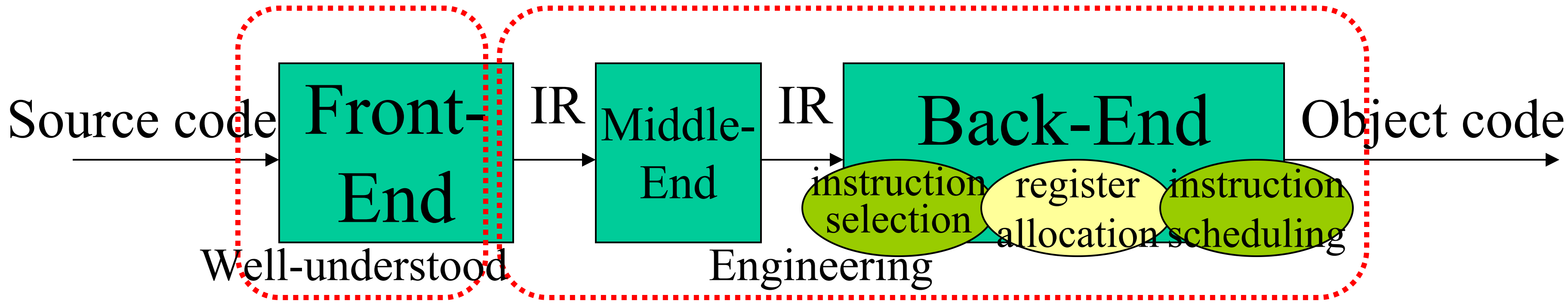
Lecture 16: Register Allocation



- Last Lecture: Instruction Selection.
- Register Allocation:
    - We assume a RISC-like (three-address) type of code.
    - The code makes use of an unbounded number of registers (**virtual registers**) but the machine has only a limited number of registers (**physical registers**), say *k*.
    - The task:
        - Produce correct *k* register code.
        - Minimise number of loads and stores (spill code) and their space.
        - The allocator must be efficient (e.g., no backtracking)

Background

- **Basic Block**: a maximal length segment of straight-line (i.e., branch-free) code. (Importance: strongest facts are provable for branch-free code; problems are simpler; strongest techniques.)
- **Local Register Allocation**: within a single basic block.
- **Global Register Allocation**: across an entire procedure (multiple BBs).
- **Allocation**: choose what to keep in registers.
- **Assignment**: choose specific registers for values.
- Modern processors may have multiple register classes:
  - General-purpose, floating-point, branch target, …
  - Problem: interactions between classes - Assume separate allocation for each class.
- **Complexity**: Only simplified cases of local allocation and assignment can be solved in linear time. All the rest (including global allocation – even for 1 register – and most sub-problems) are NP-complete. We need good heuristics!

Real compilers face real problems!

Liveness and Live Ranges

- Problem: What is the number of registers needed in a basic block?
  - Naïve: all occurrences of a variable to the same register.
  - Realistic: Compute a set of live ranges and use their name space.
- A value of a variable is **live** between its definition and its uses:
  - Find definitions (x←…) and uses (…←…x…)
  - From definition to last use is the "live range"
  - Can represent live range as an interval [i,j] in basic block.
- Over all instructions in the basic block, let:
  - MAXLIVE be the maximum number of values live at an instruction
  - k, the number of physical registers available.
    - If MAXLIVE ≤ k, allocation is trivial.
    - If MAXLIVE > k, some values must be spilled to memory.

