

Compilers - Chapter 5 part 1:

Register allocation using fewer registers

- Lecturers:
 - Paul Kelly (p.kelly@imperial.ac.uk)
 - Naranker Dulay (n.dulay@imperial.ac.uk)
- Materials:
 - materials.doc.ic.ac.uk, Panopto
 - Textbook
 - Course web pages
(<http://www.doc.ic.ac.uk/~phjk/Compilers>)
 - Piazza
(<https://piazza.com/class/kf7uelkyxk7aa>)

Overview

- We have seen a simple code generation algorithm for arithmetic expressions, uses registers when it can and stack otherwise
- We now consider an algorithm which minimises the number of registers needed—and therefore avoids “spilling” intermediate values into main store whenever possible
- Effective use of registers is vital since access to registers is much faster than access to main memory. E.g. because
 - Registers need expensive, fast circuitry, OK in small quantities if used well
 - Registers are multi-ported: two or more registers can be read in the same clock cycle
 - Registers are specified by a small field in the instruction
- Developments in compiler technology have encouraged hardware designers to provide larger register sets.

The plan

- A simple language with assignments, loops etc.
- A stack-based instruction set and its code generator
- Code generation for a machine with registers:
 - an unbounded number of registers
 - a fixed number of registers
 - **avoiding running out of registers**
 - register allocation across multiple statements

IDEA: Order *does* matter

- **Example:** $x + (3 + (y * 2))$
- Using straightforward code generator yields:
 - LoadAbs R0 "x",
 - LoadImm R1 3,
 - LoadAbs R2 "y",
 - LoadImm R3 2,
 - Mul R2 R3, ($R2 := y * 2$)
 - Add R1 R2, ($R1 := 3 + (y * 2)$)
 - Add R0 R1
- Modify the expression to $((y * 2) + 3) + x$:
 - LoadAbs R0 "y",
 - LoadImm R1 2,
 - Mul R0 R1, ($R0 := y * 2$)
 - LoadImm R1 3,
 - Add R0 R1, ($R0 := (y * 2) + 3$)
 - LoadAbs R1 "x",
 - Add R0 R1

This uses **two** registers instead of **four**.
Why?

Subexpression ordering principle:

- *Given an expression $e_1 \text{ op } e_2$, always choose to evaluate **first** the expression which will require **more** registers.*
- **Reason?**
 - During evaluation of the second subexpression, one register will be used up holding the results of evaluating the first expression.
 - So there is one more register free during evaluation of the first subexpression.

How can we put this principle to work?

- Consider binary operator application $e_1 \text{ op } e_2$:
- **Suppose:** e_1 requires L registers
 e_2 requires R registers
- If e_1 is evaluated first, we will need
 L registers to evaluate e_1
then R registers to evaluate e_2
PLUS ONE to hold the result of e_1
- If e_1 is evaluated first, the maximum number of registers in use at once is $\max(L, R+1)$
- If e_2 is evaluated first, the maximum number of registers in use at once is $\max(L+1, R)$

We choose the order which yields the smaller value

- Suppose we had a function `weight` which calculates how many registers will be needed to evaluate each subexpression. We could use this to decide which subexpression to evaluate first

