

# Code Generation for Data Processing

## Lecture 8: Register Allocation

Alexis Engelke

Chair of Data Science and Engineering (I25)  
School of Computation, Information, and Technology  
Technical University of Munich

Winter 2024/25

# Register Allocation

- ▶ Map unlimited/virtual registers to limited/architectural registers
- ▶ Assign a register to every value
  - ▶ Outputs get a (new) register, input operands often require registers
- ▶ When running out of registers, move values to stack
  - ▶ Stack *spilling* – save value register from to stack memory
- ▶  $\phi$ -nodes: ensure all inputs are assigned to same location
- ▶ Goal: produce correct code, minimize extra load/stores
  - ▶ Regalloc affects performance in orders of magnitude

# Register Allocation: Overview Example

```
gauss(%0) {  
  %2 = SUBXri %0, 1  
  %3 = MADDXrrr %0, %2, 0  
  %4 = MOVXconst 2  
  %5 = SDIVrr %3, %4  
  ret %5  
}
```

```
gauss(%0 : X0) {  
  %2 = SUBXri %0, 1 : X1  
  %3 = MADDXrrr %0, %2, 0 : X0  
  %4 = MOVXconst 2 : X1  
  %5 = SDIVrr %3, %4 : X0  
  ret %5  
}
```

- ▶ May also insert copy and stack spilling instructions

## Simplest thing that could possibly work

- ▶ Idea: allocate a one stack slot for every SSA variable/argument
  - ▶ Load all instruction operands into registers right before
  - ▶ Perform instruction
  - ▶ Write result back to stack slot for that SSA variable
- + Simple, always works, debugging easy
- *Extremely* inefficient in time and space

# Regalloc Example 1

```
gauss(%0)
```

```
  %2 = SUBXri %0, 1
```

```
  %3 = MADDXrrr %0, %2, 0
```

```
  %4 = MOVXconst 2
```

```
  %5 = SDIVrr %3, %4
```

```
ret %5
```

```
gauss(%0 : X0)
```

```
  %spills = alloca 40
```

```
  STRXi %0, %spills, 0
```

```
  %10 = LDRXi %spills, 0 : X0
```

```
  %2 = SUBXri %0%10, 1 : X0
```

```
  STRXi %2, %spills, 8
```

```
  %11 = LDRXi %spills, 0 : X0
```

```
  %12 = LDRXi %spills, 8 : X1
```

```
  %3 = MADDXrrr %11, %12, 0 : X0
```

```
  STRXi %3, %spills, 16
```

```
  %4 = MOVXconst 2 : X0
```

```
  STRXi %4,i %spills, 24
```

```
  %13 = LDRXi %spills, 16 : X0
```

```
  %14 = LDRXi %spills, 24 : X1
```

```
  %5 = SDIVrr %13, %14 : X0
```

```
  STRXi %5, %spills, 32
```

```
  %15 = LDRXi %spills, 32 : X0
```

```
ret %15
```

# Handling PHI Nodes

- ▶  $\phi$ -node needs to become register or stack slot
  - ▶ Simplest thing that could possibly work: PHI becomes stack slot
- ▶ Remember:  $\phi$ -nodes are executed on the edge
- ▶ Idea: predecessors write their value to that location at the end
  - ▶ First pass: define/allocate storage for  $\phi$ -node, but ignore inputs
  - ▶ Second pass: insert move operations at end of predecessors

## Regalloc Example 2

```
identity(%0)
  br %2
2:
  %3 = phi [ 0, %1 ], [ %4, %2 ]
  %4 = ADDXri %3, 1
  %5 = CMPXrr_BLS %4, %0
  br %5, %2, %6
6:
  ret %3
```

---

Pass 12

► Original value lost in %6!

```
identity(%0 : X0)
  %spills = alloca 24
  STRXi %0, %spills, 0
  %c0 = MOVXconst 0 : X0
  STRXi %c, %spills, 8
  br %2
2: %3 = phi [ 0, %1 ], [ %4, %2 ]
  %10 = LDRXi %spills, 8 : X0
  %4 = ADDXri %10, 1 : X0
  STRXi %4, %spills, 16
  %14 = LDRXi %spills, 16 : X0
  STRXi %14, %spills, 8
  %11 = LDRXi %spills, 16 : X0
  %12 = LDRXi %spills, 0 : X1
  %5 = CMPXrr_BLS %11, %12
  br %5, %2, %6
6: %13 = LDRXi %spills, 8 : X0
  ret %13
```

