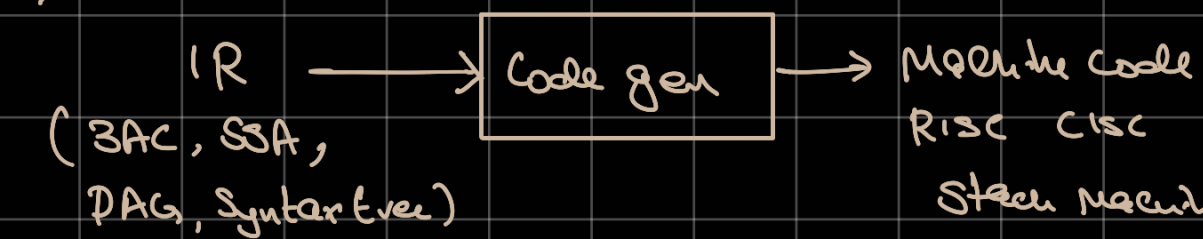
# UNIT-5: Code gen

## Introduction & assembly basics

•>




•> reqs of Code gen:

- Must preserve Source program
- High quality machine code
- Code gen efficiency

•> Tasks:

- Ins selection
- reg alloc & assign
- Ins ordering

(alloc: Set of vars that reside in regs
(assign: We pick the specific reg for a var to store)

•> We assume a RISLV ARM like architecture..

```
LD  dest(reg), Src(mem)
ST  dest(mem), Src(reg)
MOV dest(reg), Src(..)
OP  dest(reg), Src1, Src2      → Addressing modes
BR  label   (uncond jumps)
Bcond R, L
(LTZ, GTZ, EZ, LTEZ, GTEZ)
```

Arithmetic ops to do custom conditions

•) Addressing modes:

var name

i) direct addressing: ADD R2, R1, a

ii) index addressing

```
LD   R1, i
MUL  R1, R1, #4
MOV  R2, R1(a)
ST   x , R2
```

imm addressing

X = a[i]

iii) x = *p

```
LD R1, p
MOV R2, 0(R1)
ST   x, R2
```

iv) imm addressing

•) Code gen examples:

a[i] = y

```
LD   R1, y
LD   R2, i
MUL  R2, R2, #4
MOV  R2(a), R1
```

•) what to do for functions? Activation addr (see ex below)

IC of C():

100 ACTION
120 ACTION
call p 140 (ST 364, 160) 152 (BR 200)
160 ACTION
180 HALT

IC of P():

200 ACTION
return 220 (BR * 364)

Activation record:

| C() | P() |
|---|---|
| 300: | 364: 160 |

ACTIONS: 20 bytes

as always reg addressing = 0 bytes. imms, mem address, operands take 4 bytes (ops too)

1) P()
M = 5
N = m*2
call q
halt

100
400

300
q()
x = 2*x
return

600

activation q() 600: 152

activation P()

P()

| | | |
|---|---|---|
| 100 | LD R1, #5 | 8 |
| 108 | ST M, R1 | 8 |
| 116 | MUL R2, R1, #2 | 8 |
| 124 | ST N, R2 | 8 |
| 132 | ST 600, 152 | 12 |
| 144 | BR 300 | 8 |
| 152 | HALT | |

q()

| | |
|---|---|
| 300 | LD R1, x |
| | MUL R2, R1, #2 |
| | ST x, R2 |
| | BR * 600 |

# Register management

•) Uses:

Register descriptors (available regs) ⇒ allocs

Address " (address of value) ⇒ assignment

•) getReg(I) → free regs, if not: ① ② allocate else

↓ ↳ dead variable, free & allocate else

Spilling: ③ value that is need the furthest away

ORDER IS VERY IMP

a) t = a-b
u = a-c
v = t+u
a = d
a = u+u

You have 3 regs

t, u, v are temps (No need to ST them)

