Register Allocation (1)

- **Register**: One of a small number of very fast storage elements *internal* to a CPU.
- **Register allocation**: Which register to use for what purpose when.
- We have seen code generation for TAM, a simple *stack machine*:
  - All instructions target a stack.
  - A few *dedicated* registers (e.g., SB, LB, ST).
  - Register allocation thus a non-issue: allocation decided once and for all by the *design* of the TAM.

Register Allocation (2)

- Most real computers are *register machines*:
  - Most instructions **target registers**; that is,
    - instead of instructions like
      \[
      \text{ADD} \quad (\text{arguments from stack, result to stack})
      \]
    - they have instructions like
      \[
      \text{ADD} \ R3, \ R1, \ R2 \\
      (R3 := R1 + R2)
      \]
  - Additionally, instructions for **memory access**.
  - Stacks are *implemented* using memory, registers, and memory access instructions.

Register Allocation (3)

- Most real computers are *register machines*:
  - . . .
  - Very few registers, typically 8–32 word-sized ones, or 32–128 *bytes* of memory.
  - Cf. modern programs that often use *hundreds of Megabytes* of memory.
  - Additionally, registers may be:
    - general purpose
    - special purpose
Register Allocation (4)

The problem:

- On the one hand: Not enough registers to keep all data in registers all the time; most data has to be stored in the main memory.
- On the other hand: 
  - Have to use some registers because of the way the instruction set is designed.
  - Want to use many registers because registers are very fast.
  - Preferable to use registers for frequently used data over seldomly used data.

Register Allocation (5)

Register allocation is thus an optimisation problem:

- minimise the memory traffic (loads and stores) by using registers
- subject to:
  - not exceeding the available number of registers;
  - additional constraints imposed by some registers having a special purpose or not being fully general.

Register Allocation (6)

- What really is desirable is to minimise execution time and/or the size of the target code.
- However, minimising the number of (executed) load and store instructions usually reduce both the execution time and the number of generated instructions (size of the target code).

A Simple Register Machine

- Registers:
  - Rn: general purpose registers
  - SB: Stack Base
  - LB: Local Base (stack frame)
  - ST: Stack Top
- Some instructions (Ri etc. include SB, LB, ST):
  - load Ri, [Rj + d]
  - store Ri, [Rj + d]
  - add Ri, Rj, Rk \( (Ri := Rj + Rk) \)
  - mul Ri, Rj, Rk \( (Ri := Rj \times Rk) \)
- Displacement/offset d in bytes.
Exercise: RM Code Generation (1)

Given:

- Variables
  - \(x\) at SB + 0
  - \(y\) at SB + 4
  - \(z\) at SB + 8

- General purpose registers R0, R1, ... R9

Generate code for:

\[ z := z \times (x + y) \]

Exercise: RM Code Generation (2)

One possible answer:

```
load R0, [SB + 0] ; x
load R1, [SB + 4] ; y
add R2, R0, R1
load R3, [SB + 8] ; z
mul R4, R3, R2
store R4, [SB + 8] ; z
```

What if there were fewer registers available?

How many do you need?

Exercise: RM Code Generation (3)

Another possibility using only R0 and R1:

```
load R0, [SB + 0] ; x
load R1, [SB + 4] ; y
add R0, R0, R1
load R1, [SB + 8] ; z
mul R1, R1, R0
store R1, [SB + 8] ; z
```

Stack Frame or Activation Record

<table>
<thead>
<tr>
<th>address</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>LB</td>
<td>arguments</td>
</tr>
<tr>
<td>...</td>
<td>dynamic link</td>
</tr>
<tr>
<td>LB</td>
<td>static link</td>
</tr>
<tr>
<td>LB + 4</td>
<td>return address</td>
</tr>
<tr>
<td>LB + 8</td>
<td>local variables</td>
</tr>
<tr>
<td>...</td>
<td>temporary storage</td>
</tr>
</tbody>
</table>

where:

- \(argOffset = \text{size}(arguments)\)
- \(tempOffset = 12 + \text{size}(variables)\)

(Offsets in bytes for register machine.)
Register Machine Code Generation (1)

We can implement a code generation function `evaluate` in a similar way to the stack machine code generator, except that it returns the register in which the result will be stored.

