COMPILER DESIGN

UNIT-5

Runtime Environments

feedback/corrections: vibha@pesu.pes.edu

VIBHA MASTI

ISSUES in DESIGN of CODE GENERATOR

1. Input to the code generator
   (a) 3A Representations: quad, triple, indirect triples
   (b) VM Representations: bytecodes, stack-machine code
   (c) Linear Representations: postfix
   (d) Graphical Representations: syntax tree, DAG

2. Target program
   (a) RISC
   (b) CISC
   (c) Stack-based

   (a) Absolute ML Program
   (b) Relocatable ML Program
   (c) Assembly program

3. Instruction selection
   (a) Level of IR
   (b) Nature of ISA
   (c) Desired quality

4. Register allocation and assignment
   (a) Allocation: select set of vars to reside in registers
   (b) Assignment: pick specific register for variable

5. Evaluation order

Code Generation

- IR + symbol table $\xrightarrow{\text{code generation}}$ target program (machine code)

- Primary tasks
  1. Instruction selection
  2. Register allocation and assignment
  3. Instruction ordering

Representation of IR

1. 3AC Representations
- quadruples, triples, indirect triples

2. VM Representations
- bytecodes
- stack-machine code

3. Linear Representations
- postfix

4. Graphical Representations
- syntax trees
- DAGs

IR Assumptions

1. IR is low-level
2. No errors

Representation of Target Machine Code

1. RISC
2. CISC
3. Stack-based architecture

1. RISC Machines

- Many registers
- 3AC instructions
- Simple addressing modes
- Simple ISA

- Eg: ARM, Alpha, ARC, MIPS, AVR, PA-RISC, PIC, SPARC

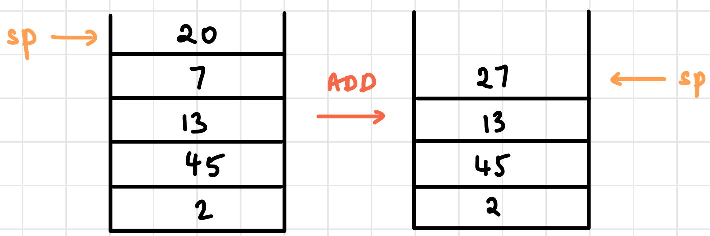
2. CISC Machines

- Few registers
- 2AC instructions
- Various addressing modes
- Several register classes
- Variable length instructions
- Instructions with side effects

- Eg: Intel x86, AMD, System/360, VAX, PDP-11, Motorola 68000 family

3. Stack-Based Architecture

- Revived with JVM



POP 20
POP 7
ADD result, 20, 7
PUSH result

Target Program

1. Absolute Machine Language Program

- Fixed location in memory
- Quick

2. Relocatable Machine Language Program

- Aka object module
- Subprograms compiled separately
- Object modules can be linked together and loaded for execution by a linking loader
- Compiler may have to provide explicit relocation info to the loader

3. Assembly Language Program

- Symbolic instructions as output
- Use assembler to generate machine code

Instruction Selection

- Complexity depends on
  1. Level of IR
  2. Nature of ISA
  3. Desired quality of generated code

1. Level of IR

- High level IR → MC : each IR statement translated to multiple MC instructions (needs further optimization)
- Low level IR → MC : more efficient code

2. Nature of ISA

- Uniformity and completeness
  - Uniformity: all triple addresses etc.
  - Complete: use all registers for any operation

3. Desired quality of generated code

- Efficiency, remove redundancy
- Can design skeleton to define target code
- Eg: 3AC of the form $x=y+z$ can be

```
LD R0, y
ADD R0, R0, z
ST x, R0
```

Register Allocation

- Minimize no. of load/store instructions
- Register allocation and assignment
  decide vars   specific allocations
- Some operations require specific registers (mult & div register pairs)

Consider the two three-address code sequences in Fig. 8.2 in which the only difference in (a) and (b) is the operator in the second statement. The shortest assembly-code sequences for (a) and (b) are given in Fig. 8.3.

```
t = a + b        t = a + b
t = t * c        t = t + c
t = t / d        t = t / d

   (a)              (b)
```

Figure 8.2: Two three-address code sequences

```
L    R1,a        L    R0, a
A    R1,b        A    R0, b
M    R0,c        A    R0, c
D    R0,d        SRDA R0, 32
ST   R1,t        D    R0, d
                 ST   R1, t

   (a)              (b)
```

Figure 8.3: Optimal machine-code sequences

Ri stands for register i. SRDA stands for Shift-Right-Double-Arithmetic and **SRDA R0,32** shifts the dividend into R1 and clears R0 so all bits equal its sign bit.

