Compilers I - Chapter 5: Register Allocation

- Lecturers:
  - Paul Kelly (phjk@doc.ic.ac.uk)
    - Office: room 304, William Penney Building
  - Naranker Dulay (nd@doc.ic.ac.uk)
    - Office: room 562
- Materials:
  - Textbook
  - Course web pages (http://www.doc.ic.ac.uk/~phjk/Compilers)
  - Piazza (http://piazza.com/imperial.ac.uk/fall2016/221)

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 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
- Conditionals and Boolean expressions

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 (3+(y*2))+x:

```
LoadImm R0 3,
LoadAbs R1 "y",
LoadImm R2 2,
Mul R1 R2, (R1 := y*2)
Add R0 R1, (R0 := 3+(y*2))
LoadAbs R1 "x",
Add R0 R1
```

This uses three registers instead of four. **Why?**
Is it possible to do better with this example?

It matters what order the subexpressions are evaluated in…

- Modify expression further to get ((y*2)+3)+x.
- We still get the same answer and the code uses even fewer registers:

```
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
```

- What general principle reduces the number of registers needed?

Subexpression ordering principle:

- *Given an expression* `e1 op e2`, *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 `e1 op e2`:
- **Suppose:** `e1` requires *L* registers

  `e2` requires *R* registers

- If `e1` is evaluated first, we will need

  L registers to evaluate `e1`

  then *R* registers to evaluate `e2`

  PLUS ONE to hold the result of `e1`

- If `e1` is evaluated first, the maximum number of registers in use at once is *max(L, R+1)*
- If `e2` is evaluated first, the maximum number of registers in use at once is *max(L+1, R)*

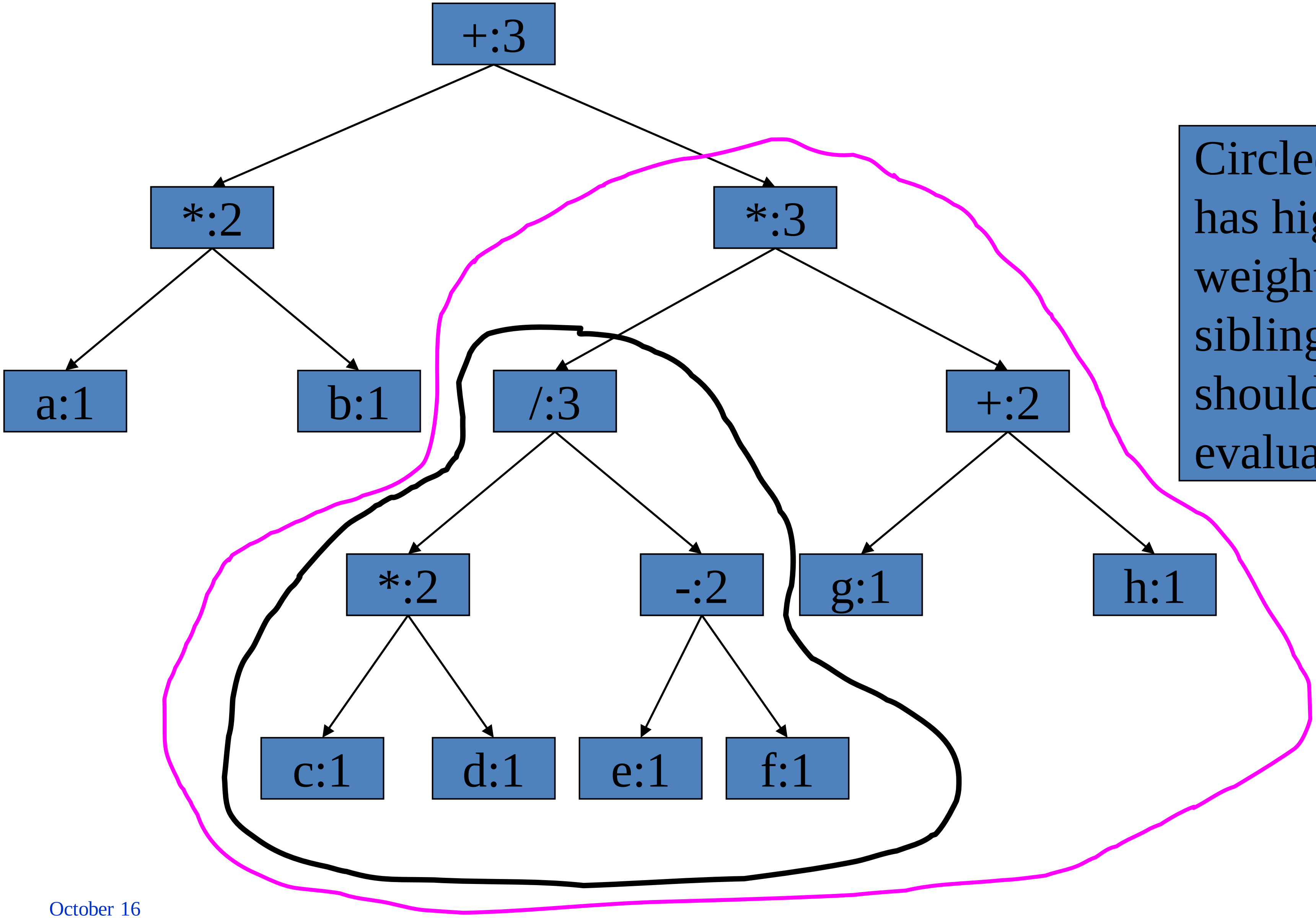
We choose the order which yields the smaller value

- 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 (assuming we do e first)
        = max [weight e1, (weight e2)+1]
   cost2 (assuming we do e first)
        = max [(weight e1)+1, weight e2 ]
```

Number shows the weight calculated for each subtree



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

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