Assuming a code generation monad `CG` for keeping track of generated code, free registers, etc., we’d get:

```
evaluate : Expression → CG Reg
```

(ignoring bookkeeping arguments such as scope level and environment.)

Register Machine Code Generation (2)

Operation for getting a currently free register:

```
freeReg : CG Reg
```

In a naive scheme (or as a precursor to a register allocation step), `freeReg` would always return a previously unused register:

```
evaluate [\[E_1 + E_2\]] = \do
  r_1 \leftarrow evaluate E_1
  r_2 \leftarrow evaluate E_2
  r \leftarrow freeReg
  emit (Add r r_1 r_2)
  return r
```

Example: A Simple Function

```
var n: Integer;
...
fun f(x,y: Integer): Integer =
  let
    z: Integer
  in begin
    z := x * x + y * y;
    return n * z
  end
```

We will consider the body less the details of storage allocation for `z` and `return`.

Naive Register Machine Code

Code for `z := x*x + y*y; return n*z`:

```
load R0, [LB - 8]; offset(x) = -8
load R1, [LB - 8]
mul R2, R0, R1; R2 := x^2
load R3, [LB - 4]; offset(y) = -4
load R4, [LB - 4]
mul R5, R3, R4; R5 := y^2
add R6, R2, R5; R6 := x^2 + y^2
store R6, [LB + 12]; offset(z) = 12
load R7, [SB + 168]; offset(n) = 4 × 42
load R8, [LB + 12]
mul R9, R7, R8; R9 := n(x^2 + y^2)
```
Stack Machine Code

TAM-code for the example for comparison:

```plaintext
LOAD [LB - 2] ; x  ADD
LOAD [LB - 2] ; x  STORE [LB + 3] ; z
MUL
LOAD [LB - 1] ; y  LOAD [SB + 42] ; n
LOAD [LB - 1] ; y  MUL
MUL
```

Note: all offsets are in words (4 bytes) for the TAM (stack of word-sized memory cells).

Notes on the Naive Code

**Fact**: reading/writing memory is extremely slow compared to reading/writing registers.
- The naive code is inefficient because many unnecessary memory accesses.

**Fact**: the number of registers is strictly limited (from a few to a few dozen)
- The naive code-generation scheme could fail because it risks running out of registers.

Better Code (1)

Basic, ad-hoc, register allocation:
- Allocate registers for \( x \) and \( y \): saves reading them twice.
- Allocate a register for \( z \): saves having to write it to memory!

Note: even the naive code-generation scheme employed a simple register allocation strategy for keeping intermediate results in registers as opposed to storing them in memory.

Better Code (2)

**R0 used for** \( x \), **R1 for** \( y \), **R2 for** \( z \), **R5 for** \( n \).**

```plaintext
load R0, [LB - 8] ; offset(\( x \)) = -8
mul R3, R0, R0 ; \( R3 := x^2 \)
load R1, [LB - 4] ; offset(\( y \)) = -4
mul R4, R1, R1 ; \( R4 := y^2 \)
add R2, R1, R1
mul R6, R5, R2 ; R6 := n(x^2 + y^2)
```

- Fewer loads and stores
- Shorter code
- Fewer registers used
Saving Registers Across Calls (1)

Assume the calling convention is that the first three arguments are passed in registers R0, R1, R2 and the result is returned in R0.

Consider the following code fragment:

\[
\begin{align*}
&\text{add} \ R5, R6, R7 \quad ; \ R5 := x + y \\
&\text{load} \ R0, [SB + 168] \quad ; \ R0 := n \\
&\text{call} \ \text{factorial} \quad ; \ R0 := n! \\
&\text{mul} \ R0, R5, R0 \quad ; \ R0 := (x + y) \times n!
\end{align*}
\]

But what if \text{factorial} uses some registers, in particular R5 as in use across the call?

Saving Registers Across Calls (2)

Two basic approaches:

- **Caller Saves**: Caller saves registers that are in use; risks saving registers callee actually will not use.

- **Callee Saves**: Callee saves registers that it will use; risks saving registers that actually were not in use in caller.

In practice, a mixed approach is often adopted: some registers are designated caller-saves, others callee-saves.