| TAC | ASM | Reg desc. | | | address | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | R1 | R2 | R3 | a | b | c | d | t | u | v |
| t = a-b | LD R1, a | a | | | a, R1 | b | c | d | | | |
| | LD R2, b | a | b | | a, R1 | b, R2 | c | d | | | |
| | MUL R2, R2, R1 | a | t | | a, R1 | b | c | d | R2 | | |
| u = a-c | LD R3, c | a | t | c | a, R1 | b | R3 c | d | R2 | | |
| | SUB R1, R1, R3 | u | t | c | a | b | R3 c | d | R2 | R1 | |
| u = t+u | ADD R3, R1, R2 | u | t | v | a | b | c | d | R2 | R1 | R3 |
| a = d | LD R2, d | u | d, a | v | R2 | b | c | d, R2 | | R1 | R3 |
| d = u+u | SUM R1, R1, R3 | d | a | v | R2 | b | c | R1 | | | R3 |
| exit | ST a, R2 | d | a | u | a, R2 | b | c | a, R1 | | | R3 |
| | ST d, R1 | | | | | | | | | | |

(must store non-temps to mem) a, d

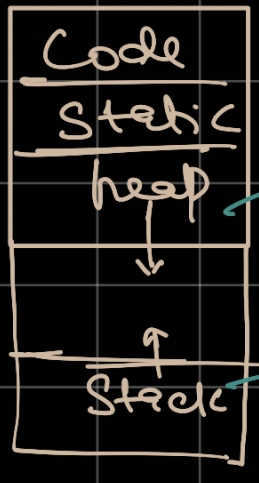
a) $x = y + z$
$z = x * x$
$y = z$
$x = y + z$

R1 & R2 only. no temps

| TAC | ASM | Reg desc. R1 | Reg desc. R2 | address desc. x | address desc. y | address desc. z |
|---|---|---|---|---|---|---|
| $x = y + z$ | LD R1, y | y | | | y, R1 | |
| | LD R2, z | y | z | | y, R1 | z, R2 |
| | ADD R1, R1, R2 | x | z | R1 | y | z, R2 |
| $z = x * x$ | MUL R2, R1, R1 | x | z | R1 | y | R2 |
| $y = z$ | ST y, z | x | z, y | R1 | y, R2 | R2 |
| $x = y + z$ | ADD R1, R2, R2 | x | z, y | R1 | y, R2 | R2 |
| exit | ST x, R1 | x | z, y | x, R1 | y, R2 | R2 |
| | ST z, R2 | x | z, y | x, R1 | y, R2 | z, R2 |

→ Runtime environment

•)




higher mem

•) static alloc: stored in Stack, addressing can be known at compile time

- needs fixed size DS
- No recursive stuff

## Dynamic : heap

- Most Compilers use a Combo of both.

Stack : when functions are called local vars are on Stack. (easy for clean up)

heap : needs GC, used to return Context & such

- Stack is linear heap is hierarchical
- " can never get fragged
- Stack only local access
- stack alloc & de alloc by Compiler on heap IBS the programmer

## → Activation tree & function calls

- Activation tree : reps all function calls in a tree

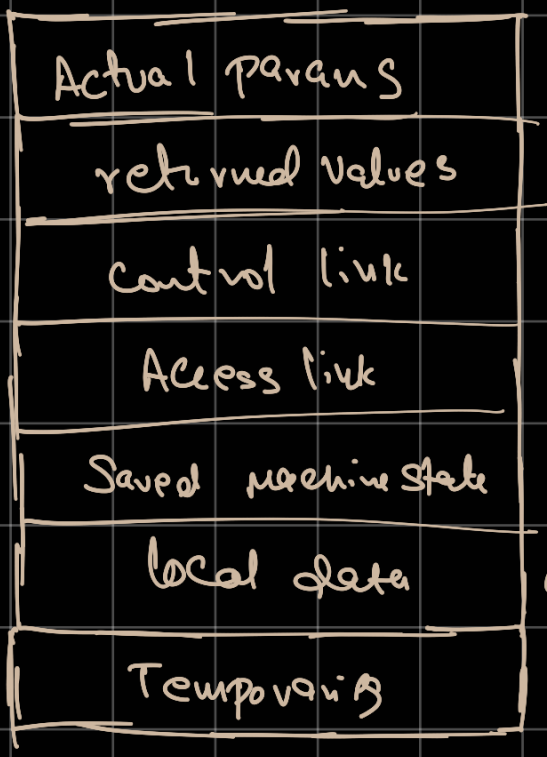
left → right top → down.