```
transExp :: Exp → [Register] → [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:

```
transExp (Const n) (destreg:restofregs)
  = [LoadImm destreg n]

transExp (Ident x) (destreg:restofregs)
  = [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

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
addql #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 some Sony Clie handhelds etc (actually Motorola Dragonball Super VZ **MC68SZ328)**

(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:

- 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* operators
  - *k-1* intermediate values
  - $\lceil log_2 k \rceil$ registers required
- So the expression size accommodated using a block of *N* registers is proportional to 2^N

Function calls

- E.g.

  `price+lookup(type,qty/share)*tax`

  where `lookup` is a user-defined function with integer arguments, and integer result
- Changing order of evaluation may change result because of side-effects
- Register targeting can be used to make sure arguments are computed in the right registers
- At the point of the call, several registers may already be in use
  - But a function might be called from different call-sites
  - Each call site might have different registers in use

Function calls

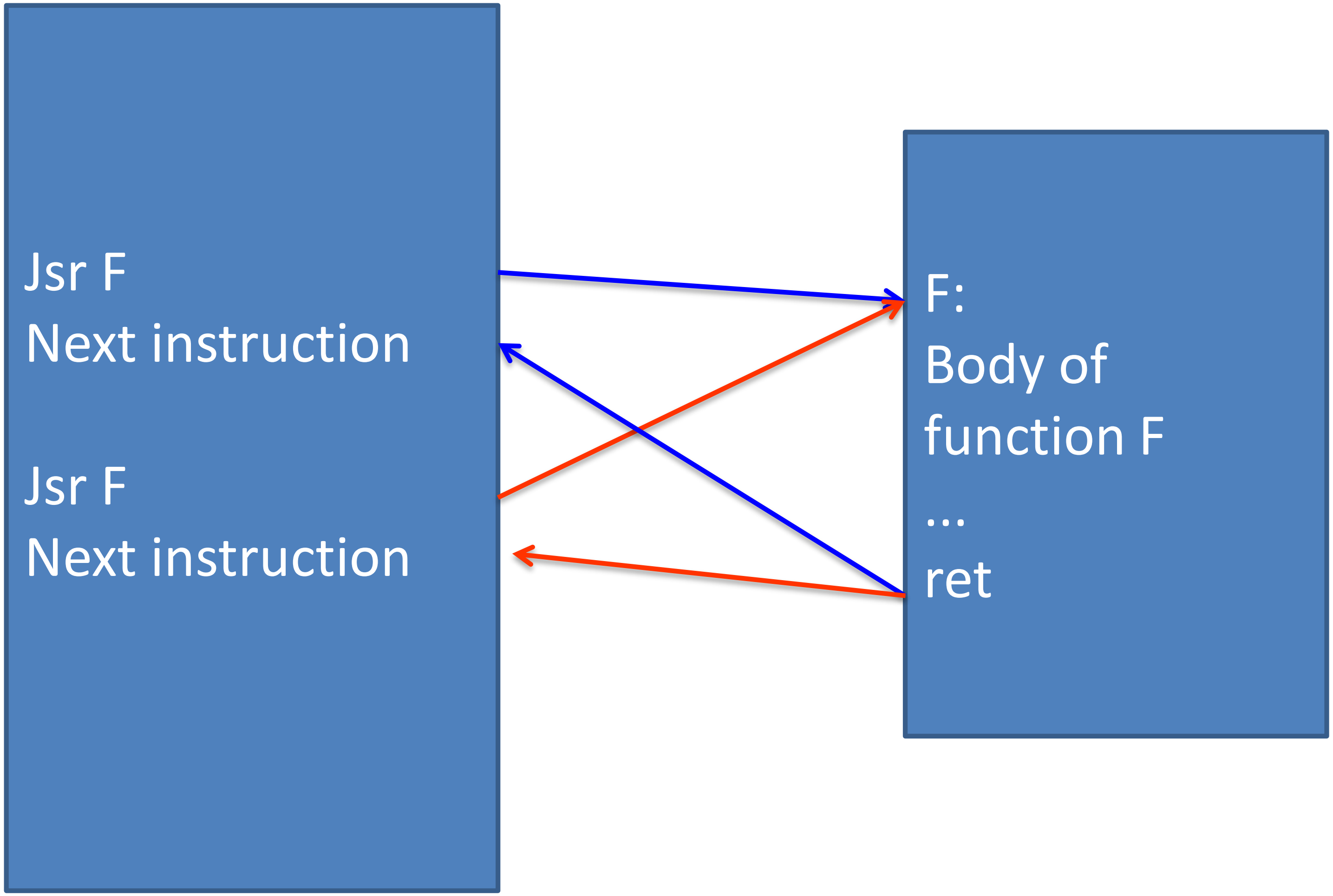
- Eg: (1+f(x))*((2*y)+(3/u-z)

- Which one should we evaluate first?

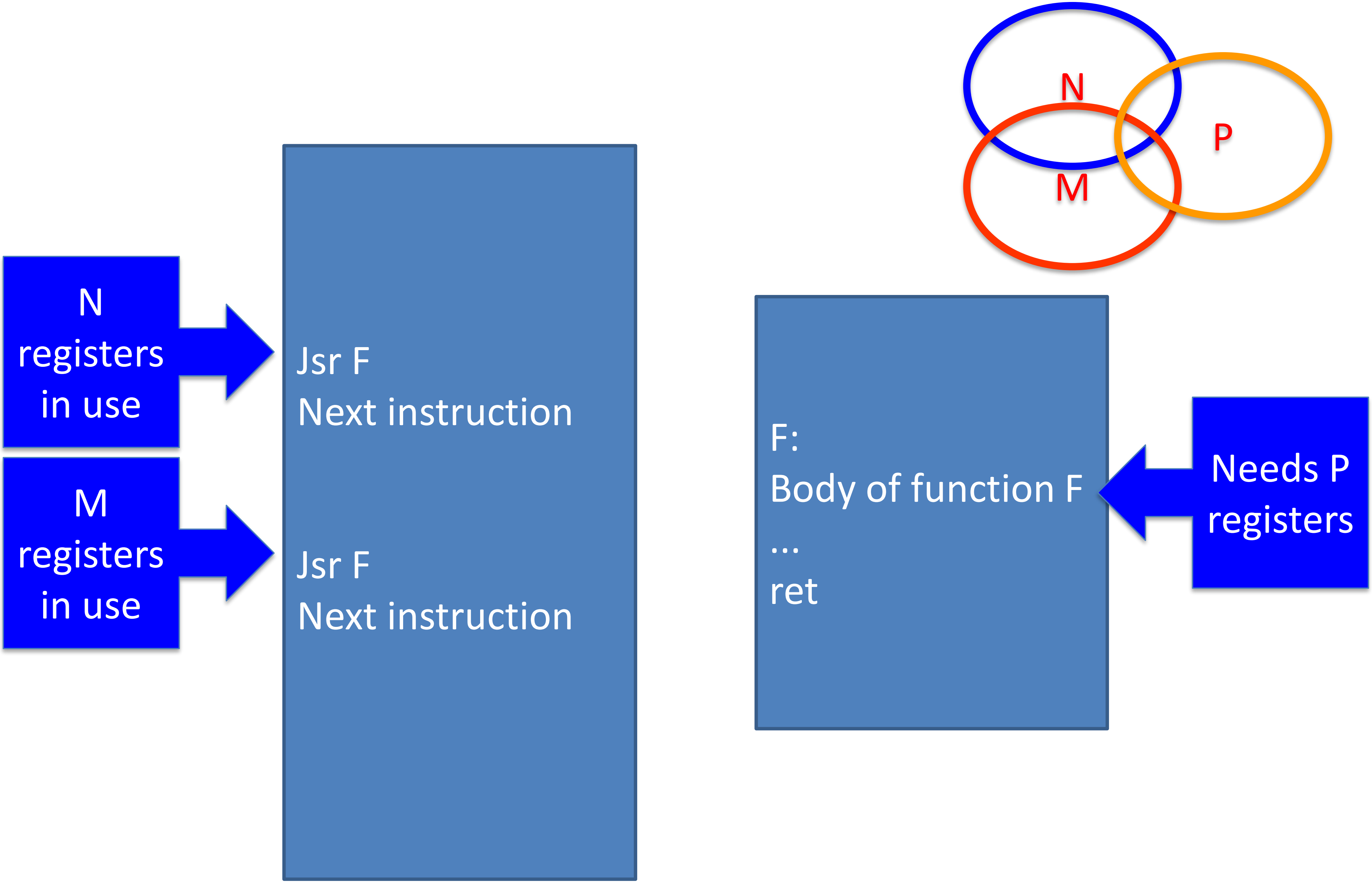
Save the registers…

- We must ensure that lookup doesn’t clobber any registers which contain intermediate values:
  - *Caller saves*: save registers being used by caller in case they are clobbered by callee
  - *Callee saves*: save only the registers which callee actually uses
- Neither protocol is always the best (examples?)