Saving Registers Across Calls (3)

Assuming R5 is a caller-saves register and the only register that is in use across the call, the code fragment becomes:

\[
\begin{align*}
&\text{add} \ R5, R6, R7 \quad ; \ R5 := x + y \\
&\text{load} \ R0, [SB + 168] \quad ; \ R0 := n \\
&\text{store} \ R5, [LB + 30] \quad ; \ \text{Save} \ R5 \\
&\text{call} \ \text{factorial} \quad ; \ R0 := n! \\
&\text{load} \ R5, [LB + 30] \quad ; \ \text{Restore} \ R5 \\
&\text{mul} \ R0, R5, R0 \quad ; \ R0 := (x + y) \times n!
\end{align*}
\]

(LB + 30 is assumed to be address of free space in the temporary area.)

Automatic Register Allocation

How can we:

- **Automatically** decide which registers to use?

- Keep the number of registers used down?
  - Only a fixed, small number of registers available.
  - Each register must thus be used for many purposes.
Register Pressure

*Register Pressure*: the number of registers used by a code fragment.

Desirable to keep register pressure low:

- Minimizing the pressure maximizes the size of the code for which no auxiliary storage (primary memory) is needed.
- Low pressure means fewer registers to preserve (in primary memory) across subroutine calls (both caller and callee saves schemes).

Liveness (1)

- Need to take *liveness* of variables and intermediate results into account to make it possible to use *one* register for *many* purposes.
- A variable \( v \) is *live* at point \( p \) if there is an execution path from \( p \) to a use of \( v \) along which \( v \) is not updated.
- No need to keep dead variables in registers!

Liveness (2)

Example:

```
1  x := 3 * m;
2  y := 42 + x;
3  z := y * x;
4  if z > 0 then u := x else u := 0;
5  y := u;
6  return y;
```

- \( x \) is *live* immediately before line 4 because it *may* be used at line 4.
- \( y \) from line 2 is *dead* immediately before line 4 because \( y \) is *updated* before being used again.

Liveness (3)

Example:

```
1  x := 3 * m;
2  y := 42 + x;
3  z := y * x;
4  if z > 0 then u := x else u := 0;
5  y := u;
6  return y;
```

- \( u \) is *dead* before line 4 because it is updated in *both* branches of the *if* at line 4.
**Liveness (4)**

But consider this variation instead:

1. \( x := 3 \times m; \)
2. \( y := 42 + x; \)
3. \( z := y \times x; \)
4. if \( z > 0 \) then \( u := x \) else \( v := 0; \)
5. \( y := u; \)
6. return \( y; \)

- Now \( u \) is **live** before line 4 because there exists at least one path to the use of \( u \) at line 5 along which \( u \) is not updated.

**Liveness for the Running Example**

- \( x \) (R0) and \( y \) (R1) only used once.
- \( z \) (R2) is alive only for a short time, and only once \( x \) and \( y \) are dead.
- \( n \) (R5), interm. results (R4, R6) also short-lived.

**Exercise: Liveness**

Consider:

1. \( i := m; \)
2. \( n := 1; \)
3. while \((i < 10)\) do begin
   4. \( n := n \times p; \)
   5. \( i := i + 1 \)
   6. end
   7. return \( n; \)

Which of \( i, m, n, p \) are live immediately before:
- line 1
- line 5
- line 3
- line 7

**Graph Colouring**

Common approach for register allocation. Idea:

- Represent each variable by a node in a graph. Called **interference graph**.
- Add an edge between two nodes if the variables are live simultaneously.
- Colour the graph so that no two adjacent nodes get the same colour, using as **few colours** as possible.
- Each colour corresponds to a register.
- Hard optimization problem (NP-complete).
Example: Interference Graph

Consider:

\[
\begin{align*}
y &:= x \times x; \\
z &:= y + 42; \\
\text{return } y \times z
\end{align*}
\]

Interference graph:

\[
\begin{array}{c}
x \\
y \\
z
\end{array}
\]

How many colours?