Evaluation Order

- Order can affect efficiency

HYPOTHETICAL TARGET MACHINE MODEL

- **Our target computer models a three-address machine** with
  1. Load and Store operations,
  2. Computation operations,
  3. Jump operations, and
  4. Conditional jumps.
- **The underlying computer is a byte-addressable machine with n general-purpose registers, R0, R1, . . . , R_{n-1}.**
- To avoid hiding the concepts in a myriad of details, **we shall use a very limited set of instructions and assume that all operands are integers.**
- **Most instructions consists of an operator, followed by a target, followed by a list of source operands.**

Instructions

1. Load   LD dest (reg), src (mem)
2. Store   ST dest (mem), src (reg)
3. Move   MOV R1 (dest), R2 (src)
4. Computations   op, dest, src1, src2
   - ADD
   - SUB
   - MUL
   - DIV

5. Unconditional jumps   BR   L

6. Conditional jumps   Bcond   R, L

cond: LTZ
GTZ
EZ
LTEZ
GTEZ

Addressing Modes

1. Direct

LD   R1, a

2. Index

$x = a[i] \longrightarrow$
$t1 = 4 * i$
$t2 = a[t1]$
$x = t2$

op	dest	src1	src2
LD	R1	i	
MUL	R1	R1	#4
MOV	R2	R1(a)	
ST	x	R2	

3. Indirect

$x = *p \longrightarrow$
$t1 = *p$
$x = t1$

op	dest	src1	src2
LD	R1	p	
MOV	R2	0(R1)	
ST	x	R2	

4. Immediate

$$a = 100$$

op	dest	src1	src2
MOV	R1	#100	
ST	a	R1	

Q: Generate target code for the following

(a) $x = 1$

```
MOV  R0, #1
ST   x, R0
```

(b) $x = a + b$

```
LD   R0, a
LD   R1, b
ADD  R1, R1, R0
ST   x, R1
```

(c) $x = b * c$
$y = a + x$

```
LD   R0, b
LD   R1, c
MUL  R1, R1, R0      // x = b*c
```

```
LD   R2, a
ADD  R2, R2, R1      // y = a+x
ST   x, R1
ST   y, R2
```

(d) $x = a[i]$ Assume a, b arrays with 4-byte elements

$y = b[j]$

$a[i] = y$

$b[j] = x$

```
LD   R0, i
MUL  R0, R0, #4       // R0 = 4*i
MOV  R1, R0(a)        // R1 = a+4*i
LD   R2, j
MUL  R2, R2, #4       // R2 = 4*j
MOV  R3, R2(b)        // R3 = b+4*j

ST   x, R1
ST   y, R3

ST   R0(a), R3
ST   R2(b), R1
```

Q: Generate 3AC and target code for

$a[i] = c$

3AC

```
t1 = i*4
a[t1] = c
```

Target

```
LD   R0, i
MUL  R0, R0, #4
LD   R1, c
ST   R0(a), R1
```

Q: Generate target code for

if $x < y$ goto L

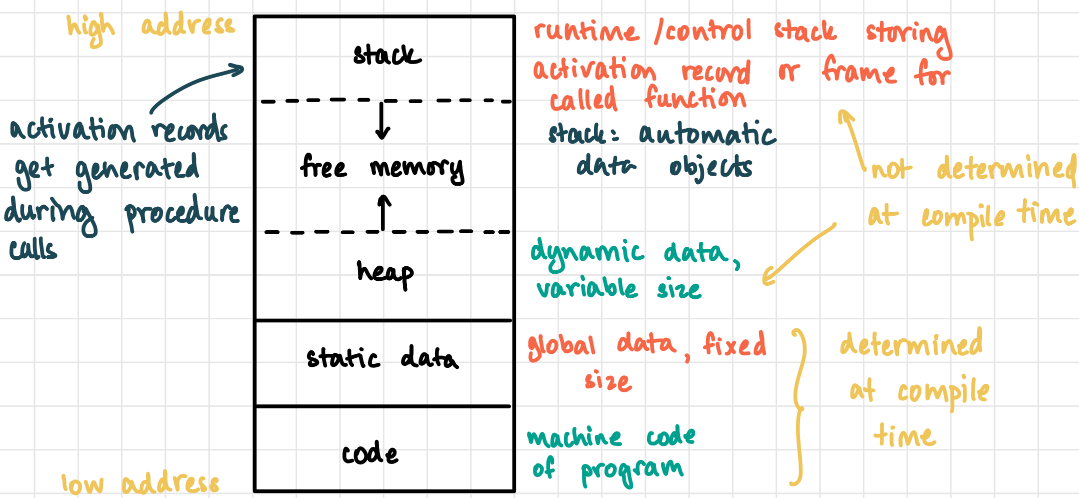
```
LD R0, x
LD R1, y
SUB R0, R0, R1
BLZ R0, M
```