Calling a function/method



- A function might be called from different places
- In each case it must return to the right place
- Address of next instruction must be saved and restored



N registers in use

M registers in use

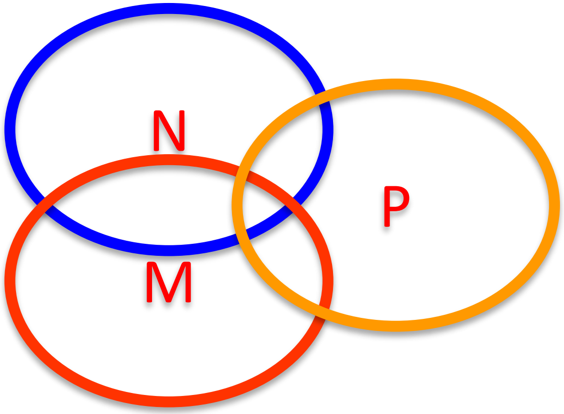
Jsr F
Next instruction

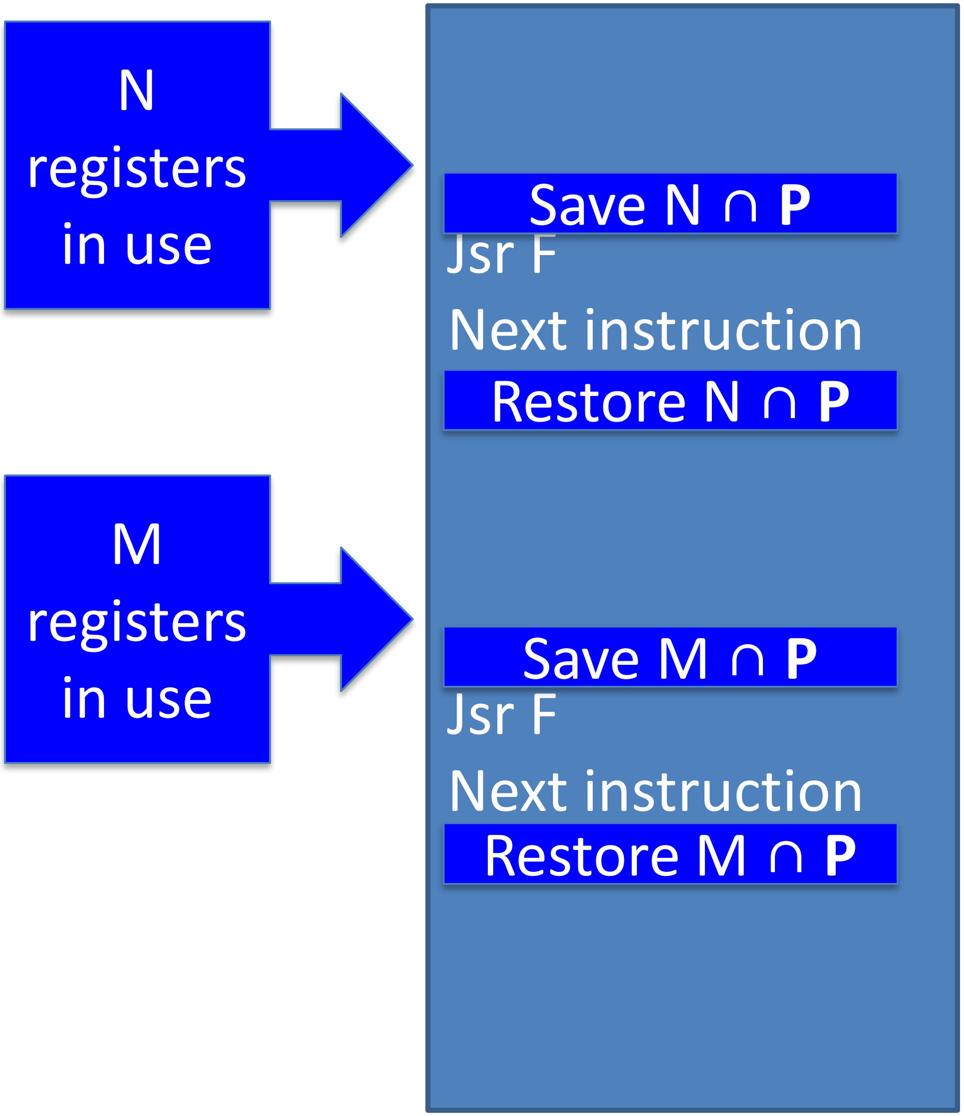
Jsr F
Next instruction

F:
Body of function F
…
ret

Needs P registers

Caller saves





N registers in use

Save N ∩ **P**

Jsr F

Next instruction

Restore N ∩ **P**

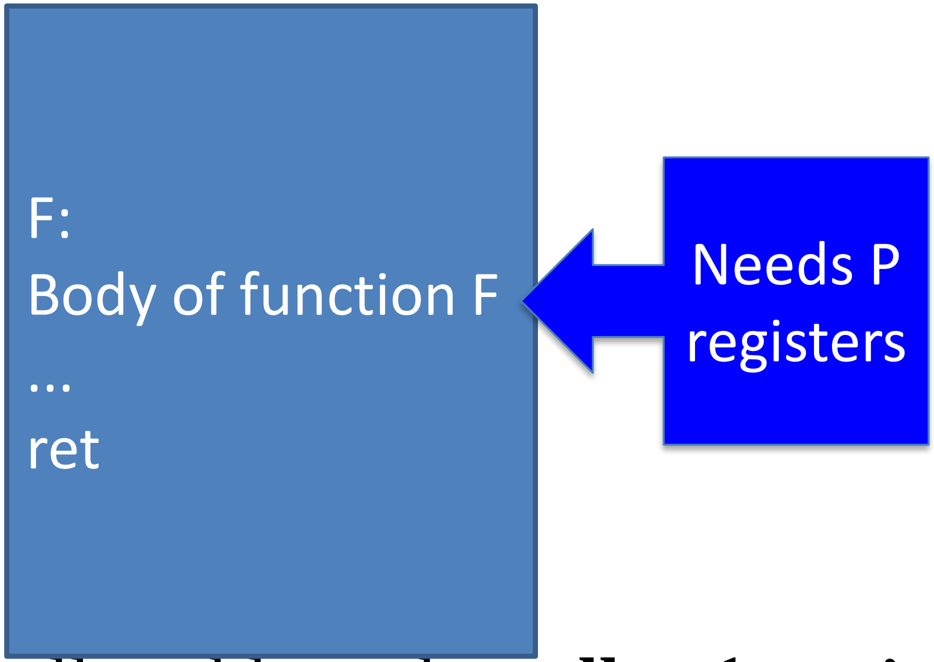
M registers in use

Save M ∩ **P**

Jsr F

Next instruction

Restore M ∩ **P**



F:

Body of function F

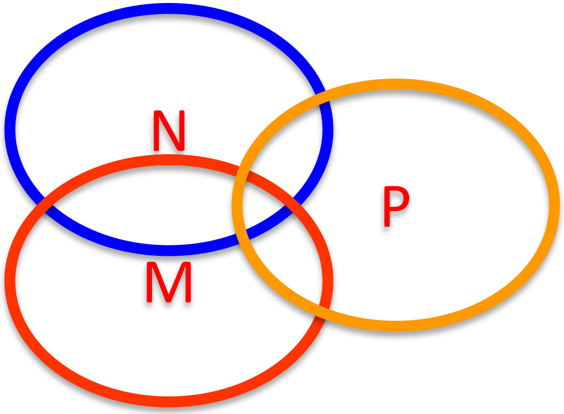
…

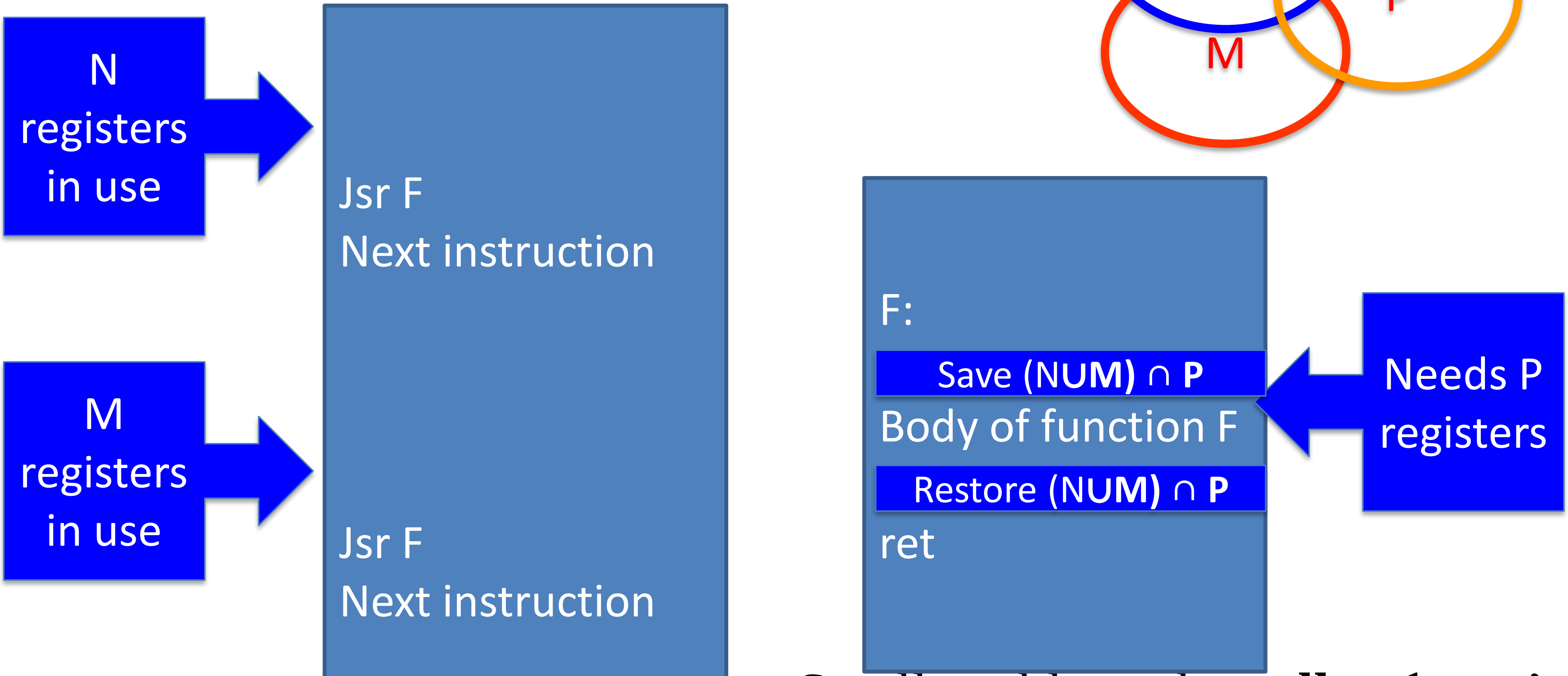
ret

Needs P registers

- Small problem: the **caller** doesn’t know which registers the **callee** will need – it has to save any register that *might* be used

Callee saves




N registers in use

Jsr F

Next instruction

M registers in use

Jsr F

Next instruction

F:

Save (NUM) ∩ **P**

Body of function F

Restore (NUM) ∩ **P**

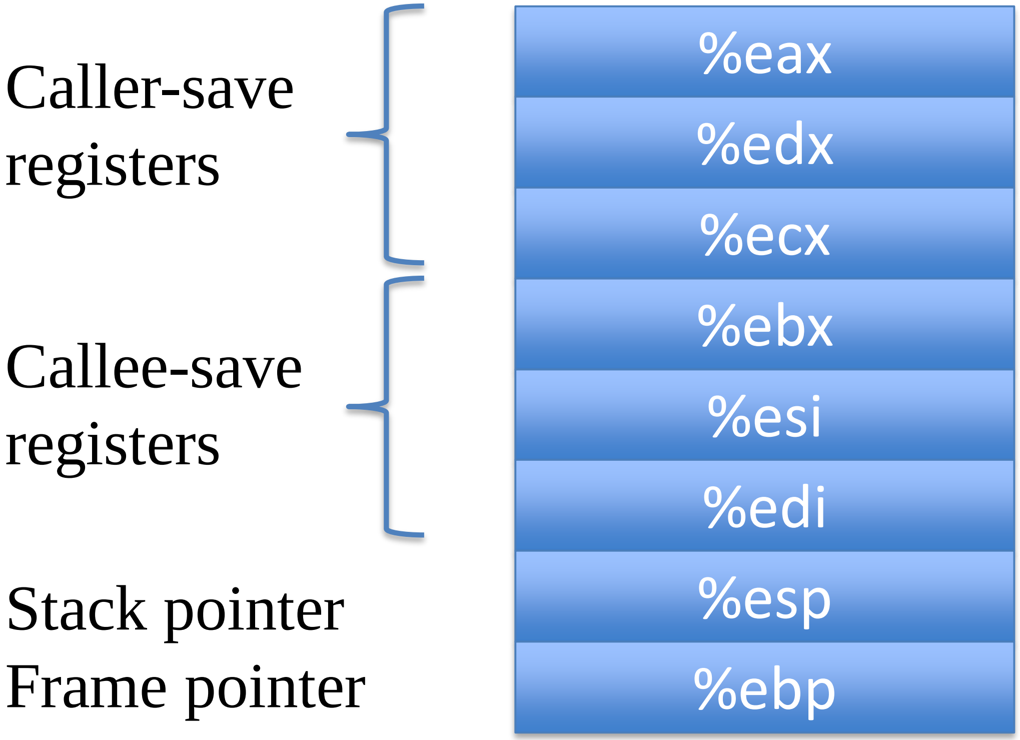
ret

Needs P registers

- Small problem: the **callee** doesn’t know which registers the **callers** will need – it has to save any register that *might* be in use

Intel IA32 register saving convention

- In general you can generate any code you like, but if you want it to work with other people's libraries, you have to obey the platform's Application Binary Interface (ABI)



Caller-save registers

Callee-save registers

Stack pointer

Frame pointer

%eax

%edx

%ecx

%ebx

%esi

%edi

%esp

%ebp

- Eg for Intel IA32 running Linux
- Actually there are many more rules
- Eg parameter passing, stack frame layout