- A proc call is considered recursive if a new activation begins before an earlier activation of same function has ended.

Calls : preorder returns : post order

flow of control : depth first traversal.

- Activation record:
  - Stores Context of the calling function
  - Calling func pushes a new activation record to stack
  - returning func pops a stack element




Space to store returned values

ptr to act record of caller (return to caller)

to access data needed by proc but else where

just before call (content of regs)

of called func

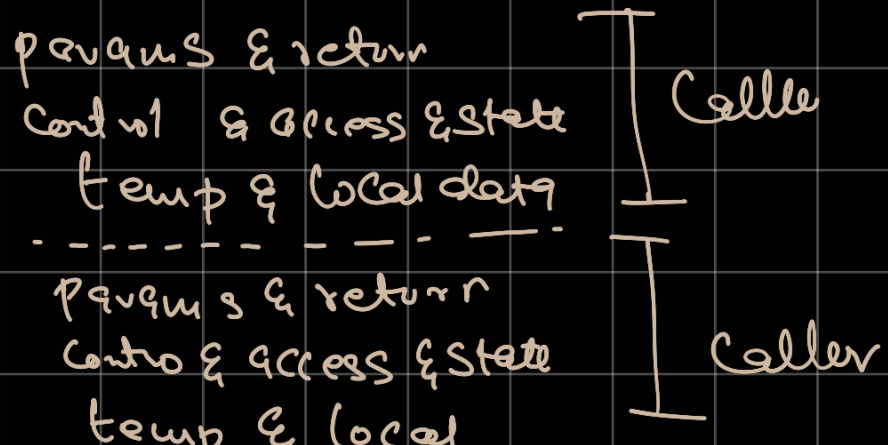
•> Calling sequence is a bunch of code that impl proc calls & handle Activation record.

return sequence: helps restore correct system state on return call.

•> Calling Sequence is divided b/w Callee & Caller

(*) Calling Sequence guidelines

i) Value comm b/w callee & caller are placed at beginning of callee record so they are close to caller.



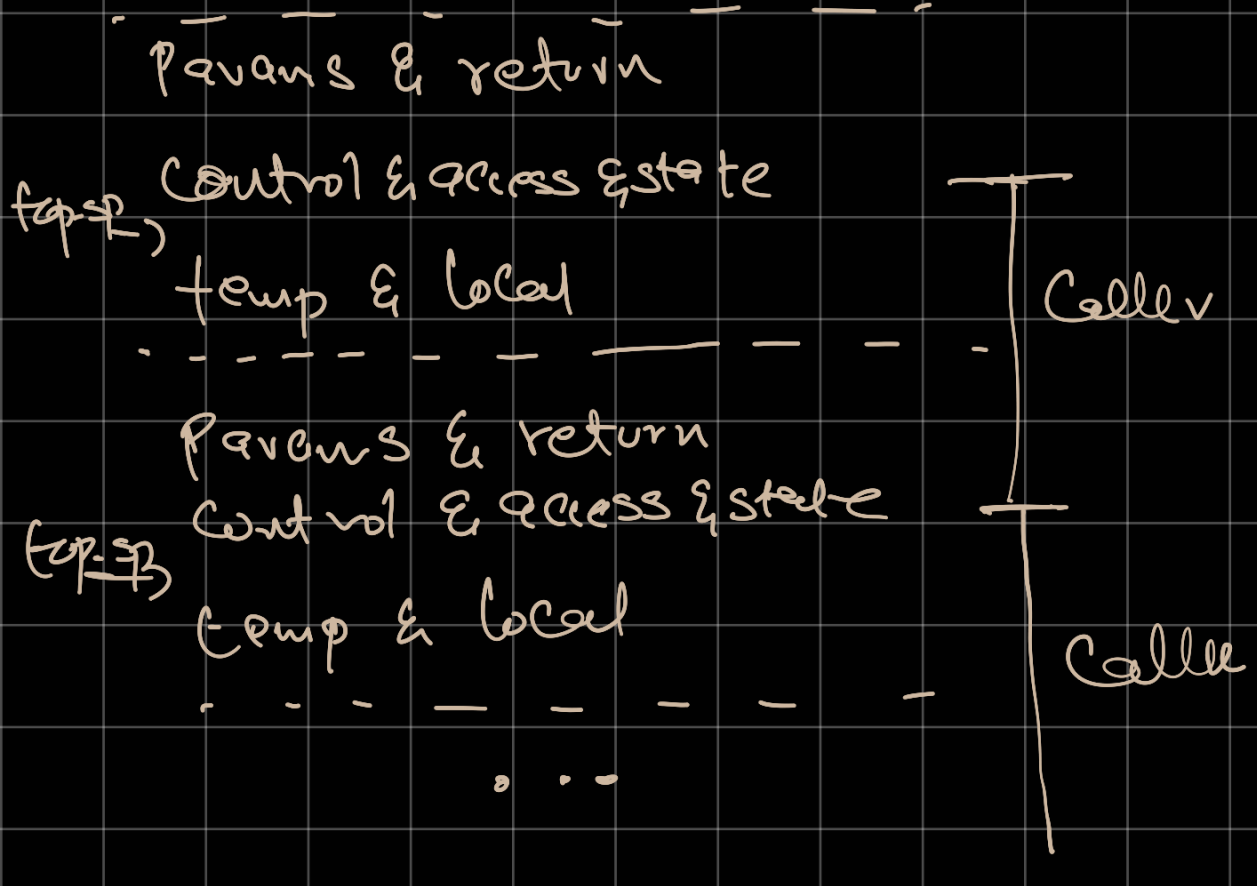
ii) fixed len items are often in middle.
(Control, access, state)