M: equivalent machine instruction for label L

TARGET CODE WITH PROCEDURES

- Need mechanisms for
  - Passing arguments
  - Local storage
  - Returning results
  - Linking control

Memory Layout of Executable Program



Static allocation

- Whenever procedure call is made, store return address in AR of callee
- Then branch to code area of callee
- Once done with callee function, branch to address stored in AR of callee

Q:

	code	AR
c()	100	300
p()	200	364

Assume each 3AC is 20 bytes

```
c() {
    action
    action
    call p
    action
    halt
}

p() {
    action
    return
}
```

Assuming static allocation, provide target code

Code Area (contains procedure code)	Static Area (Contains Activation Record)
//code for c()	//Activation Record of c()
100 : ACTION	300 :
120 : ACTION	
140 : ST 364, 160	
152 : BR 200	
160 : ACTION	
180 : HALT	
//procedure for p()	//Activation Record of p()
200 : ACTION	364 : 160
220 : BR *364	

any 3AC

return add: #here + 20

assume 4 bytes each

branch to content of 364

Assume, Target code for action : ACTION and it takes 20 bytes

Q: Generate target code.

Code for p() at 100
Code for q() at 300
AR for p() at 400
AR for q() at 600

Assume m, n, x represent addresses

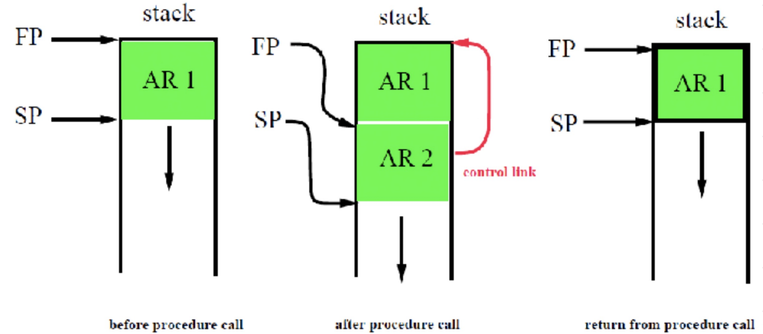
```
// procedure p()
m=5
n=m*2
call q
halt
```

```
// procedure q()
x = 2*x
return
```

Code Area	Static Area
// code for p()	// AR for p()
100: MOV R1, #5 use 0 bytes	400:
108: ST m, R1	
116: MUL R1, R1, #2	
124: ST n, R1	
132: ST 600, #152	
144: BR 300	
152: HALT	
// code for q()	// AR for q()
300: LD R1, x	600: 152
308: MUL R1, R1, #2	
316: ST x, R1	
324: BR *600 or 0(600)	

Stack allocation

- Function calls: stack of activation records / frames
- Stack pointer (SP) register points to top of stack
- Frame pointer (FP) register points to start of current activation record (AR)



Activation Record

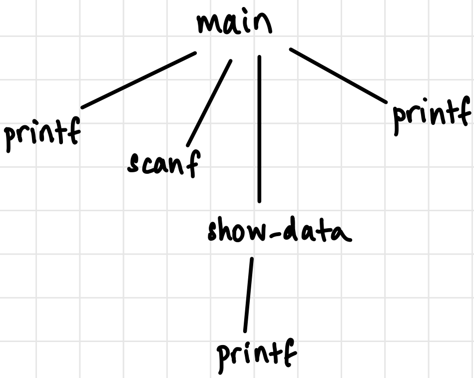
- Data about the execution of a procedure
- Lifetime of an activation: time b/w first and last steps of a procedure

Activation Tree

- Root node: activation of main procedure
- Each node: 1 activation
- Left to right in order of calling (depends on runtime and not just the compile time)
- Child must finish executing before execution to its right can begin

Q: Draw activation tree

```
int main() {
    printf("Enter your name: ");
    scanf("%s", username);
    show_data(username);
    printf("Press any key to continue..");
     :
}

int show_data(char *user) {
    printf("Your name is %s\n", user);
    return 0;
}
```



Note: Activation Tree

- Flow of control: DFS traversal
- Sequence of procedure calls: preorder traverseal
- Sequence of returns: postorder traversal

Meta Language (ML)