- Eg virtual function tables for C++

http://www.cs.princeton.edu/courses/archive/spring04/cos217/lectures/IA32-III.pdf

ARM (32-bit) register saving convention

- “A subroutine must preserve the contents of the registers r4-r8, r10, r11 and SP”
- Similar rules apply to floating-point registers: s0-15 are caller-saves, s16-s31 are callee-saves

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/IHI0042E_aapcs.pdf

MIPS register saving convention

- R2-R3: function result
- R4-R7: incoming parameters (if there are >4 parameters, they are passed on the stack)
- R16-R23: callee-save
- R8-R15: caller-saves temporaries
- R16-R25 callee-save
- R31: procedure return address
- R30: stack pointer
- R29: Frame pointer (may be used for data in leaf procedures)
- R26, R27 reserved for operating system
- R28: pointer to global area
- (R0 is always zero)

Limitations of Sethi-Ullman register allocation scheme

- The tree-weighting translator is a typical syntax-directed (“tree walking”) translation algorithm: it works well in the terms of its input tree, but fails to exploit the *context* of the code being generated:
  - It makes no attempt to use registers to keep a value from statement to statement
  - In particular it does not try to use registers to store *variables*
  - Doesn’t handle repeated uses of a variable
- It is this exploitation of the context of the generated code which distinguishes an “optimising” compiler from the straightforward compilers we have considered so far
- Because of contextual dependences optimising compilers are very much harder to test, and therefore less reliable

Importance of more sophisticated register allocation

Example:

Naive implementation (roughly from `cc -S`):

