Aho, Sethi, and Ullman Machine Model

For code generation, Aho, Sethi, and Ullman propose a simple machine model.

- byte-addressable machine with four byte words
- n general purpose registers
- two-address instructions – op, src, dest
- all instructions have latency = 1

Mode	Form	Address	Address cost
absolute	M	M	1
register	R	R	0
indexed	off(R)	off + c(R)	1
ind. register	*R	c(R)	0
ind. indexed	*off(R)	c(off + c(R))	1

In comparison, we'll assume all operands are registers

- three-address instructions – op dest, src1, src2

Code generation for trees

Sethi-Ullman algorithm for

- generating code for expression trees
- while “minimizing” number of registers used

Overview of algorithm

Phase1

- compute number of registers required to evaluate a subtree without storing values to memory
- label each interior node with that number

Phase2

- walk the tree and generate code
- evaluation order guided by labels

Phase 1

if n is a leaf then

label(n) $\leftarrow$ 1

else begin /* n is an interior node */

let $n_1, n_2, \ldots, n_k$ be the children of n,

ordered so that

$label(n_1) \geq label(n_2) \geq \ldots \geq label(n_k)$

$label(n) \leftarrow \max_{1 \leq i \leq k}(label(n_i) + i - 1)$

Can compute labels in postorder

For $n \leq 2$, label is defined recursively as:

$$label(n) = \begin{cases} l_1 + 1 & \text{if } l_1 = l_2 \\ \max(l_1, l_2) & \text{if } l_1 \neq l_2 \end{cases}$$

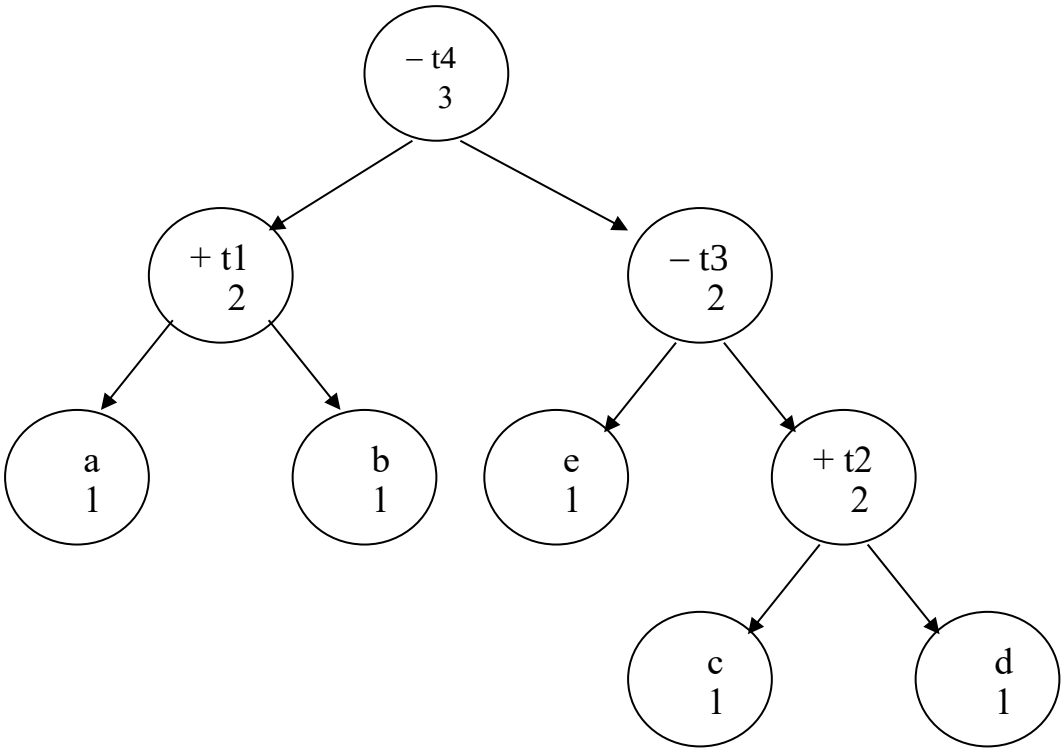
label = minReg(minimum # of registers)

