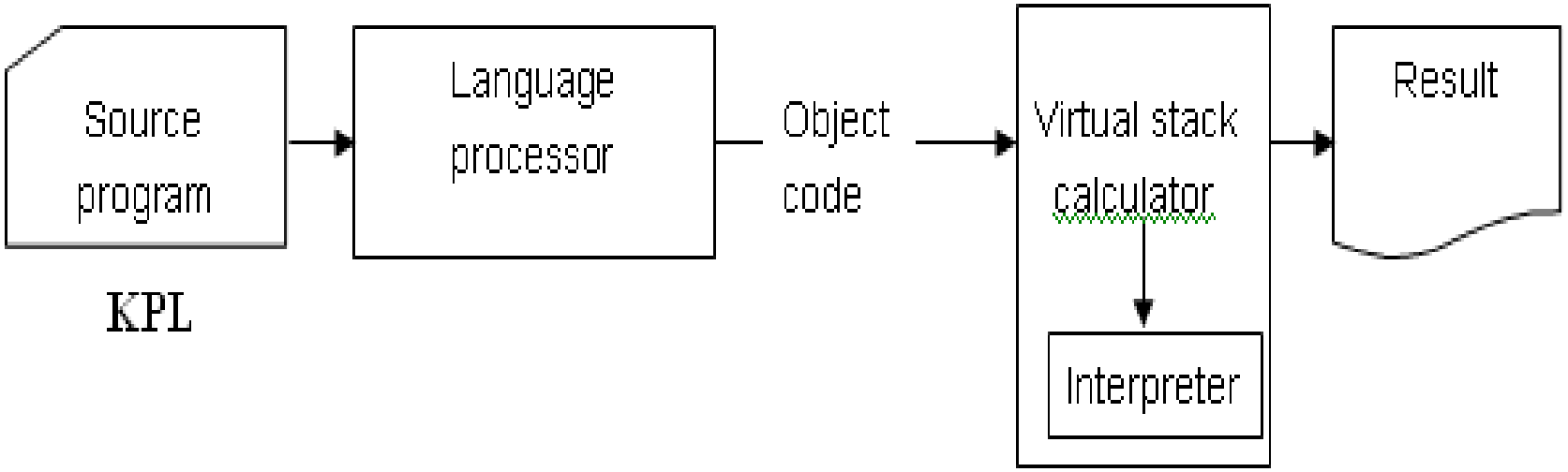
Unit 13
Code Generation

Program execution by KPL



Input of Code Generator

- Immediate code
- Source code

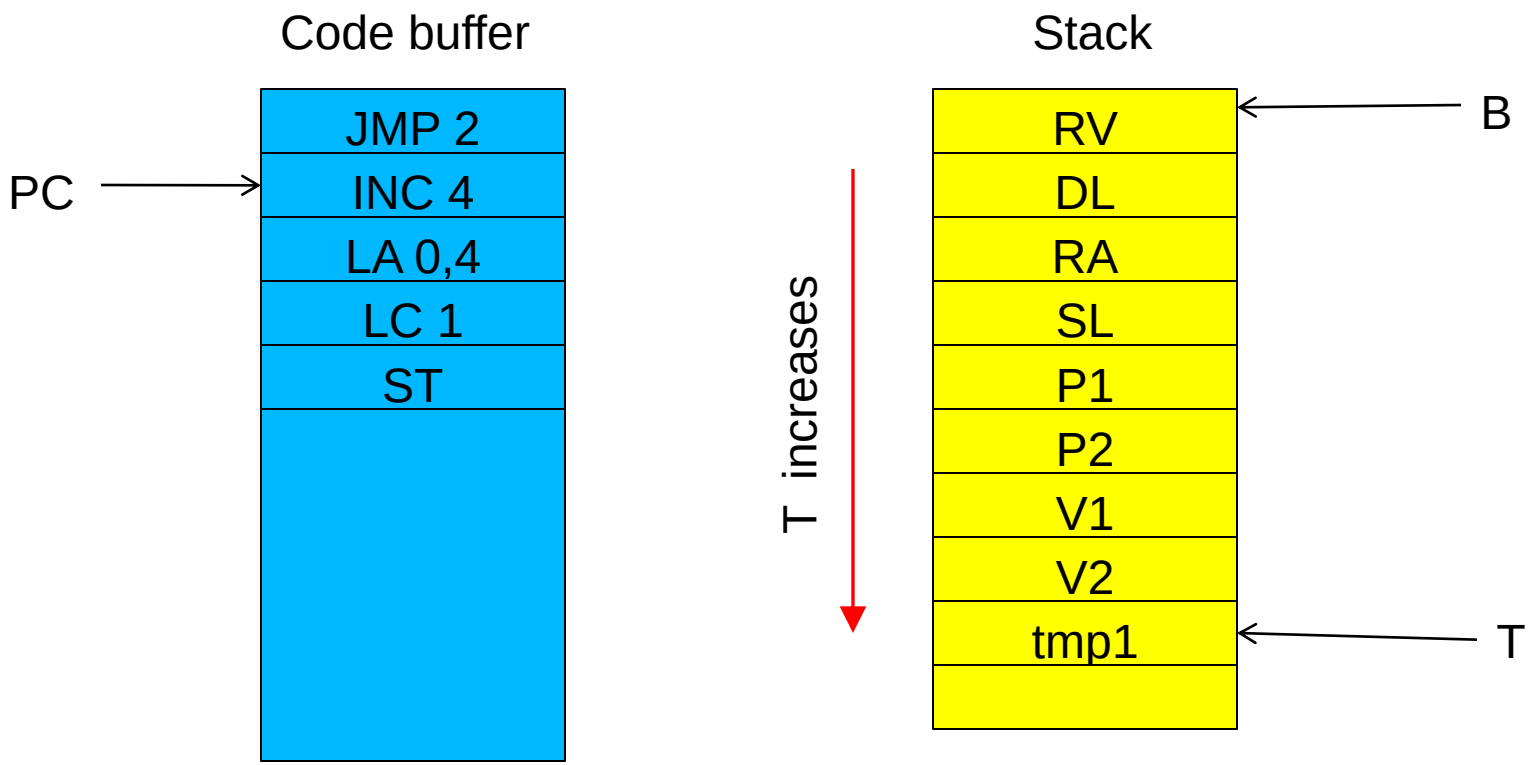
Output of Code Generator

- Machine (executable) code
- Assembly code
- ***Intermediate code for a virtual machine***

Stack calculator

- Stack calculator is a computing system
  - Using stack to store intermediate results during computation process.
  - Simple organization
  - Simple instruction set