`weight :: Exp -> Int`

`weight (Const n) = 1` *(assuming have to load constant into register)*

`weight (Ident x) = 1` *(assuming have to load variable into register)*

`weight (Binop op e1 e2)`

`= min [cost1 , cost2]`

where

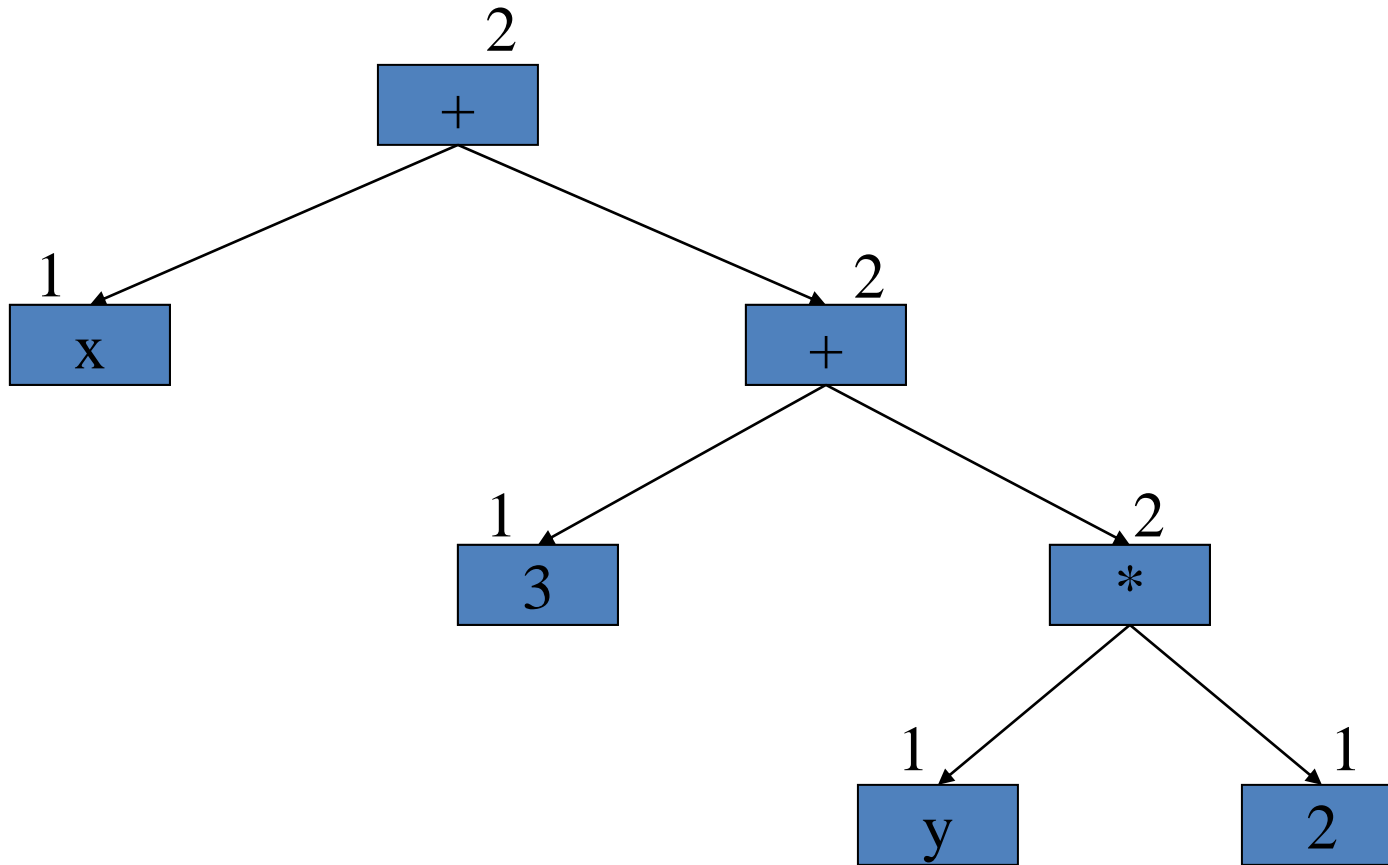
`cost1` *(if we choose to do e1 first)*

