s]$; $P_s \leftarrow P_s-1$; $t2 \leftarrow S[P_s]$; $S[P_s] \leftarrow GT(t1,t2)$;
<	Test if the top of S is less than the secondary.	$t1 \leftarrow S[P_s]$; $P_s \leftarrow P_s-1$; $t2 \leftarrow S[P_s]$; $S[P_s] \leftarrow LT(t1,t2)$;
SAVE	Save a temporary evaluation value from top of S to heap H.	$a \leftarrow S[P_s]$; $P_s \leftarrow P_s-1$; $v \leftarrow S[P_s]$; $P_s \leftarrow P_s-1$; $H[a] \leftarrow v$; NEXT
LOAD	Load the value of a variable at heap H to stack S.	$a \leftarrow S[P_s]$; $S[P_s] \leftarrow R[a]$; NEXT
HALT	The state of the machine will keep.	$P_c \leftarrow P_c$

2 Task Examples

Tasks of Addition and Sort, by C and HNCP.

(1) Elementary Multidigit Addition



```
int c = 0, s, sum[10];
//n: length of augend (summand), p: the current additive position.
int lt[2]={4,1}, rt[2]={5,9}, n=2, p=0;
int multiAdd(){
  if(p == n){
        return;
  else
        s = lt[p] + rt[p] + c;
        c = s / 10;
        s = s % 10;
        sum[p]=s;
        p++;
        multiAdd();
  }
}
multiAdd()
print(c, sum)
```

(a) The C-like Add.

```
int c = 0, s, sum[10];
//n: length of augend (summand), p: the current additive position.
int lt[2]={4,1}, rt[2]={5,9}, n=2, p=0;
int multiAdd(){
  if(p == n){
        return;
  else
        <%s=encoder(observe(lt[p],rt[p],c])):transform(linear(30)):decoder(choose(0,1,2,3,4,5,6,7,8,9))%>;
        <%c=encoder(observe(lt[p],rt[p],c)):transform(linear(10)):decoder(choose(0,1))%>;
        sum[p]=s;
        p++;
        multiAdd();
  }
}
multiAdd()
print(c, sum)
```

(b) The HNCP Add Framework.

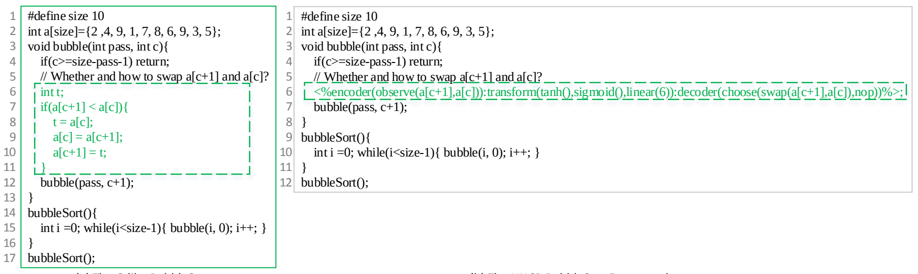
Fig. 1. Demo of Add.

It's step by step single digit addition from the least significant digit position to the most. Each single digit addition is calculating high carry and sum based on low carry, augend and summand, and the multi-digit addition is recursion of single digit addition. In Fig. 1, the left is the C implementation, and the right is the HNCP framework, in which, the burden of computing of sum and carry are represented by HNCP's neural primitive. As shown in this figure, the behaviours writen by programmers in the dotted line frame of Fig. 1(a) are the counterpart of the Fig. 1(b)'s dotted line frame which can learn these behaviours automatically by training from data, therefore the programers do not need to memory the rules of getting sum or carry.

(2) BubbleSort

As shown in Fig. 2, the neural net component of encoder-transform-decoder primitive in the dotted line frame on the figure's right side takes the burden of deciding when and how to swap two elements off human. That is, the behaviours in the dotted line frame of Fig. 2(a) are generated automatically in the HNCP's BubbleSort version of Fig. 2(b).

Fig. 2 shows an example program. In this figure, the right is the HNCP version and the left is the C implementation. Programmers have to describe every aspect including dirty details accurately in the C version, however in HNCP, programmers only provide recursive structure of bubble sorting, and submit the behavior details of decribing whether and how to swap to neural network components to generate automatically, as shown in the dotted line frames of this figure.



```
#define size 10
int a[size]={2 ,4, 9, 1, 7, 8, 6, 9, 3, 5};
void bubble(int pass, int c){
    if(c>=size-pass-1) return;
    // Whether and how to swap a[c+1] and a[c]?
    int t;
    if(a[c+1] < a[c]){
        t = a[c];
        a[c] = a[c+1];
        a[c+1] = t;
    }
    bubble(pass, c+1);
}
bubbleSort(){
    int i =0; while(i<size-1){ bubble(i, 0); i++; }
}
bubbleSort();
```

(a) The C-like BubbleSort.

```
#define size 10
int a[size]={2 ,4, 9, 1, 7, 8, 6, 9, 3, 5};
void bubble(int pass, int c){
    if(c>=size-pass-1) return;
    // Whether and how to swap a[c+1] and a[c]?
    <%encoder(observe(a[c+1],a[c])):transform(tanh(),sigmoid(),linear(6)):decoder(choose(swap(a[c+1],a[c]),nop))%>;
    bubble(pass, c+1);
}
bubbleSort(){
    int i =0; while(i<size-1){ bubble(i, 0); i++; }
}
bubbleSort();
```

(b) The HNCP BubbleSort Framework.

Fig. 2. Demo of Sort. The left side is the C-like code, and the right is the HNCP Framework

which is incomplete, but can be trained to generate the complete deterministic program with the same behaviours as the left side automatically.