Phase 2

REG = current register number(initialized to 1)

```
procedure gencode(n)
    if n is leaf "name"
        /* case 0 – just load it */
        emit(load, REG, name)
    else if n is interior node "op n1, n2" then
        if label(n1) ≥ label(n2) then
            /* case 1 – generate left child first */
            gencode(n1)
            REG = REG + 1
            gencode(n2)
            REG = REG – 1
            emit(op, REG, REG, REG + 1)
        else label(n1) < label(n2) then
            /* case 2 – generate right child first */
            gencode(n2)
            REG = REG + 1
            gencode(n1)
            REG = REG – 1
            emit(op, REG, REG + 1, REG)
        endif
    endif
```

Example




```
gencode(t4)                                  case 1
        gencode(t1)                          case 1
                gencode(a)                   case 0
                        load r1, a
                gencode(b)                   case 0
                        load r2, b
                add r1, r1, r2
        gencode(t3)                          case 2
                gencode(t2)                  case 1
                        gencode(c)           case 0
                                load r2, c
                        gencode(d)           case 0
                                load r3, d
                        add r2, r2, r3
                gencode(e)                   case 0
                        load r3, e
                sub r2, r3, r2
        sub r1, r1, r2
```

Extensions to the labeling scheme

Multiple register operations

- increase base case to reserve registers
- paired registers may require triples

Algebraic properties

- commutativity, associativity to lower labels
- deep, narrow, left-biased trees

Common subexpressions(dags)

- increases complexity of code generation(NP-complete)
- partition into subtrees that have cses as roots
- order trees and apply Sethi-Ullman

More “optimal code generation”

Sethi-Ulman is optimal on simple machine model

- minimizes register use
- minimizes execution time

What about a more realistic machine model?

Delayed-load architectures are more complex

- issue load, result appears delay cycles later
- execution contains unless result is referenced
- premature reference causes hardware to stall(interlock)

Many microprocessor-based systems have this property

So what's wrong with Sethi-Ullman

Assumptions

1. delayed-load, RISC architecture
   - register-to-register ops, load, &store
   - 1 cycle/instruction
   - non-blocking, 2 cycle load

The problem

- loads will interlock if delay > 0

Overview of the solution

- move loads back at least delay slots from ops
- this increases register pressure
- want to minimize this extra register pressure

Brute force solution

Obvious approach

- issue all the loads
- execute all the operators

Unfortunately, this can create too much register pressure

Phase ordering

- allocate first => poor schedule
- schedule first => poor allocation

Scheduling & allocation are like oil and water

Proebsting and Fischer

- retain good properties of Sethi-Ullman
  - contiguous evaluation
  - minimal register use
- consider two problems together

a recurrent theme for the nineties

The DLS(Delayed-Load Scheduling) algorithm

The big picture

- schedule the operation
- schedule the loads

Legal ordering

- children of an operator appear before it
- each load appears before operator that uses it

The final schedule

- preserve relative order of operations
  (ops $\leftrightarrow$ ops)
- preserve relative order of loads
  (loads $\leftrightarrow$ loads)
- changes relative order of loads to operations

The DLS algorithm

The canonical order

Given R registers

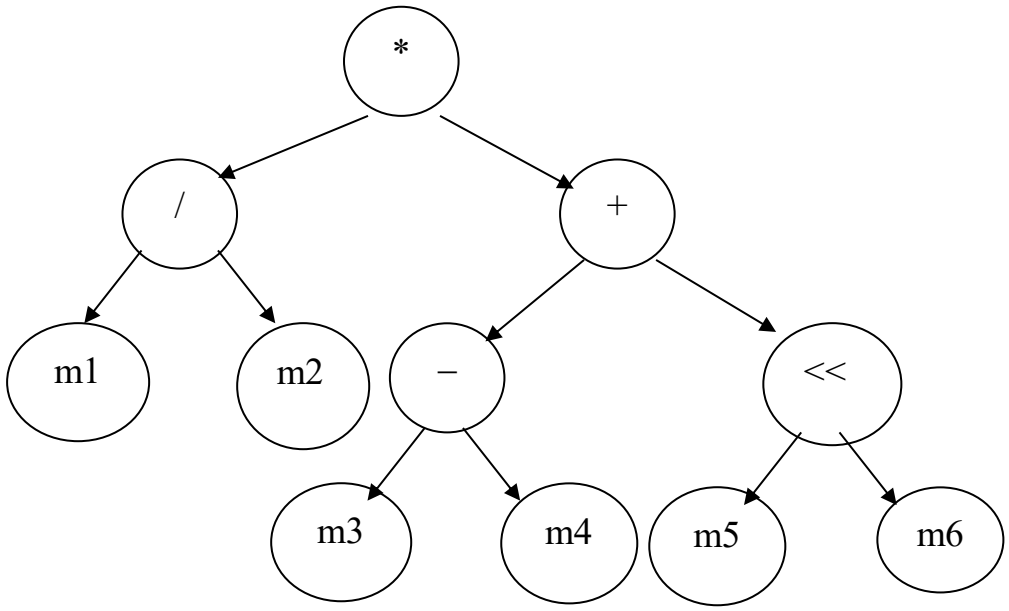
1. schedule R loads
2. schedule a series of (op, load) pairs
3. schedule the remaining R – 1 ops