`= max [weight e1, (weight e2)+1]`

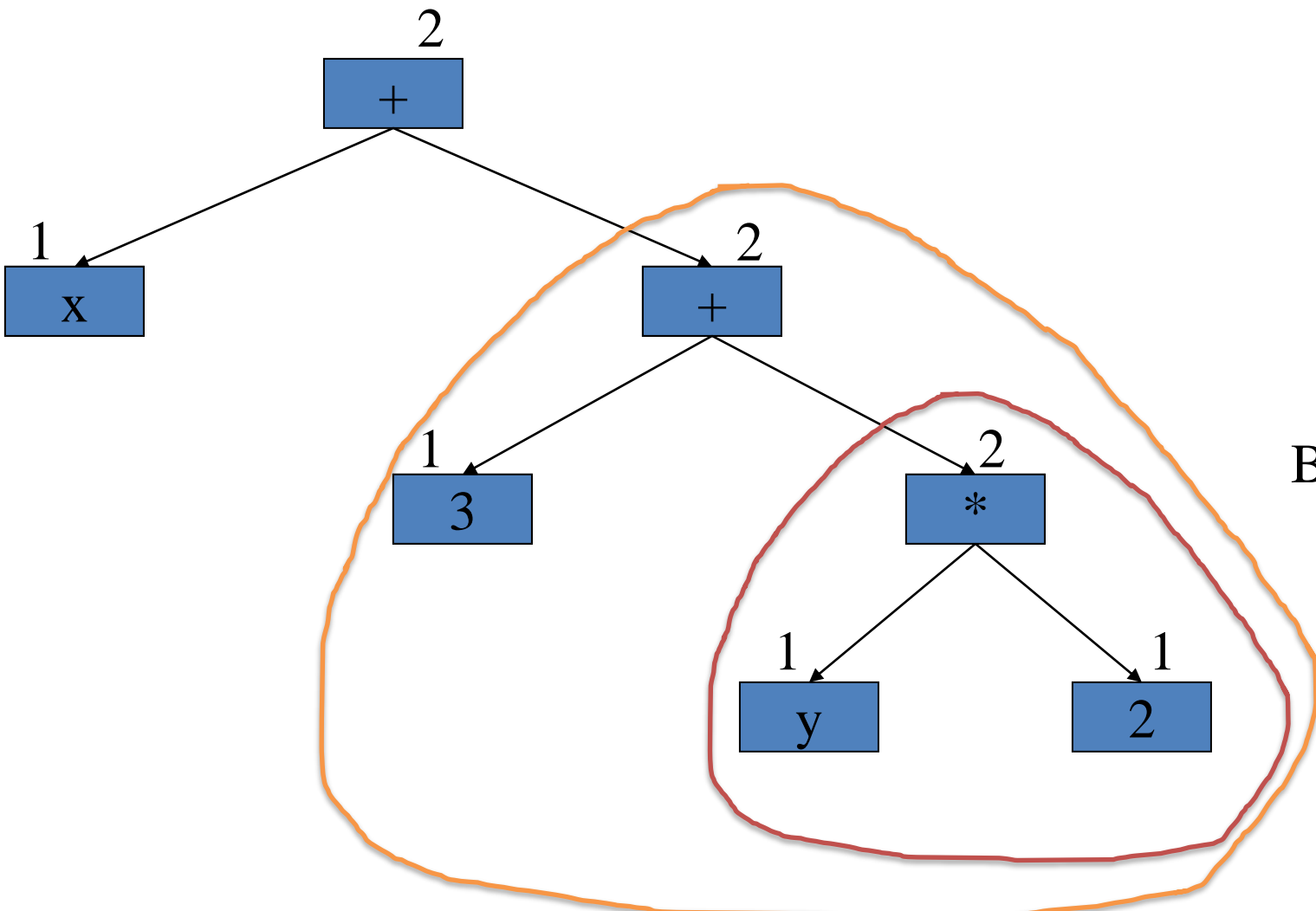
`cost2` *(if we choose to do e2 first)*

`= max [(weight e1)+1, weight e2]`

Consider example earlier: $x + (3 + (y * 2))$



Consider example earlier: $x + (3 + (y * 2))$



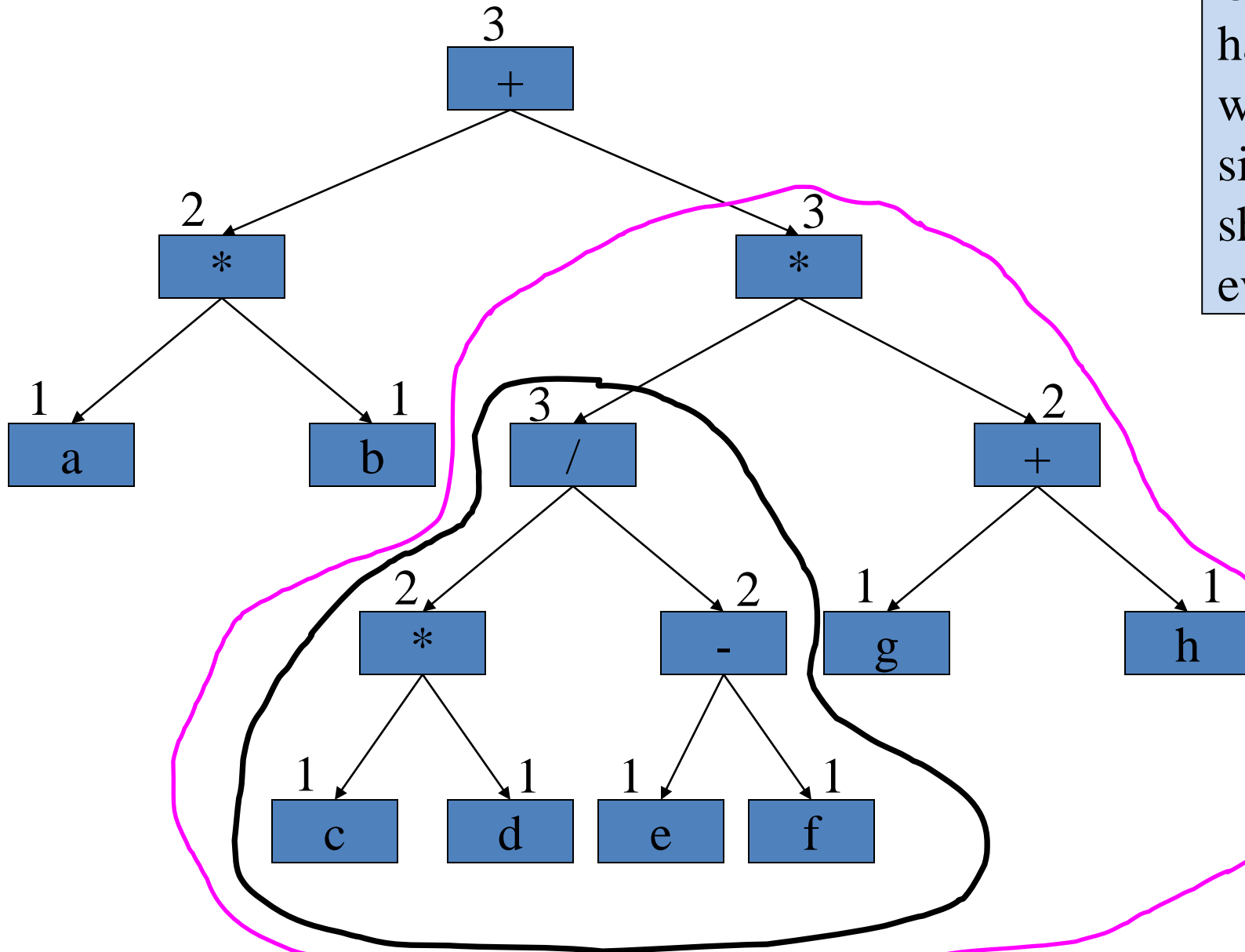
Bad ordering:

LoadAbs R0 "x",
LoadImm R1 3,
LoadAbs R2 "y",
LoadImm R3 2,
Mul R2 R3,
Add R1 R2,
Add R0 R1

Better ordering:

LoadAbs R0 "y",
LoadImm R1 2,
Mul R0 R1,
LoadImm R1 3,
Add R0 R1,
LoadAbs R1 "x",
Add R0 R1

More complicated example: $((a*b)+(((c*d)/(e-f))*(g+h)))$



Circled subtree has higher weight than its sibling, so should be evaluated first.

