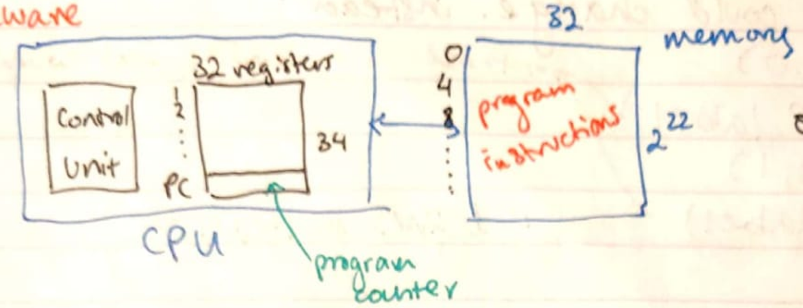
CS241e

Hardware

Sept 11, 2018



State: $\{0,1\}^{32 \cdot 2^{22} + 32 \cdot 34}$

Control Unit: Implements a function from state → state.

$s_i := step(s_{i-1})$.

↳ need a general purpose step fcn.

```
step(state)={
    instruction = state.mem(state.reg(PC)) //fetch instr.
    state2 = state.setReg(PC, state.reg(PC)+4) // PC+=4
    instruction match {....} //decode/encode instr.
}
```

Registers: 0^{th} register is saved for quick access to ∅ value.

Arithmetic: Same answer for add/sub no matter signed/unsigned

↳ not true for multi since you do mod 2^{64} over mod 2^{32}

OpCodes:

Sept 13, 2018

Word that represents a machine lang. instr.

↳ ADD, JR, etc... first abstraction

↳ Assembly is a lang. for writing m. lang programs using opcodes. An assembler translates assembly → machine (also gives you labels)

Example:

Find abs of reg $1.

```
0   SLT(2,1,0)   // is $1 < $0 (always 0)? dest: $2
4   BEQ(2,0,1)   // skip 1 after next. (compare $2 to $0)
8   SUB(1,0,1)   // put $0-$1 in $1
12  JR(31)       // exit
```

$\frac{12-(4+4)}{4} = 1$

Skip number could change. instead:

words: 32 bit instructions (words ⊂ codes)

SLT(2,1,0)
beq(2,0,label)
SUB(1,0,1)
Define(label)   // defines a location
JR(31)

these are codes, not words

How to eliminate Labels?

A symbol table maps label names to meanings. (mem. addys)

pass 1 1) Build symbol table
pass 2 2) Convert uses of each label to corresponding addy/offset
→ Need 2 passes b/c can def. label after usage.

Example: Sept 18, 2018

1) Define(p)
0 lis $1
4 use(e)
8 jr $1
define(e)
12 jr $31

asm →

label	addr
p	0
e	12

use(e) @4
↑ gets changed during relocation too

→ lis $1, 12, jr $1, jr $31

Really, these are 32 bit binary (m. lang)

Relocation and Linking:

Object File: contains machine lang w/ metadata (incl. symbol tb/e)

info on label defs and uses

lis $1
use(p)
jalr $1

calls above example as procedure. Assume assembled separately.

TBC? ↳ asm

l	a

use(p) @4

→ lis $1, 0, jalr $1

Linking:

Process of combining obj. files into one program/lib.

1) Relocation

0	li $1
4	~~0~~ 12
8	jalr $1
12	li $1
16	~~12~~ 24
20	jr $1
24	jr $31

2) Connect labels from dif. obj files. "resolving"

Must adjust label values. "Relocation"

Variables:

Abstraction of storage location that can hold a value.

Extents:

Extent of a var. instance is time interval where var accessible

	var kind	extent
fixed location →	global	entire execution
stack →	proc-local	one exec. of proc.
heap →	field of object	obj creation → last use

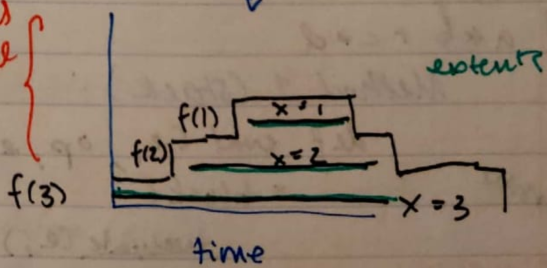
ex: fact(3)

= fact(2) * 3

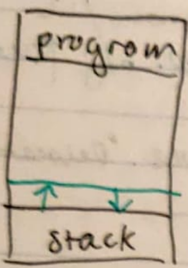
= fact(1) * 2 * 3

= 1 * 2 * 3

many instances of the same variable.



Stack Implementation:



Designate reg $30 to store addr of top of stack. "Stack ptr"

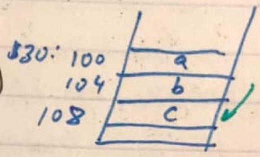
push: -= 4   pop: += 4

Can read vals not on top with lw (with offset).