Example / Exercise

Compute live ranges for all registers and MAXLIVE in the following Basic Blocks:

```
1.  load r1,@a
2.  load r2,2
3.  load r3,@b
4.  load r4,@c
5.  load r5,@d
6.  mult r1,r1,r2
7.  mult r1,r1,r3
8.  mult r1,r1,r4
9.  mult r1,r1,r5
10.store r1
```

r1	[1,6]	*	*	*	*	*	*				
r1	[6,7]						*	*			
r1	[7,8]							*	*		
r1	[8,9]								*	*	
r1	[9,10]									*	*
r2	[2,6]		*	*	*	*	*				
r3	[3,7]			*	*	*	*	*			
r4	[4,8]				*	*	*	*	*		
r5	[5,9]					*	*	*	*	*	

```
1.  load r1,@a
2.  load r2,@y
3.  mult r3,r1,r2
4.  load r4,@x
5.  sub  r5,r4,r2
6.  load r6,@z
7.  mult r7,r5,r6
8.  sub  r8,r7,r3
9.  add  r9,r8,r1
```

r1	[1,9]	*	*	*	*	*	*	*	*	*
r2	[2,5]		*	*	*	*				
r3	[3,8]			*	*	*	*	*	*	
r4	[4,5]				*	*				
r5	[5,7]					*	*	*		
r6	[6,7]						*	*		
r7	[7,8]							*	*	
r8	[8,9]								*	*
r9	[9,9]									*
# of live values		1	2	3	4	5	4	5	4	3
						-1		-1	-1	-1

Top-Down (Local) Allocation

- Allocator must reserve *f* registers to ensure feasibility (e.g., for use in computations that involve values allocated to memory; 2 to 4 depending on the target processor).
- Idea (frequency count algorithm): keep *k–f* most frequently used values in the BB in a register; use *f* for the rest:
  - 1. Count number of uses for each virtual register.
  - 2. Assign top *k-f* virtual registers to physical registers.
  - 3. Rewrite code: if a virtual register was assigned to a physical register, replace. Else spill: use reserved registers to load before use and store after definition.
- Weakness: a value heavily used in the 1st half of the basic block and unused in the 2nd half, essentially wastes the register for the latter.

Example

- Assume 3 physical registers – two needed for feasibility.

```
1. load r1,@a
2. load r2,@y
3. mult r3,r1,r2
4. load r4,@x
5. sub  r5,r4,r2
6. load r6,@z
7. mult r7,r5,r6
8. sub  r8,r7,r3
9. add  r9,r8,r1
```

r1	2	[1,9]	*	*	*	*	*	*	*	*	*
r2	2	[2,5]		*	*	*	*				
r3	1	[3,8]			*	*	*	*	*	*	
r4	1	[4,5]				*	*				
r5	1	[5,7]					*	*	*		
r6	1	[6,7]						*	*		
r7	1	[7,8]							*	*	
r8	1	[8,9]								*	*
r9	-	[9,9]									*
# of live values			1	2	3	4	4	4	4	3	2

```
1. load r1,@a
2. load r2,@y
3. mult r3,r1,r2
4. store r3 //spill r3
5. load r3,@x
6. sub  r3,r3,r2
7. load r2,@z
8. mult r3,r3,r2
9. load r2, … //spilled value
10.sub  r3,r3,r2
11.add  r3,r3,r1
```

r1 is assigned to the most commonly used value (ok, there is a tie; we choose the first one), and r2 and r3 are used for feasibility!

This assumes that the compiler can realize that **y** is already in register **r2**, hence it is not necessary to do a **load r2,@y** again!

Bottom-Up (Local) Allocation

- Let multiple values occupy a single register – Best's algorithm:

```
for each operation, i, 1 to N  (op vr3, vr2, vr1)
    ensure that vr1 is in r1
    ensure that vr2 is in r2
    if r1 not needed after i, free(r1)
    if r2 not needed after i, free(r2)
    allocate r3 for vr3
    emit code – op r3,r2,r1
```

- **ensure**: if a vr is not in a physical register, **allocate** register and make sure that occurrences of vr are tied to this physical register.
- **allocate**: return a free physical register, or select the register that is used farthest in the future, store its value and return it.
- Due to Sheldon Best (1955) – often reinvented. Many have argued for its optimality…
- What does it remind you?