When subtrees have equal weight, neither order is better – it doesn't matter

Modifications to the code generator

- Basically, what we'd like to do is choose the subtree evaluation order depending on their weights. We can do this with commutative operators as follows:

```
transExp (Binop op e1 e2) r
  = if weight e1 > weight e2
    then
      transExp e1 r ++
      transExp e2 (r+1) ++
      transBinop op r (r+1),
    else
      transExp e2 r ++
      transExp e1 (r+1) ++
      transBinop op r (r+1)
```

What might go wrong?

Modifications to the code generator

- Basically, what we'd like to do is choose the subtree evaluation order depending on their weights. We can do this with commutative operators as follows:


```
transExp (Binop op e1 e2) r
  = if weight e1 > weight e2
    then
      transExp e1 r ++
      transExp e2 (r+1) ++
      transBinop op r (r+1),
    else
      transExp e2 r ++
      transExp e1 (r+1) ++
      transBinop op r (r+1)
```

Unfortunately
this is OK only
if the operator is
commutative.

- A tempting solution is to leave the result in the right register in the first place – something like:

```
transExp (Binop op e1 e2) r
= if weight e1 > weight e2
  then
    transExp e1 r ++
    transExp e2 (r+1) ++
    transBinop op r (r+1),
  else
    transExp e2 (r+1) ++
    transExp e1 r ++
    transBinop op r (r+1)
```

The problem with this is that the code generated by `transExp e1 r` might clobber the value in register `r+1`



Register Targeting

- **Problem:** we want to be able to tell `transExp` to leave the result in register `r`, but that it cannot use register `r+1`.
- **Idea:** give `transExp` a list of the registers it is allowed to use...

Register targeting - implementation

$\text{transExp} :: \text{Exp} \rightarrow [\text{Register}] \rightarrow [\text{Instruction}]$

- The translator transExp is *given* a list of the registers it is allowed to use. It should leave the result in the first register in the list.
- The base cases:

$\text{transExp} (\text{Const } n) (\text{destreg}:\text{restofregs})$
 $= [\text{LoadImm destreg } n]$

$\text{transExp} (\text{Ident } x) (\text{destreg}:\text{restofregs})$
 $= [\text{LoadAbs destreg } x]$

Register targeting – implementation...

- The interesting case is the binary operator:

transExp (Binop op e1 e2) (dstreg:nxtreg:regs)

= if weight e1 > weight e2 then

transExp e1 (dstreg:nxtreg:regs) ++

transExp e2 (nxtreg:regs) ++

transBinop op dstreg nxtreg

else

transExp e2 (nxtreg:dstreg:regs) ++

transExp e1 (dstreg:regs) ++

transBinop op dstreg nxtreg

e1 can use all regs
e2 can use all but one
e1 & e2 still delivered
to right registers

e2 can use all regs
e1 can use all but one
e1 & e2 still delivered
to right registers

Register targeting – implementation...

- The interesting case is the binary operator:

transExp (Binop op e1 e2) (dstreg:nxtreg:regs)

= if weight e1 > weight e2 then

transExp e1 (dstreg:nxtreg:regs) ++

transExp e2 (nxtreg:regs) ++

transBinop op dstreg nxtreg

else

transExp e2 (nxtreg:dstreg:regs) ++

transExp e1 (dstreg:regs) ++

transBinop op dstreg nxtreg

The arithmetic instruction ends up the same either way – because the operands are in the same place, whichever order we chose

Embellishment: immediate operands

- As we saw before, it is important to use immediate addressing modes wherever possible, i.e.

LoadImm R1 100, (eg. `movl $100,%ebx`)

Mul R0 R1 (`imull %ebx,%eax`)

- Can be improved using immediate addressing:

MulImm R0 100 (eg. `imull $100,%eax`)

- The translator can use pattern-matching to catch opportunities to do this
- **The weight function must be modified so that it correctly predicts how many registers will be used**

How good is this scheme?