LW(Reg(1), 8, Reg(30))

// reads "c"

use 29 instead



offset (8) is stored for each variable.

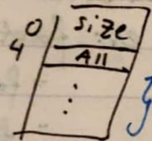
Frame Pointers:

Copy of stack ptr made at beginning of procedure (doesn't chng)

$29

Memory:

Chunks.



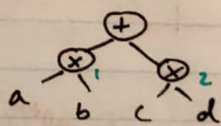
block of consec. mem locations.

offsets indexed by vars.

Example:

1) a*b + c*d

Method 1 (stack):



recursive

✓ general, easy

x inefficient

```
def eval(e1, op, e2) {
    block(
        evaluate(e1)
        push $3
        evaluate(e2)
        pop $4
        $3 = $4 op $3
    )
}
```

$*_1$ { read a into $3
push $3
read b into $3
pop $4
$3 = $4 * $3 }

push $3

$*_2$ { same as $*_1$.

pop $4

$3 = $4 + $3

Scanned by CamScanner

Method 2: Variables

$t_1 = a \cdot b \quad t_2 = c \cdot d \quad t_3 = t_1 + t_2$

✓ easy to improve

× need reg's for ops

× many vars

```
def evaluate(e1, op, e2): (Code, Variable) = {
    (c1, v1) = evaluate(e1)
    (c2, v2) = evaluate(e2)
    v3 = new Variable("temp")
    code = c1 ++ c2 :+ (v3 = v1 op v2)
    (code, v3)
}
```

need "register allocation"

Sept 25, 2018

Method 3: Hybrid (vars, operations on registers)

```
def evaluate(e1, op, e2): Code {
    t1 = new Variable("temp")
  → Scope(t,
    block(
        evaluate(e1)
        write(t1, $3)
        evaluate(e2)
        read($4, t1)
        $3 = $4 op $3
    )
  →)
```

Scope: type of Code to keep track of used vars

If statements:

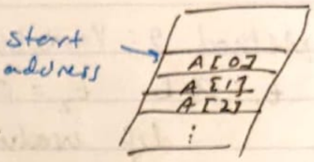
if (e1 op e2) T else E

```
evaluate(e1)
write(t1, $3)
evaluate(e2)
read($4, t1)
bne($3, $4, else)
beq($0, $0, end)
define("else")
E
define("end")
```

can reuse evaluate ==

*try:
while loop
while (e1 op e2) body

Arrays:



- Contiguous area of memory
- Compute offset from start address
- deref(address): put val at addr in \$3. *(address)
  - ↳ LW for assignment.
- assignToAddr(address, code) *address = code.
  - ↳ SW

Register Allocation: Sept 27, 2018

- Exam, but not assignments.
- Assigns registers/stack locations to variables.
- a+b+c+d+e:

$t_2 = 42$

*1	$t_1 = a+b$	$r_1 = a+b$
*2	$t_2 = t_1+c$	$r_1 = r_1+c$
*3	$t_3 = t_2+d$	$r_1 = r_1+d$
*4	$t_4 = t_3+e$	$r_1 = r_1+e$

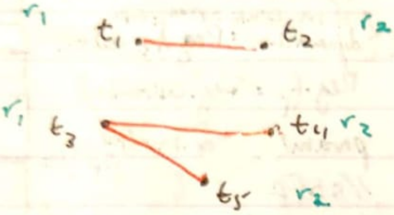
- A variable is "live" at program point p if the value at p may be read sometime after p. (watch for overwrite b4 read)
  - ↳ at *1, t_1 is live, t_2 is not (overwrite)
  - ↳ at *2, t_2 is live, t_1 isn't anymore (no more reads)
  - ↳ at *4, t_4 is not live.
- 2 vars can share a reg if they're never both live at once.

Example:

1) $t_1 = a*b$ — t_1 live
$t_2 = c*d$ — t_1, t_2 live
$t_3 = t_1+t_2$ — t_3 live
$t_4 = e*f$ — t_3, t_4 live
$t_5 = t_3-t_4$ — t_5, t_3 live
$g = t_3+t_5$

r_1 for t_1, t_3
r_2 for t_2, t_4, t_5

- Interference Graph:
  - Variables are vertices.
  - Edges iff vars are live at once.



- Graph Colouring:
  - Assigns colour to each vertex where each edge connects dif. colours. → Finding a minimal colouring for an arb. graph is NP hard. :(

→ Greedy algos work ok.

for each vertex v {
    colour v with the least colour not yet used
} by it's neighbours

Valid, but not nec. minimal.

Procedures:

- Reusable seq [code].
- Calling code + procedure must agree on conventions.
  - where in memory/registers to pass arguments + return val.
  - who allocates space on stack?
  - Which registers the procedure may modify "callee-save" - modify "caller-save" - save

Calling Code   Procedure.   Oct 2, 2018

Define(proc).