- General-purpose functional programming language
  - Only constants
- Supports higher order functions
- Deduces types at compile time

Syntax

1. Variable definition

val (name) = (expression)

2. Function definition

fun <name> (<arguments>) = <body>

3. Function bodies

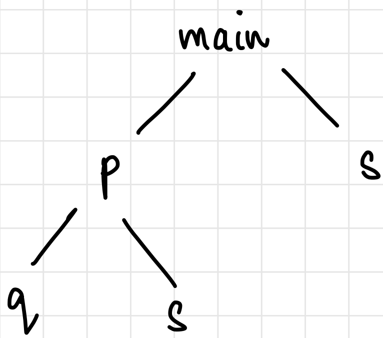
let <list of definitions> in $\underbrace{\text{<statements>}}_{\text{executed}}$ end

Q: Provide possible activations

```
fun main() =
let
    fun s() =
        ... .....
    fun p() =
        let
            fun q() =
                .... ......
        in
            ... q
            ... s
        end
in
    ... p()
    ... s()
end
```

execution order (q, s)

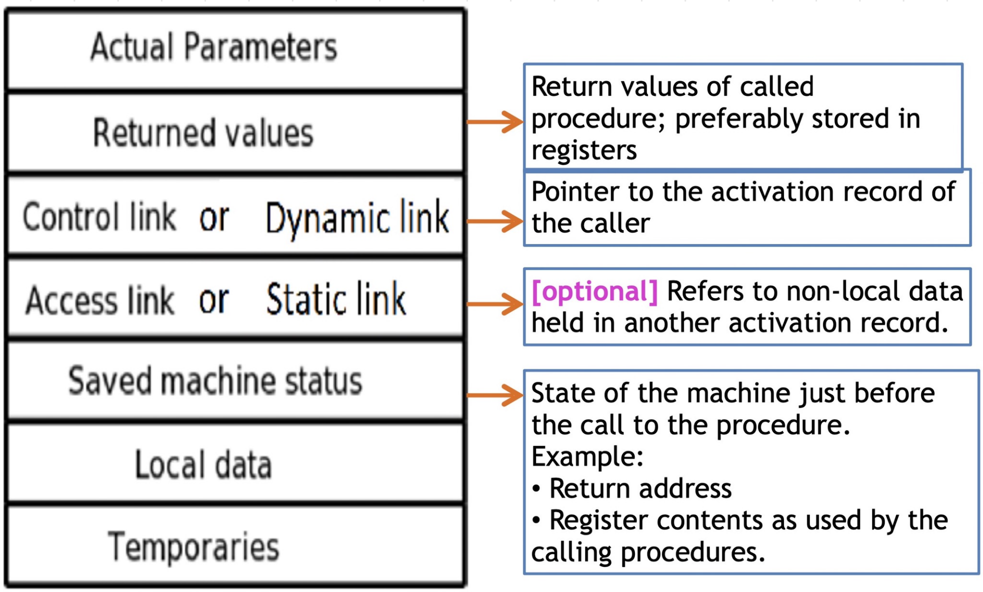
execution order (p(), s())



Activations:

- enter main
  - enter p
    - enter q
    - exit q
    - enter s
    - exit s
  - exit p
  - enter s
  - exit s
- exit main

Activation Record



function in which it was created

↓ rest of stack (will grow downwards)

1. Calling sequence

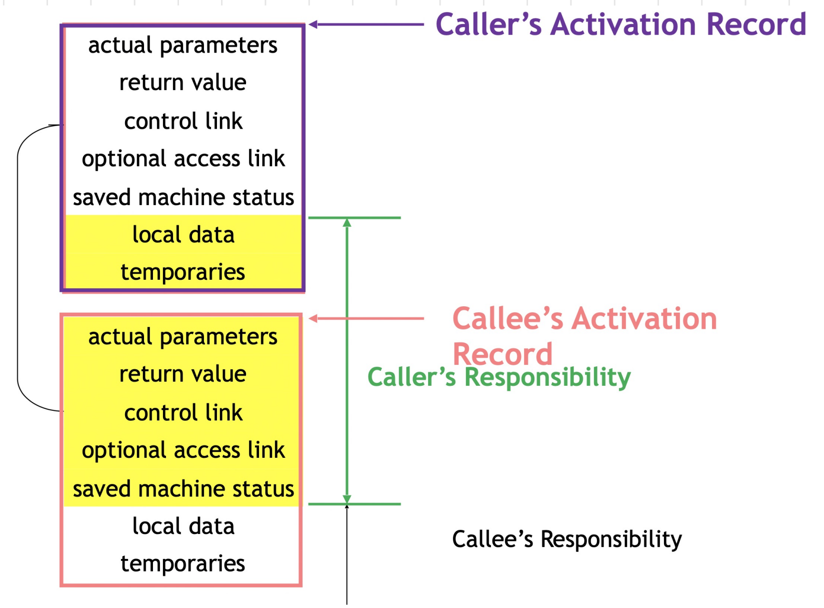
- Code that allocates activation record on stack