- Stack calculator consists of 2 memory areas
  - Code buffer: containing execution code corresponding to source program
  - Stack: storing intermediate results

Stack calculator



Stack calculator

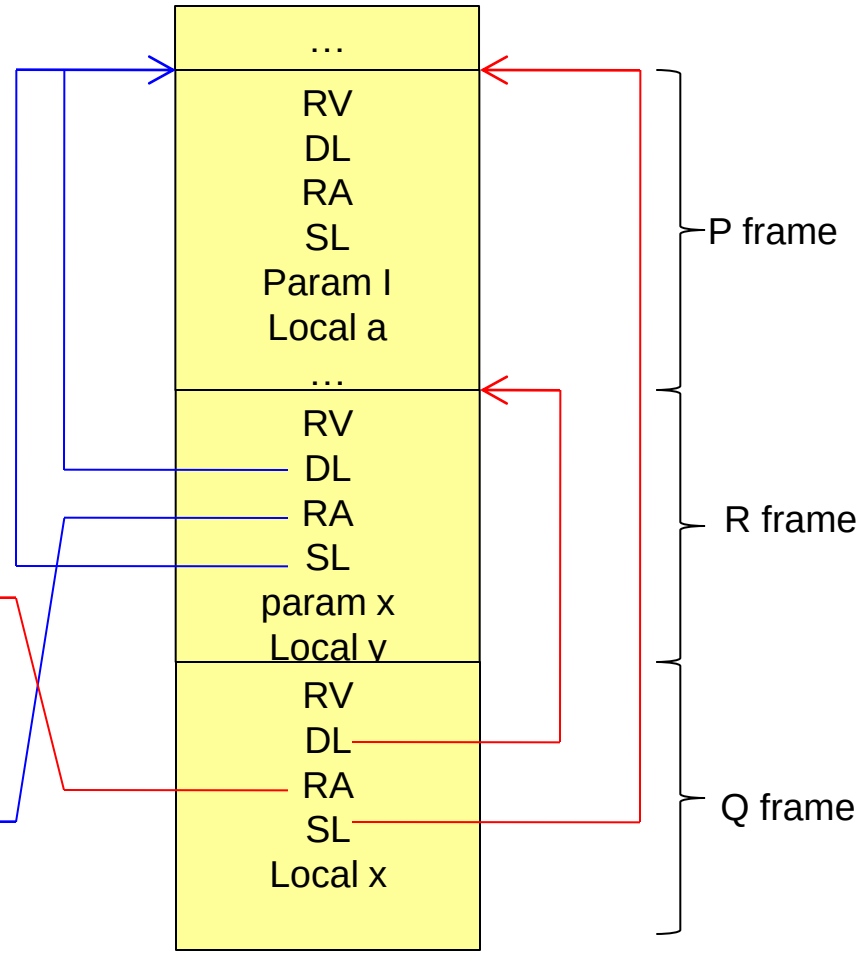
- Registers
  - PC (program counter): pointing to currently being executed instruction on Code buffer
  - B (base): pointing to the base address of data area of active block on Stack. Local varialbes are accessed via B
  - T (top): pointing to Stack's top element