- Simple example: “result = a/(x+1)”:

```
movl _x,d0
addq #1,d0
movl _a,d1
divsl d0,d1
movl d1,_result
```

- This is the code produced by Sun’s compiler for the 68000 processor, as found in the Space Shuttle main engine controller; also widely used eg in Sony Clie PDAs

(I chose to illustrate this using the 68000 instruction set because the Intel IA-32 instruction set has constraints (eg very few registers) which make examples unhelpfully complicated)

A more complicated example:

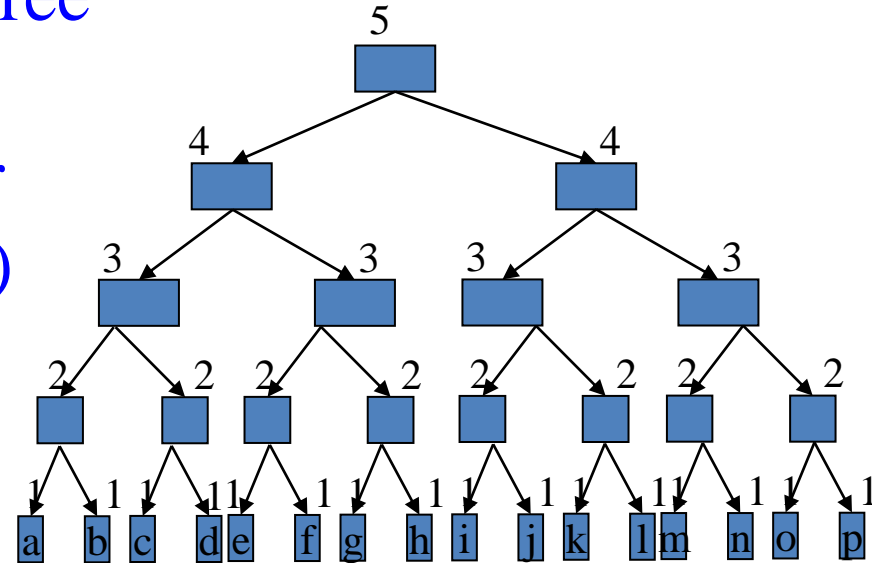
- $\text{result} = ((a/1000)+(1100/b))*((d/1001)+(1010/e));$

```
movl a,d0
divsl #1000,d0
movl #1100,d1
divsl b,d1
addl d1,d0
movl d,d1
divsl #1001,d1
movl #1010,d2
divsl e,d2
addl d2,d1
mulsl d1,d0
movl d0, result
```

- This is what your code generator would produce, if you follow the design presented in these lectures

Effectiveness of Sethi-Ullman numbering

- Identify worst case:
 - Perfectly-balanced expression tree (since an unbalanced tree can always be evaluated in an order which reduces register demand)
 - k values
 - $k/2-1$ operators
 - $\lceil \log_2 k \rceil$ registers required



- So the expression size accommodated using a block of N registers is proportional to 2^N

Register Allocation: Summary

- Sethi-Ullman numbering minimises register usage in arithmetic expression evaluation
- It works by choosing subexpression evaluation order: do register-hungry subexpressions first because later registers will be occupied by the results of earlier evaluations
- **Sethi-Ullman numbering is optimal** in a very restricted sense: it fails to handle reused variables and it fails to put user variables in registers
- However it is fast, reliable and reasonably effective (e.g. was used in C compilers for many years)
- Optimising compilers commonly use more sophisticated techniques, for example based on graph colouring

Feeding curiosity...

- The Sethi-Ullman-Ershov “weights” algorithm finds a schedule that uses the minimum number of registers – for expression *trees* only. When sub-expressions are shared (so we have a DAG), the problem is NP-complete. You can get much better performance by re-ordering your code before you give it to your compiler – but you need a good heuristic. See “**Register optimizations for stencils on GPUs**”, Prashant Rawat et al, PPOPP 2018.
- Registers are not the only way to achieve better efficiency than direct execution of stack code. For example, instructions in the “Mill” architecture specifies each operand using the relative offset of the recent instruction that generated it. Results are placed on a fixed-size conveyor “belt”. See <https://millcomputing.com/docs/belt/>. In the WaveScalar design it’s the other way round: instructions send their results to a specified destination instruction –see <http://wavescalar.cs.washington.edu/wavescalar.pdf>