- Steps
  1. Caller evaluates actual parameters
  2. Caller stores return address, SP in callee's AR
  3. Caller increments top-sp to point to corresponding point in callee's AR
  4. Callee saves registers and other info
  5. Callee allocates and initializes local data
  6. Callee begins execution

2. Return sequence

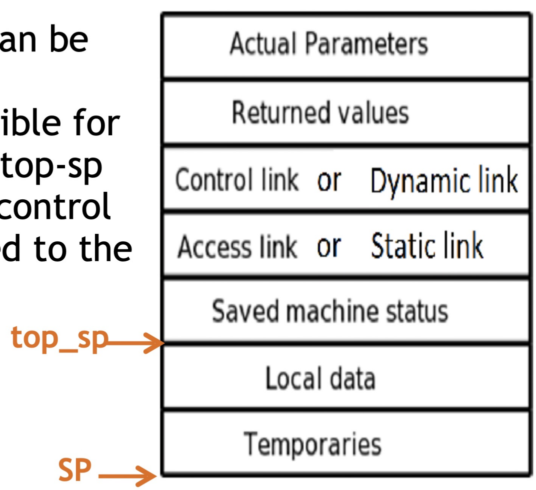
- Code that restores state of the machine so that the calling procedure can continue execution

- Steps
  1. Callee places return value in AR
  2. Callee restores old values of SP and top-sp and other registers and then branches to return address