```
void f() {
  int i, a;
  for (i=1; i<=10000000;
           i++)
   a = a+i;
}
```

```
 movl #1,a6@(-4)
 jra L99
L16:
 movl a6@(-4),d0
 addl d0,a6@(-8)
 addql #1,a6@(-4)
L99:
 cmpl #10000000,a6@(-4)
 jle L16
```

Unoptimised:

```
  movl #1,a6@(-4)
  jra L99
L16:
  movl a6@(-4),d0
  addl d0,a6@(-8)
  addql #1,a6@(-4)
L99:
  cmpl #10000000,a6@(-4)
  jle L16
```

5 instr’ns in loop, 4 memory references. Execution time on Sun-3/60: 16.6 seconds (1.66 microseconds/iteration)

Optimised:

```
  moveq #1,d7
L16:
  addl d7,d6
  addql #1,d7
  cmpl #10000000,d7
  jle L16
```

4 instructions in the loop, no references to main memory

Execution time on Sun 3/60: 8.3 seconds (0.83µseconds/iteration)

Notice that time per instruction has been reduced from 0.332µs to 0.208µs — because register instructions are faster than memory instructions

Importance of more sophisticated register allocation

Example: Naive implementation (roughly from `cc -S`):

```
void f() {
   int i, a;
   for (i=1; i<=1000000000;
                i++)
    a = a+i;
}
```

X86 code (slightly tidied but without register allocation)

```
    movl $1,-4(%ebp)
    jmp .L4
.L5
    movl -4(%ebp),%eax
    addl %eax,-8(%ebp)
    incl -4(%ebp)
.L4:
    cmpl $1000000000,-4(%ebp)
    jle .L5
```

Unoptimised:

```
    movl $1,-4(%ebp)
    jmp .L4
.L5
    movl -4(%ebp),%eax
    addl %eax,-8(%ebp)
    incl -4(%ebp)
.L4:
    cmpl $1000000000,-4(%ebp)
    jle .L5
```

5 instructions in the loop

Execution time on 2.13GHz Intel Core2Duo: 3.87 seconds (3.87 nanoseconds/iteration, 8.24 cycles)

Optimised:

```
    movl $1,%edx
.L6:
    addl %edx,%eax
    incl %edx
    cmpl $1000000000,%edx
    jle .L6
```

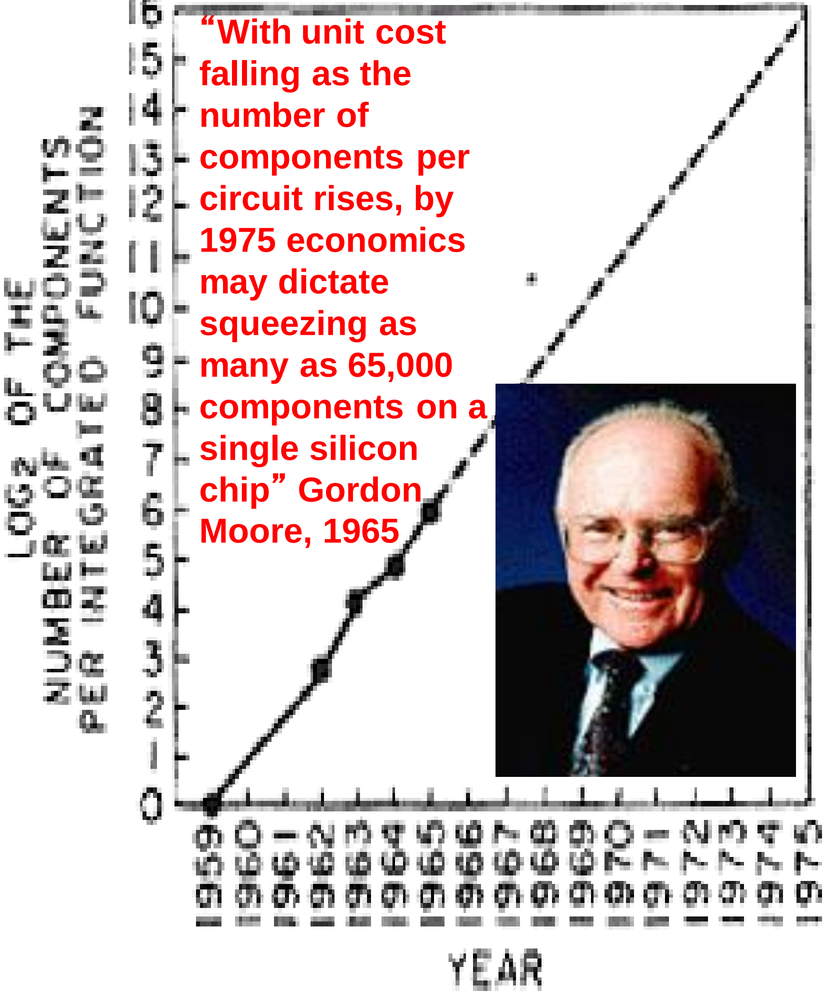
4 instructions in the loop, no references to main memory

Execution time on 2.13GHz Intel Core2Duo: 0.48 seconds (0.48 nanoseconds/iteration, 1.02 cycles)

Notice that time per instruction has been reduced from 0.77 nanoseconds to 0.12 — because register instructions are faster than memory instructions

Performance over time…

- Sun 3/60 introduced ca.1987
  - Based on 20MHz Motorola 68020+68881 FPU
  - No data cache
  - Unoptimised: 1.66us/iteration (33 cycles, 6.6 cycles per instruction)
  - Optimised: 0.83us/iteration (16.6 cycles, 4.15 cycles per instruction)
- Intel Xeon 2.2GHz introduced ca.2002
  - Based on Pentium 4 "Netburst" architecture
  - 8KB level 1 data cache, 512 KB level 2 data cache
  - Unoptimised: 2.8ns/iteration (6.16 cycles, 1.23 cycles per instructions)
  - Optimised: 0.7ns/iteration (1.54 cycles, 0.385 cycles per instruction)
- Moore's Law: microprocessor performance doubles every 18 months
  - 1987-2002 = 15 years = 10*18months
  - Predicts improvement of 2^10=1024
  - Unoptimised ratio: 1.66us:2.8ns = 592
  - Optimised ratio: 0.83us:0.7ns = 1185



"With unit cost falling as the number of components per circuit rises, by 1975 economics may dictate squeezing as many as 65,000 components on a single silicon chip" Gordon Moore, 1965

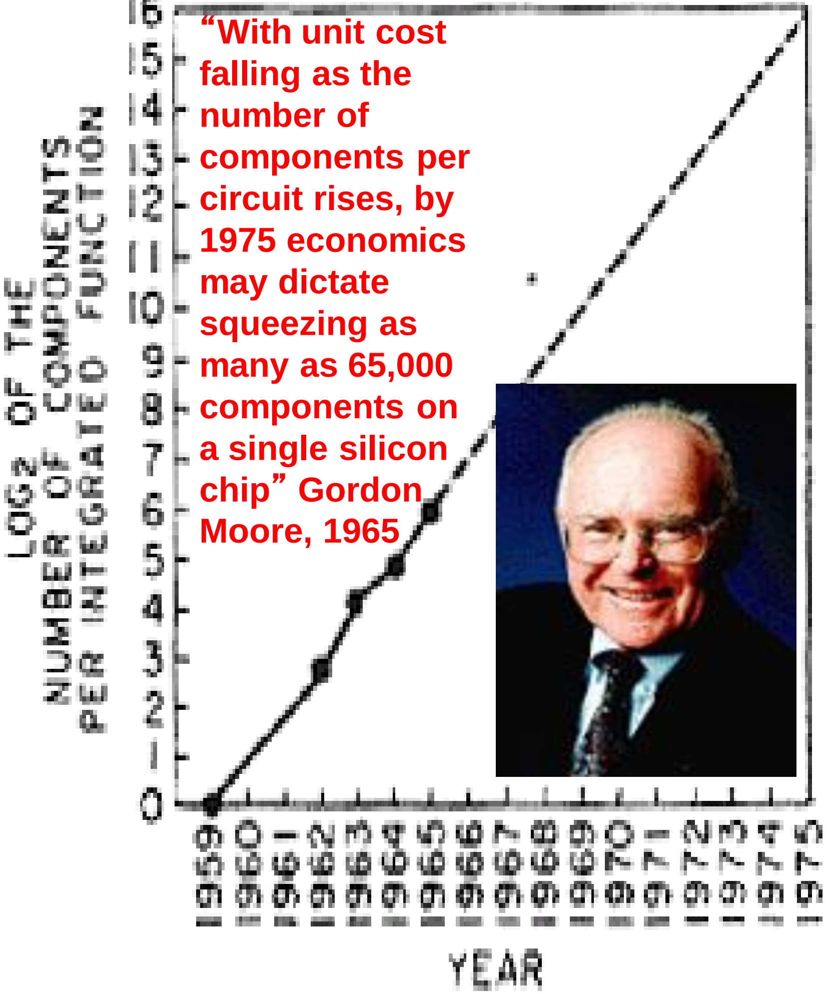
- How much longer can we expect Moore's Law to hold?
- What if it's another 15 years?

Performance over time…

- Sun 3/60 introduced ca.1987
  - Based on 20MHz Motorola 68020+68881 FPU
  - No data cache
  - Unoptimised: 1.66us/iteration (33 cycles, 6.6 cycles per instruction)
  - Optimised: 0.83us/iteration (16.6 cycles, 4.15 cycles per instruction)

- Intel Core2Duo 6420 “Conroe” introduced ca.2007
  - Two cores per chip
  - 32KB L1 data cache, 32KB L1 instruction cache
  - 4MB shared L2 cache
  - Unoptimised: 3.87 nanoseconds/iteration, 1.65 cycles per instruction
  - Optimised: 0.48ns/iteration (1.54 cycles, 0.255 cycles per instruction)

- Moore’s Law: microprocessor performance doubles every 18 months
  - 1987-2007= 20 years = 13.3*18months
  - Predicts improvement of 2^13.3=10085
  - Unoptimised ratio: 1.66us:3.87ns = 429
  - Optimised ratio: 0.83us:0.48ns = 1729



“With unit cost falling as the number of components per circuit rises, by 1975 economics may dictate squeezing as many as 65,000 components on a single silicon chip” Gordon Moore, 1965

- How much longer can we expect Moore’s Law to hold?
- That’s not what he said...

Common subexpressions

- Example:

```
a1 := b1 + s * k;
a2 := b2 + s * k;
```

- When the common subexpression is known to have the same value, we can write this as

```
t := s * k;
a1 := b1 + t;
a2 := b2 + t;
```

(where `t` is a new temporary variable introduced by the compiler)

- Unfortunately our clever weighted tree translation scheme cannot easily arrange for t to be stored in a register

To overcome limitations of simple syntax-directed scheme, need to consider *all* variables on equal terms: not just programmer’s variables, but all intermediate values during the computation

A brief look at a smarter allocator

- As an example of a more sophisticated register allocator we will look at ***graph colouring***.
- The algorithm consists of three steps:

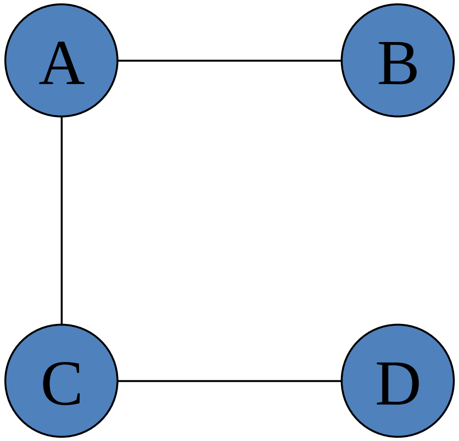
1. Use a simple tree-walking translator to generate an *intermediate code* in which temporary values are always saved in named locations. (In the textbook this is referred to as "three address code": resembles assembler but with unlimited set of named registers)
2. Construct the *interference graph*: the nodes are the temporary locations, and each pair of nodes is linked by an arc if the values must be stored simultaneously—if their "live ranges" overlap
3. Try to colour the nodes, with one colour for each register, so no connected nodes have the same colour

Example:

- Program fragment:

```
A := e1
B := e2
...
... B ...     ← B used
C := A+B ← A and B used
...
D := A*5 ← A used
... D ...     ← D used
... C ...     ← C used
```

- Interference graph:

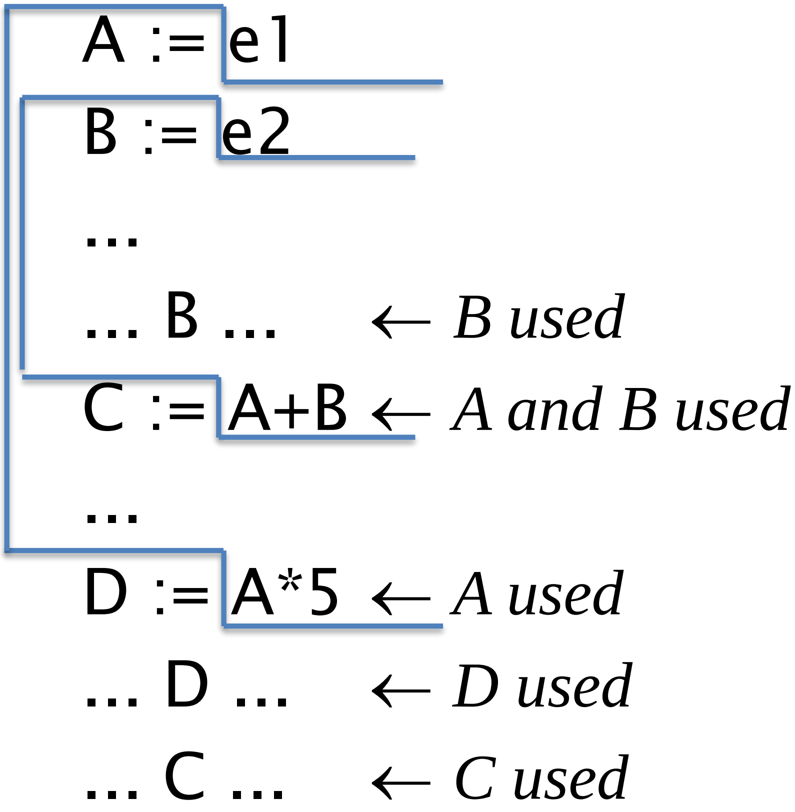


Live ranges of A and B, A and C, C and D overlap

B and C do not overlap; could be stored in same register

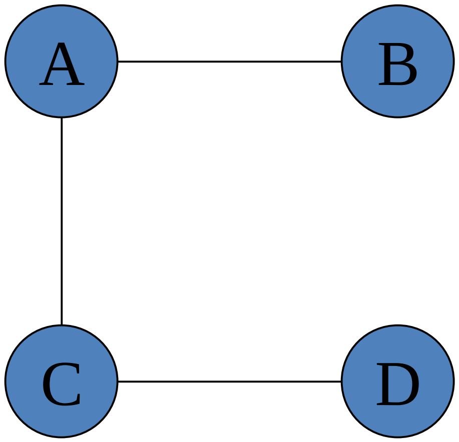
Example:

- Program fragment:



```
A := e1
B := e2
...
... B ...    ← B used
C := A+B ← A and B used
...
D := A*5 ← A used
... D ...    ← D used
... C ...    ← C used
```

- Interference graph:



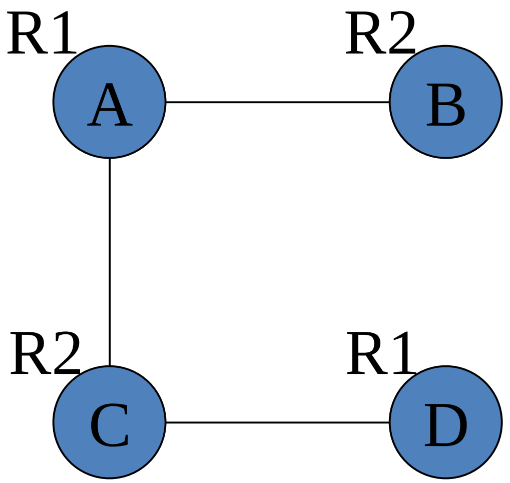
Live ranges of A and B, A and C, C and D overlap

B and C do not overlap; could be stored in same register

Colouring

- We colour the nodes, with one colour for each register, so no connected nodes have the same colour.
- Because if a pair of nodes are linked, their live ranges overlap so they can't be stored in the same place. If they *do not* overlap, they can be assigned the same register if necessary

- Example interference graph after colouring:



Register-allocated code:

Three-address code	After register allocation
A := e1	R1 := e1
B := e2	R2 := e2
…	…
… B …	… R2 …
C := A+B	R2 := R1+R2
…	…
D := A*5	R1 := R1*5
… D …	… R1 …
… C …	… R2 …

Graph colouring: implementation

- Finding the live ranges is easy in straight-line code
- In code with branching and loops, data flow analysis is needed (see EaC Section 9.2.1, Appel Chapter 10, Dragon book pp.608ff).
- The problem of determining whether a given graph can be coloured with a given number of colours is very hard - “NP Complete”
- This is not such a serious problem as a good, fast heuristic is adequate and not hard to invent (see Eac Sections 13.5.4-5, Appel 11.1, Dragon book pp.545-546)

Spilling

- If attempt to colour the graph using available registers fails, must *spill* some register
  - i.e. choose an arc in the graph and break it
  - i.e. choose a variable whose live range is causing trouble, and split its live range
  - Do this by adding code to store it to memory and reload it later
  - Then redo analysis:
    - Update interference graph.
    - Attempt colouring again; if no success, split another live range
  - Key: strategy to choose values to spill:
    - Avoid adding spill code to the innermost loop (e.g. prioritize values by their nesting depth).
    - Split a live range that will enable colouring

Some register spill options

```
A = ....

For  (i-0; i<N; ++i) {
 .... // high register pressure
}

 = A
```

```
A = ....
For  (i-0; i<N; ++i) {
 ....
     = A;
}
For  (i-0; i<N; ++i) {
 .... // high register pressure
}
For  (i-0; i<N; ++i) {
 ....
     = A;
}
```

- Allocate a temporary to the stack?
- Allocate to a register but spill it to the stack - split the live range
- Profile-directed?

```
A = ....
For  (i-0; i<N; ++i) {
 .... // high register pressure
 if (...) {
     = A;
 }
}
```

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 based on graph colouring

Textbooks

- EaC covers the Sethi-Ullman algorithm briefly – see Section 7.3.1 (page 315)
- EaC’s recommended solution is more complex and ambitious: they separate the problem into three stages:
  - instruction selection (Chapter 11), by tree pattern matching
  - instruction scheduling (Chapter 12), accounting for pipeline stalls
  - register allocation (Chapter 13) by graph colouring (Section 13.5)
- Appel covers the Sethi-Ullman algorithm in section 11.5 (page 260)
- Appel concentrates on graph colouring – see Chapter 11
- Graph colouring relies on live range analysis, which is covered in Chapter 10

Code generation in Appel’s textbook

- In Appel’s compiler, the input for the code generation phase is a low-level intermediate representation (IR), which is defined in Figure 7.2 (page 157)
- The translation from the AST to the IR Tree is where:
  - ‘For’ loops are translated into ‘Label’ and jump
  - Array access is translated into pointer arithmetic + bounds checking
- The IR tree does not specify how the low-level work should be packed into machine instructions, nor which registers should be used; two phases follow:
  1. Instruction selection (Chapter 9)
  2. Register allocation (Chapter 11)
- Instruction selection works with ‘temporaries’ – names for locations for storing intermediate values; register allocation determines how temporaries are mapped to registers

Code generation in Appel’s textbook…

- The data type for Instructions is defined on page 210 (section 9.3)
- In an instruction set like Intel’s IA-32, an instruction may often perform several elementary operations in the IR Tree
- Instruction selection consists of pattern-matching – finding chunks of IR Tree corresponding to an instruction; Appel implements this in much the same way as we did in Haskell – see Programs 9.5 and 9.6 (page 213-4)
- Appel’s ‘translate_exp’ function ‘munchExp’ has the effect of emitting the assembler instructions, and returns as its result the temporary in which the result will be stored

Code generation in Appel's textbook…

- We allocate registers *before* (or at the same time as) instruction selection
- Appel selects instructions first, *then* allocates registers – see discussion Section 9.3 (page 209)
- This is not a straightforward decision…
  - You may be able to avoid using a register for an intermediate value if you can use a sophisticated addressing mode
  - You may run out of registers, so some temporaries will have to be stored in memory – which changes the addressing modes you will be able to use
- Our approach leads to a much simpler compiler; see discussion on page 269