Example

By spilling the value here, note that MAXLIVE ≤ 3

- Assume 3 physical registers

```
1. load r1,@a
2. load r2,@y
3. mult r3,r1,r2
4. load r4,@x
5. sub  r5,r4,r2
6. load r6,@z
7. mult r7,r5,r6
8. sub  r8,r7,r3
9. add  r9,r8,r1
```

r1	2	[1,9]	*	*	*	*	*	*	*	*	*
r2	2	[2,5]		*	*	*	*				
r3	1	[3,8]			*	*	*	*	*	*	
r4	1	[4,5]				*	*				
r5	1	[5,7]					*	*	*		
r6	1	[6,7]						*	*		
r7	1	[7,8]							*	*	
r8	1	[8,9]								*	*
r9	-	[9,9]									*
# of live values			1	2	3	4	4	4	4	3	2

```
1. load r1,@a
2. load r2,@y
3. mult r3,r1,r2
4. store r1 // spill the one used farthest
5. load r1,@x
6. sub  r1,r1,r2
7. load r2,@z
8. mult r1,r1,r2
9. sub  r1,r1,r3
10.load r2, … // load spilled value (load r2,@a)
11.add  r1,r1,r2
```

A 'clever' compiler may recognise that the store may not be needed since the value may be available from memory location @a (needs to guarantee that the value of this location won't change)

Exercise

(register allocation using Best's algorithm and 10 registers)

```
// a really useless program
mov r1,1
// generate 1 to 2^16
shl r2, r1, r1
shl r3, r2, r1
shl r4, r3, r1
shl r5, r4, r1
shl r6, r5, r1
shl r7, r6, r1
shl r8, r7, r1
shl r9, r8, r1
shl r10, r9, r1
shl r11, r10, r1
shl r12, r11, r1
shl r13, r12, r1
shl r14, r13, r1
shl r15, r14, r1
shl r16, r15, r1
shl r17, r16, r1
```

```
// now sum them spending registers to save adds
add r20, r1, r2
add r21, r3, r4
add r22, r5, r6
add r23, r7, r8
add r24, r9, r10
add r25, r11, r12
add r26, r13, r14
add r27, r15, r16
add r30, r20, r21
add r31, r22, r23
add r32, r24, r25
add r33, r26, r27
add r34, r30, r31
add r35, r32, r33
add r36, r35, r34
add r37, r36, r17
// wow! Now store the result
store r37,@a
// sum i=1 to 16 (2^i) is 2^17 -1!
// that was a really useless calculation…
add r40, r5, r1
shl r41, r1, r40
sub r42, r41, r1
store r41, @b
```

More complex scenarios

- Basic blocks (BB) rarely exist in isolation:

  BB1: … store r17, @a   is followed by

  BB2: load r12, @a …

  – Could replace load with a move; needs control-flow graph.
- Blocks with multiple (control-flow) predecessors:

  BB1: … store r4,@x   and   BB2: … store r7,@x followed by

  BB3: load r1, @x

  – What if BB1 has x in a register but BB2 not? (BB3 follows)
- Multiple basic blocks increase complexity:
  – How to compute the “farthest” in Best’s algorithm?

Global Register Allocation

- Taking a global approach:
  - Abandon the distinction between local and global.
  - Generalised frequency counts: weigh uses and defs and BBs; apply to each BB; try to remove loads and stores between adjacent BBs (Fortran H; IBM 360,370)
- Graph colouring paradigm:
  - Build an interference graph:
  - (try to) construct a k-colouring
    - Minimal colouring is NP-complete
    - Spill placement becomes a critical issue
  - Map colours onto physical registers.

Conclusion

- Register allocation in real cases is NP-complete.
- Best's algorithm, which has been reinvented repeatedly, performs well for local register allocation.

- Reading:
  - Aho2, pp. 553-556; Aho1, pp.541-546 (too condensed)
  - Cooper, Sections 13.1-13.4.1.

- Next time: Register allocation via graph colouring.