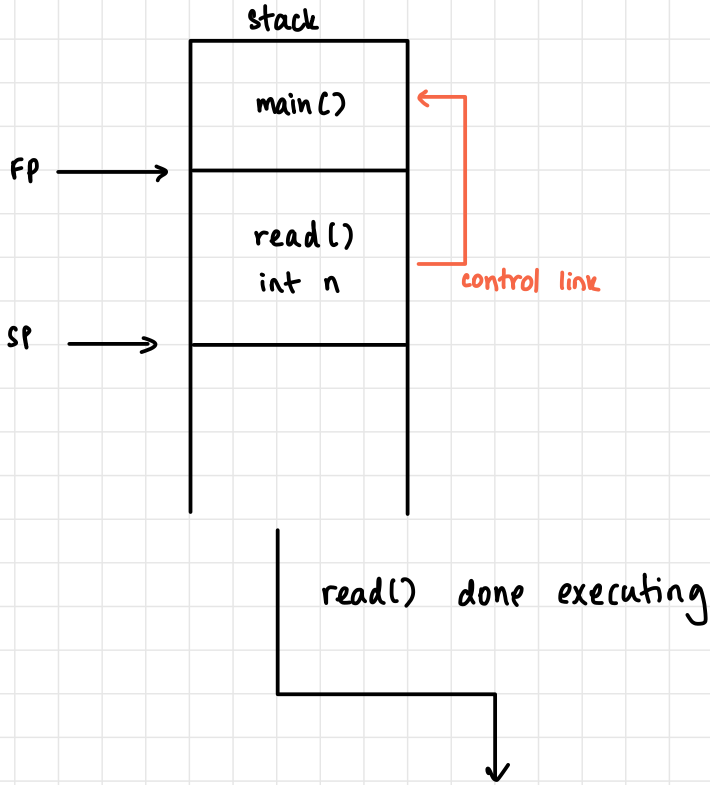
- top-sp: end of fixed fields

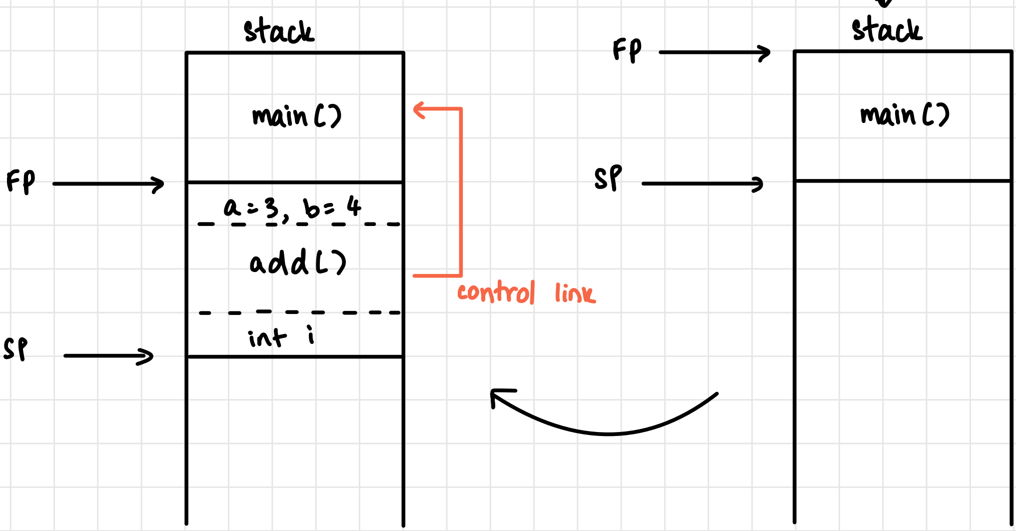
caller can be made responsible for setting top-sp before control is passed to the callee



Activation Record

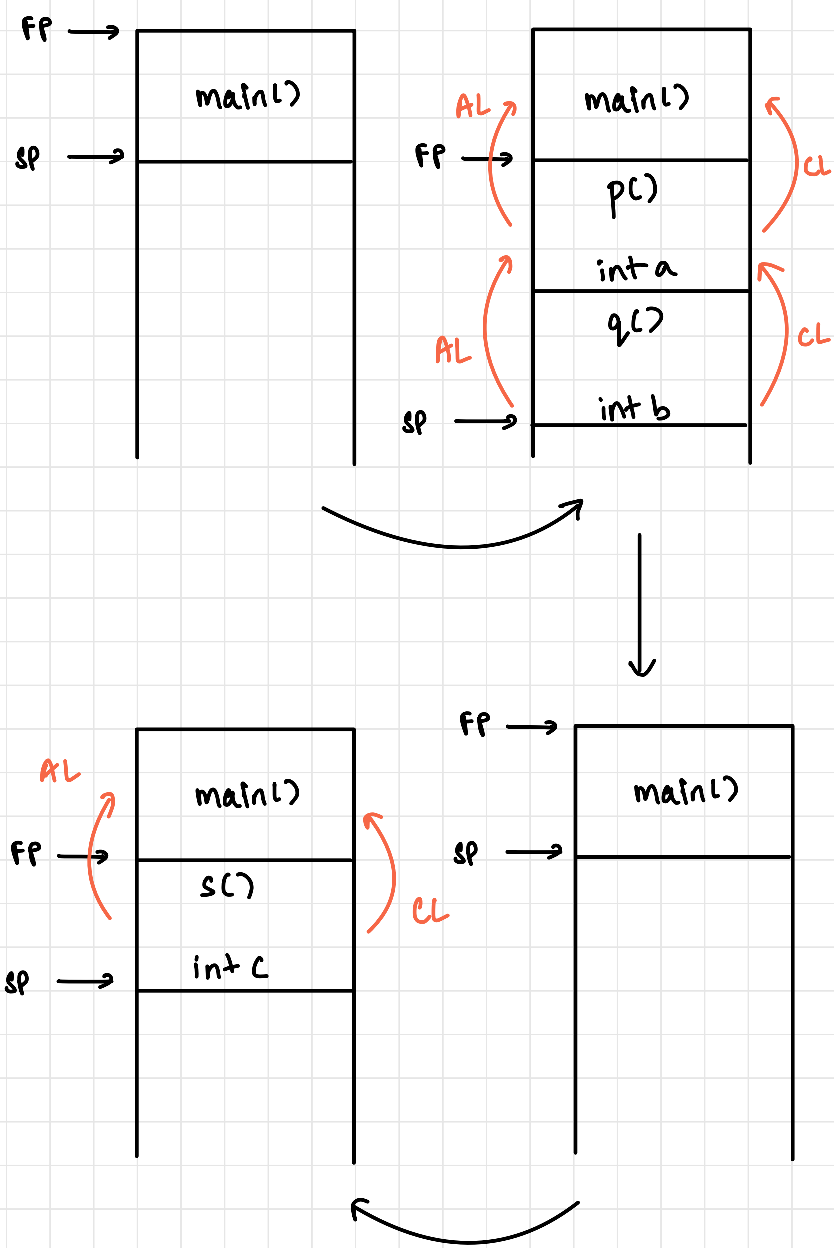
```
main () {
    read();
    add(3, 4);
}

read () {
    int n;
    ...
    return;
}

add (a, b) {
    int i;
    ...
}
```