This keeps extra register pressure down

The algorithm

1. run Sethi-Ullman algorithm
   - calculate minReg for each subtree
   - create an ordering of the operator
2. put loads into canonical order
   - uses minReg + 1 regs
   - requires some renaming

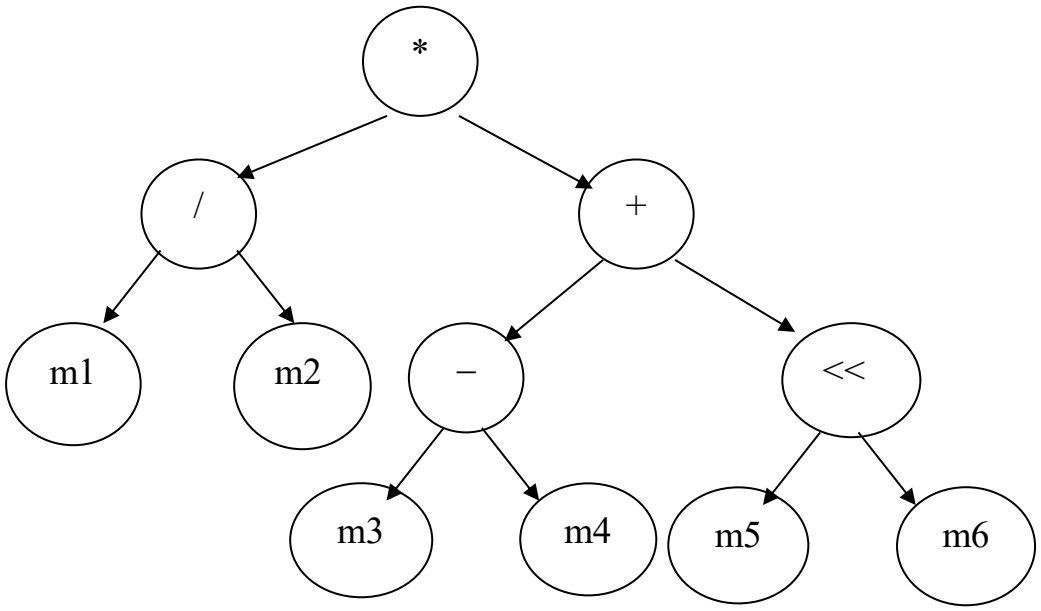
Example



Canonical ordering

Operators	Loads
1. sub	1. load m3
2. shift	2. load m4
3. add	3. load m5
4. div	4. load m6
5. mult	5. load m1
	6. load m2

Example



	Sethi-Ullman	DLS(3)	DLS(4)
1.	load m3, r1	load m3, r1	load m3, r1
2.	load m4, r2	load m4, r2	load m4, r2
3.	–stall–	load m5, r3	load m5, r3
4.	sub r1, r1, r2	sub r1, r1, r2	load m6, r4
5.	load m5, r2	load m6, r2	sub r1, r1, r2
6.	load m6, r3	–stall–	load m1, r2
7.	–stall–	shift r2, r3, r2	shift r3, r3, r4
8.	shift r2, r2, r3	load m1, r3	load m2, r4
9.	add r1, r1, r2	add r1, r1, r2	add r1, r1, r3
10.	load m1, r2	load m2, r2	div r2, r2, r4
11.	load m2, r3	–stall–	mult r1, r2, r1
12.	–stall–	div r2, r3, r2	
13.	div r2, r2, r3	mult r1, r2, r1	
14.	mult r1, r2, r1		

Limitations

Input

- handles trees, not dags
- limited to a single basic block
- values not kept in registers

Output

- delay > 1 => optimality not guaranteed
- non-constant delay causes deeper problems

Strengths

- fast, simple algorithm
- clever metric for spilling
- no excuse to do worse