Graph Colouring for the Running Ex.

load R0, [LB - 8] ;  \quad offset(x) = -8
mul R3, R0, R0 ;  \quad R3 := x^2
load R1, [LB - 4] ;  \quad offset(y) = -4
mul R4, R1, R1 ;  \quad R4 := y^2
add R2, R3, R4 ;  \quad R2 := x^2 + y^2
load R5, [SB + 168] ;  \quad offset(n) = 4 \times 42
mul R6, R5, R2 ;  \quad R6 := n(x^2 + y^2)

- Draw and colour the interference graph
- Use the result to do a register allocation with the minimal number of registers.

Code Using Minimal Number of Regs

The number of registers used reduced from 7 to 2:

load R0, [LB - 8] ;  \quad offset(x) = -8
mul R0, R0, R0 ;  \quad R0 := x^2
load R1, [LB - 4] ;  \quad offset(y) = -4
mul R1, R1, R1 ;  \quad R1 := y^2
add R0, R0, R1 ;  \quad R0 := x^2 + y^2
load R1, [SB + 168] ;  \quad offset(n) = 4 \times 42
mul R0, R1, R0 ;  \quad R0 := n(x^2 + y^2)

Implementation

Code generation usually proceeds in two passes:

1. Generate code assuming \textit{arbitrarily} many \textit{virtual} registers (essentially the “naive” approach).

2. Use graph colouring to bind each virtual register to a \textit{physical} register.

Note: above we started from a code where basic (ad hoc) register allocation already had been done for illustrative purposes.
Register Spilling

- What if the register pressure exceeds the number of available registers?
  - **Register Spilling**: storing the content of a register into memory so as to free it and thus reduce the register pressure.
  - Intermediate results stored into the the "temporary" storage area of the stack frame/activation record.
  - Deciding *which* register(s) to spill is (another) hard optimization problem.

Register Spilling: Example (1)

Consider:

\[ x \times x + y \times z \]

If three or more registers available:

- load R0, [LB - 8] ; \( \text{offset}(x) = -8 \)
- mul R0, R0, R0 ; \( R0 := x^2 \)
- load R1, [LB - 4] ; \( \text{offset}(y) = -4 \)
- load R2, [LB + 12] ; \( \text{offset}(z) = 12 \)
- mul R1, R1, R2 ; \( R1 := yz \)
- add R0, R0, R1 ; \( R0 := x^2 + yz \)

Register Spilling: Example (2)

\[ x \times x + y \times z \]

If only two registers available

- load R0, [LB - 8] ; \( \text{offset}(x) = -8 \)
- mul R0, R0, R0 ; \( R0 := x^2 \)
- load R1, [LB - 4] ; \( \text{offset}(y) = -4 \)
- store R0, [LB + 16] ; Temporary storage
- load R0, [LB + 12] ; \( \text{offset}(z) = 12 \)
- mul R0, R1, R0 ; \( R0 := yz \)
- load R1, [LB + 16] ; \( R1 := x^2 \)
- add R0, R0, R1 ; \( R0 := x^2 + yz \)

Is Fewer Registers Always Better? (1)

We have seen there are reasons to minimize the number of registers used:

- Ability to get by with as few registers as possible reduces likelihood of having to spill.
- Fewer registers to save and restore across subroutine calls.
Is Fewer Registers Always Better? (2)

But can there be downsides to not making use of all registers there are?

Consider:

```plaintext
add R2, R0, R1
store R2, [...]
mul R3, R0, R1
```

A superscalar CPU can execute the `add` and `mul` instructions in parallel because there is no data dependence between them.

Is Fewer Registers Always Better? (3)

Consider instead:

```plaintext
add R2, R0, R1
store R2, [...]
mul R2, R0, R1
```

One fewer registers used, but no longer possible to execute `add` and `mul` in parallel!

`mul` is anti-dependent on `store` (or Write After Read (WAR) dependent).

Reducing the number of used registers might have hurt the performance!

Is Fewer Registers Always Better? (4)

But then again, a really clever CPU might use hardware register renaming: using extra registers behind the scenes.


Commonly used; e.g. Pentium II/III/4, Athlon.

See e.g. Wikipedia for details.

Register Allocation: Complications

Register allocation may be further complicated by architecture-specific issues:

- Special purpose registers; e.g. dedicated registers for result of multiplication, memory addressing, etc.
- Registers of varying size.
- Non-uniform instruction set, and thus complicated interaction between code selection and register allocation.