iii) Items whose size may not be know early at the end

iv) top-sp points to end of fixed len data (right before temps & local data). Hence top-sp can access fixed len item using neg offset & var len with pos offset

This is because size of some local vars might depend on parameter given or computed & can need more space at run time.

-> top-sp is known by Caller hence Caller can be made responsible to set Sp (X) responsibility change.



•) responsibility:

Caller Calculate actual params & Store

Caller assign return address

Caller Stores old sp in Control & increment top-sp

Caller Stores access link & State.

Caller Save states & status

Caller Handles local data

•) return Sequence:

i) Callee place return value next to params

ii) Using control link it restores sp & branches

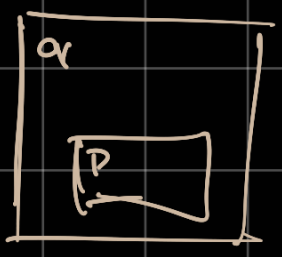
iii) even though sp is decrement Caller knows where ret value is.

→ ~~Var len var in Stack~~ !!??

•) a "top" ptr points to the actual end of stack record (after all var len vars)

•) now the top-sp can access all the data using +ve offset & in case of new call, we know where to start from.

## Non local data

- data used in p but not in p's act record.
- You might think should not all params be passed? think of closure surrounding context sharing in Rust. This gets hard to do.
- 

if p is within q, it's location is dynamic & can't be known before hand. & Move over q or p can be recursive
- 2 strategies for this: Access links & Displays.
- nesting dept. a function starts with nesting depth of
- hence by know nesting dept we know what access link we must be accessing for data (nesting depth-1)
- access links is not always to the calling function. They are to closest nesting function wrt code.
- proc p stack on top & dept of $n_p$ & p needs access to x. defined within proc q that surrounds at depth $n_q$

most usually $n_q < n_p$

follow access link $n_p - n_q$ time

if $n_p == n_{ar}$ (only happens if p given as param to q)

Access link of q & p are same.

hence when they are defined at same level but procedure is passed as param, the caller must control the access link.

actual parameters carry access link with them.

## Displays

- If nesting depth gets large, we'll have to traverse a lot to get data. hence displays are an efficient alternative.
- an auxillary array d called display
- at any point d(i) is a pointer to highest stack at nesting depth i (This is so stupidly SMART!!!)
- hence giving constant lookup time! Unlike cost of maintaining is also constant! Access links;