Q: show stack during execution

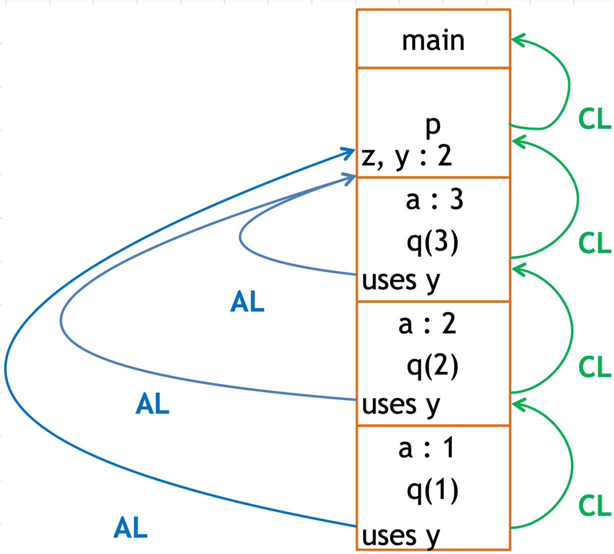
```
main(){
    p(){
        int a;
        q(){
            int b
            ...
        }
        q();
    }

    s(){
        int c;
        ...
    }

   p();
   s();
}
```



Access Links in Nested Procedures

- Function that defines another function ; all locals of defining function accessible to defined function
- Nesting depth of a function : level of nesting of a procedure

```
main() {        // ND(main) = 1
  p() {      // ND(p) = 2
    int z, y = 2;
    int q(int a) {      // ND(q) = 3
    if  (a=1)
        return 1;
    else
        return(a + y + q(a-1));
     }

    z = q(3);
     }

  p();
 }
```

DISPLAYS

- Array of pointers to ARs — called displays
- Avoid long chains of ALs
- d[i]: pointer to activation record at ND i
  - if more than one AR at ND i, pointer to other ARs obtained in a linked list fashion from d[i]
  - highest AR on stack stored in d[i]
- Consider this program:

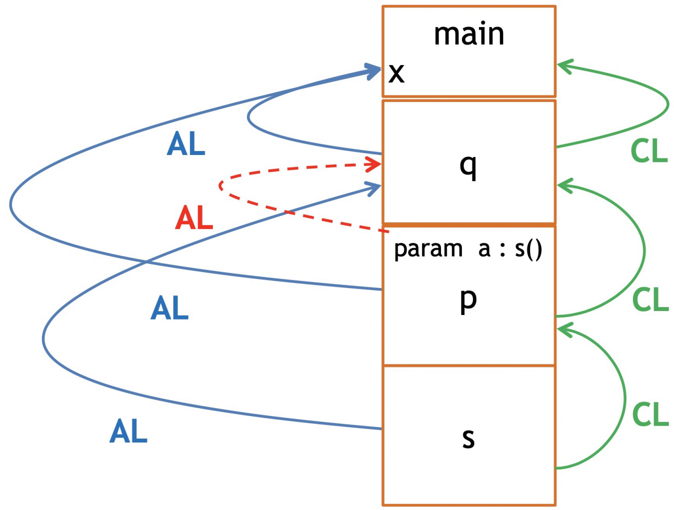
```
main()  //ND(main) = 1
{
   int x;
   p(a) {//ND(p)= 2
       ...
      }
   q() {  //ND(q)= 2
      s() {
          //ND(s)= 3
          ..x..  }
     p(s());
    }

   q();
}
```

s's access link needs to be passed on to p

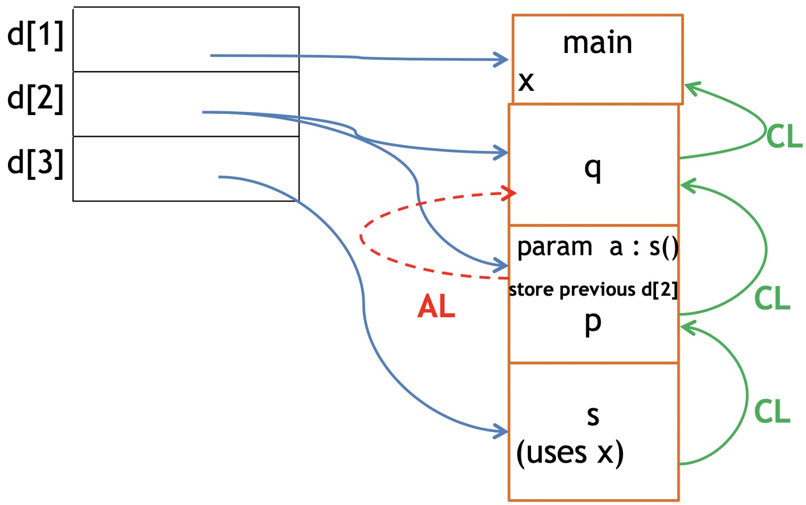
r(p())

caller needs to pass proper access link for that parameter

- Without displays:



- Using displays



CODE GENERATION for PROCEDURES

- Register SP : top of stack

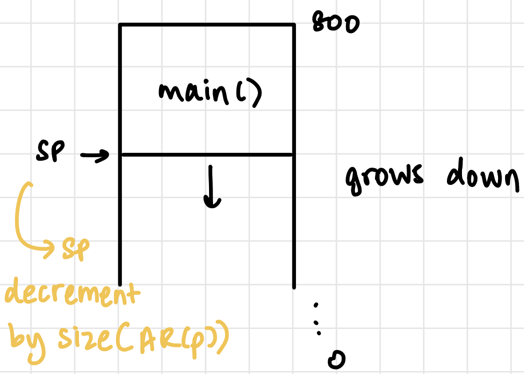
- Steps

1. Initialize SP (stack area)

   MOV SP, #800 ← where stack area starts

2. Push AR for main (done by OS)

3. If main() { ... p(); ... }, allocate AR for p()



   - decrement SP by size of AR(p)

     SUB SP, SP, #30

4. Store return address of p() → next line in main() (calling function)

   - textbook: simplifying assumption that return addr stored at last loc of AR

	Actual params
	Return value
	⋮
SP →	Return addr

ST 0(SP), <return addr>

(0 + SP) is an address

5. Branch to code area of p()

BR <code area>

- Dedicated registers
  - R6: return values
  - R7-R10: passing parameters

Simple Code Generator

- Input: basic block
  Output: target code
- Algorithm: keeps track of values in registers to avoid unnecessary loads and stores

Register & Address Descriptors

- Code generator tracks regs and addresses while generating target code
- Data structures

1. Register Descriptor

- Inform CG about reg availability
- Reg descriptor keeps track of variable names whose values are in the registers (for each register)
- Assume all register descriptors are empty initially
- As code generation progresses, each register holds value of zero or more names

2. Address Descriptor

- For each prog variable, address descriptor keeps track of locations where current value of that var can be found
- Could be reg, mem, stack, combinations
- Info can be stored in symbol table entry

4 Principal Uses of Registers

1. Operands
2. Temps
3. Globals
4. Runtime storage

Function getReg(I)

- I : 3AC ⟶ eg: getReg($x = y + z$)
- Selects registers for x, y, z
- Has access to register descriptors, address descriptors, data flow info (vars live on exit from block)

Special case of Copy: $x = y$

- If y not in reg

  LD R_y, y

- For $x = y$, do not need to store & load y again
- Update RD for R_y to hold x & y
- All stores at end

Q: Consider basic block. Show RD and AD contents while generating code

$$\boxed{\begin{array}{l} t = a-b \\ u = a-c \\ v = t+u \\ a = d \\ d = v+u \end{array}}$$

temp: t, u, v
var: a, b, c, d
regs: $R1, R2, R3$

Instruction	Register Desc			Address Desc						
	R1	R2	R3	a	b	c	d	t	u	v
	—	—	—	a	b	c	d	—	—	—
$t = a - b$										
LD R1, a	a	—	—	a, R1	b	c	d	—	—	—
LD R2, b	a	b	—	a, R1	b, R2	c	d	—	—	—
b no longer req										
SUB R2, R1, R2	a	t	-	a, R1	b	c	d	R2		
$u = a - c$										
LD R3, c	a	t	c	a, R1	b	c, R3	d	R2	—	
SUB R1, R1, R3	u	t	c	a	b	c, R3	d	R2	R1	—

Instruction	Register	Desc		Address	Desc					
	R1	R2	R3	a	b	c	d	t	u	v
v = t + u t no longer req ADD R2, R2, R1	u	v	c	a	b	c, R3	d	–	R1	R2
a = d LD R3, d	u	v	d	a	b	c	d, R3	–	R1	R2
update RD, AD	u	v	d, a	R3	b	c	d, R3	–	R1	R2
d = v + u ADD R1, R2, R1	d	v	a	R3	b	c	R1	–	–	R2
ST a, R3	d	v	a	a, R3	b	c	R1	–	–	R2
ST d, R1	d	v	a	a, R3	b	c	d, R1	–	–	R2