Next page. →

Conventions:

- callee save preserved register - SP, FP, frees param chunk, alloc/frees frame.
- caller save modified registers - PC with JALR, allocates param chunk
  - → passes param addr in Reg.allocated.
  - → Reg.allocated.
  - → Return val in \$3
  - → All other registers

Calling Code

before call: evaluate(args) into temp vars

↳ Stack.allocate(parameters)

Need Reg.allocated Not FP: param 1 = arg 1 (temp var) ⋮ param n = arg n (temp var)

LIS(Reg.targetPC)

Use(Proc)

JALR(Reg.TargetPC)

Procedure:

Define(proc)

Reg.savedParamPtr = allocated

Stack.allocate(frame)

Need Reg.Alloc'd: dyn.Link = Reg.FP, SavedPC = Reg.savedPC

Reg.fp = Reg.allocated

paramPtr = Reg.savedPP

//stuff

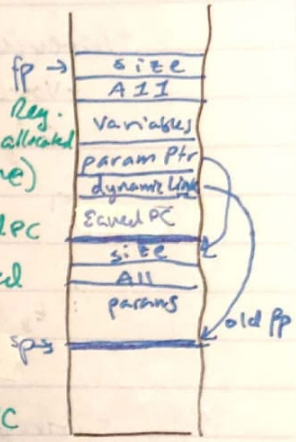
Reg.savedPC = savedPC

Reg.fp = dynamic link

Stack.pop() //frame

Stack.pop() //params

JR(Reg.savedPC)



eliminate Var Acc A5:

if v is Var:
  access v in frame

else: //param
  read paramPtr from frame to Reg.scratch
  access v in param chunk ← base reg = Reg.scratch

Oct 4, 2018

Dynamic Links:
Points to frame of caller.

Prologue/Epilogue:
Instrs at beg/end of proc that sets up/clears frame + reg's.

Static Link:
Points to frame that function is defined in

Nesting Depth:
depth(top) = 0. if p is nested in p', depth(p) = depth(p') + 1
depth(static link) = depth(current) - 1

Scanned by CamScanner

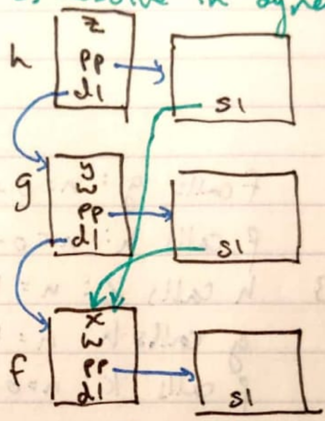
Nested Procedures:

'nesting depth'

```
0 def f() = {
    val x=2
    val w=5
1   def g() = {
        val y=3
        val w=7
        x+y+h()
    }
1   def h() = {
        val z=4
        z+w
    }
    g()
}
```

1) h uses f()'s w: 5. Static scoping
2) h uses g()'s w: 7. Dynamic scoping.

1) Resolve in statically enclosing procs
2) Resolve in dynamically calling procs.



To access variable v:

- n = depth(current) - depth(proc w/ v)
- Follow sl n times to get v.

f calls g:
g's sl: f's pp

check depth. ← g calls h:
if ==, take → h's sl: g's sl

While loop → nested procedure

```
def m() = {
    var i=0
    var j=0
    def loop() = {
    if ~~while~~ (i<10) {
        i=i+1
        j=j+i
        loop()
    }
    }
    loop()
    i+j
}
```

To compute SL:

- d(SL) = d(current) - 1
- n = depth(caller) - depth(s.l)
  = depth(caller) - depth(curr) + 1

If n=0, target s.l. = caller's pp
else, target s.l. = follow caller's s.l. n times

Oct 11, 2018

Example:

1) f { 0

g { } 1

h { 1

k { } 2

}

}

f ( ) {

g ( ) { }

h ( ) { S.L.

k ( ) { }

}

}

f calls g : n = 0 - 1 + 1 = 0

f calls h : n = 0 - 1 + 1 = 0

h calls k : n = 1 - 2 + 1 = 0

} callee callers S.L. = FP

g calls h : n = 1 - 1 + 1 = 1 h's SL = g's SL

f calls k : n = 0 - 2 + 1 = -1 ?? F can't call K

Anon. Functions:

No name (lambdas)

increase = { x => x + 2 }

Free variables:

Undefined / not bound variable in an expression.

in { x => x + increment } :

↳ free

Closed Expressions:

No free vars.

def increaseBy(inc : Int) {

def proc(x) = {x => x + inc} ← in LACS, this would be defined.

} Proc closure

Call(proc)

increase = increaseBy(8)

increase(5) // 8 + 5 = 13

↑ call Closure (closure : Code, ...)

computes closure