Stack calculator

- Activation record / Stack frame
  - Is the memory area allocated to every function, procedure and the main program when it is activated (becoming active block)
    - Storing parameters' values
    - Storing local variables's values
    - Other information
      - Return value – RV
      - Dynamic link – DL
      - Return address – RA
      - Static link – SL
  - A function/procedure may have several Stack frames on Stack

Stack Calculator

```
Procedure P(I : integer);
  Var a : integer;
  Function Q;
    Var x : char;
    Begin
      …
       return
    End;
  Procedure R(X:
integer);
    Var y : char;
    Begin
      …
      y = Call Q;
      …
    End;
  Begin
  …
```



Stack calculator

- RV (return value): stores return value of a function
- DL (dynamic link): is the base address of caller's Stack frame. DL is used to recover caller's context when the callee ends.
- RA (return address): address of caller's instruction that would be executed when callee ends.
- SL (static link): base address of outer's Stack frame. SL is useful when we track non-local variables.

Stack calculator

- Instruction set

LA	Load Address	t:=t+1; s[t]:=base(p)+q;
LV	Load Value	t:=t+1; s[t]:=s[base(p)+q];
LC	Load Constant	t:=t+1; s[t]:=q;
LI	Load Indirect	s[t]:=s[s[t]];
INT	Increment T	t:=t+q;
DCT	Decrement T	t:=t-q;

Stack calculator

- Instruction set

op	p	q

J	Jump	pc:=q;
FJ	False Jump	if s[t]=0 then pc:=q; t:=t-1;
HL	Halt	Halt
ST	Store	s[s[t-1]]:=s[t]; t:=t-2;
CALL	Call	s[t+2]:=b; s[t+3]:=pc; s[t+4]:=base(p); b:=t+1; pc:=q;
EP	Exit Procedure	t:=b-1; pc:=s[b+2]; b:=s[b+1];
EF	Exit Function	t:=b; pc:=s[b+2]; b:=s[b+1];

Stack calculator

- Instruction set

RC	Read Character	read one character into s[s[t]]; t:=t-1;
RI	Read Integer	read integer to s[s[t]]; t:=t-1;
WRC	Write Character	write one character from s[t]; t:=t-1;
WRI	Write Integer	write integer from s[t]; t:=t-1;
WLN	New Line	CR & LF

Stack calculator

- Instruction set

op	p	q

AD	Add	t:=t-1; s[t]:=s[t]+s[t+1];
SB	Subtract	t:=t-1; s[t]:=s[t]-s[t+1];
ML	Multiply	t:=t-1; s[t]:=s[t]*s[t+1];
DV	Divide	t:=t-1; s[t]:=s[t]/s[t+1];
NEG	Negative	s[t]:=-s[t];
CV	Copy Top of Stack	s[t+1]:=s[t]; t:=t+1;

Stack calculator

- Instruction set

op	p	q

EQ	Equal	t:=t-1; if s[t] = s[t+1] then s[t]:=1 else s[t]:=0;
NE	Not Equal	t:=t-1; if s[t] != s[t+1] then s[t]:=1 else s[t]:=0;
GT	Greater Than	t:=t-1; if s[t] > s[t+1] then s[t]:=1 else s[t]:=0;
LT	Less Than	t:=t-1; if s[t] < s[t+1] then s[t]:=1 else s[t]:=0;
GE	Greater or Equal	t:=t-1; if s[t] >= s[t+1] then s[t]:=1 else s[t]:=0;
LE	Less or Equal	t:=t-1; if s[t] <= s[t+1] then s[t]:=1 else s[t]:=0;

Generate code for ASSIGN statement

v := exp

```
<code of l-value v>  // load address of v
<code of exp>        // load value of of exp
ST
```

Generate code for IF statement

If <cond> Then statement;

```
  <code of cond>     // load value of condition
  FJ L
  <code of statement>
L:
  …
```

If <cond> Then st1 Else st2;

```
  <code of cond>     // load value of condition
  FJ L1
  <code of st1>
  J L2
L1:
  <code of st2>
L2:
  …
```

Generate code for WHILE statement

While <cond> Do statement

```
L1:
  <code of cond> // load value of condition
  FJ L2
  <code of statement>
  J L1
L2:
  …
```

Generate code for FOR statement

For v := exp1 to exp2 do statement

```
  <code of l-value v>
  CV   // copy top of stack – duplicate address of v
  <code of exp1>
  ST   // store original value of v
L1:
  CV
  LI   // get value of v
  <code of exp2>
  LE
  FJ L2
  <code of statement>
  CV;CV;LI;LC 1;AD;ST;  // increase v's value by 1
  J L1
L2:
  DCT 1
  …
```