# Critical Edges

- ▶ Critical edge: edge from block with mult. succs. to block with mult. preds.
- ▶ Problem: cannot place move on such edges
  - ▶ When placing in predecessor, they would also execute for other successor  
⇒ unnecessary and – worse – incorrect



- ▶ *Break* critical edges: insert an empty block



## Regalloc Example 2 – Attempt 2

```
identity(%0)
  br %2
2:
  %3 = phi [ 0, %1 ], [ %4, %6 ]
  %4 = ADDXri %3, 1
  %5 = CMPXrr_BLS %4, %0
  br %5, %6, %7
6:
  br %2
7:
  ret %3
```

---

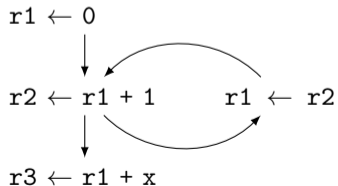
Pass 12

```
identity(%0 : X0)
  %spills = alloca 24
  STRXi %0, %spills, 0
  %c0 = MOVXconst 0 : X0
  STRXi %c, %spills, 8
  br %2
2:%3 = phi [ 0, %1 ], [ %4, %6 ]
  %10 = LDRXi %spills, 8 : X0
  %4 = ADDXri %10, 1 : X0
  STRXi %4, %spills, 16
  %11 = LDRXi %spills, 16 : X0
  %12 = LDRXi %spills, 0 : X1
  %5 = CMPXrr_BLS %11, %12
  br %5, %6, %7
6:%14 = LDRXi %spills, 16 : X0
  STRXi %14, %spills, 8
  br %2
7:%13 = LDRXi %spills, 8 : X0
  ret %13
```

# Handling Critical Edges

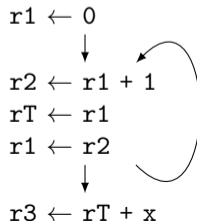
## Breaking Edges

- ▶ Insert new block for moves
- + Simple, no analyses needed
- Bad performance in loops



## Copy Used Values

- ▶ Move values still used to new reg.
- + Performance might be better
- Needs more registers



## Regalloc Example 3

```
odd(%0)
  br %2
2:
  %3 = phi [ %0, %1 ], [ %8, %7 ]
  %4 = phi [ 1, %1 ], [ %5, %7 ]
  %5 = phi [ 0, %1 ], [ %4, %7 ]
  %6 = CBNZX(%3)
  br %6, %7, %9
7:
  %8 = SUBXri %3, 1
  br %2
9:
  ret %4
```

► Value of  $\phi$  node lost!

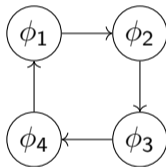
```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0
  %l3 = LDRXi %spills, 0 : X0; STRXi %l3, %spills, 8
  %c0 = MOVXconst 1 : X0; STRXi %c0, %spills, 16
  %c1 = MOVXconst 0 : X0; STRXi %c1, %spills, 16
  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %l0 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%l0)
  br %6, %7, %9
7:%l1 = LDRXi %spills, 8 : X0
  %8 = SUBXri %l2, 1 : X0; STRXi %8, %spills, 32
  %l4 = LDRXi %spills, 40 : X0; STRXi %l4, %spills, 8
  %l5 = LDRXi %spills, 24 : X0; STRXi %l5, %spills, 16
  %l6 = LDRXi %spills, 16 : X0; STRXi %l6, %spills, 24
  br %2
9:%l2 = LDRXi %spills, 24 : X0
  ret %l2
```

# PHI Cycles

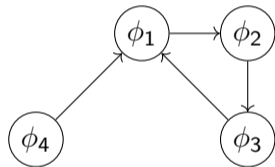
- ▶ Problem:  $\phi$ -nodes can depend on each other
- ▶ Can be chains (ordering matters) or cycles (need to be broken)
- ▶ Note: only  $\phi$ -nodes defined in same block are relevant/problematic



$$\begin{aligned}\phi_1 &= \phi(\phi_2, \dots) \\ \phi_2 &= \phi(\phi_3, \dots) \\ \phi_3 &= \phi(v, \dots)\end{aligned}$$



$$\begin{aligned}\phi_1 &= \phi(\phi_2, \dots) \\ \phi_2 &= \phi(\phi_3, \dots) \\ \phi_3 &= \phi(\phi_4, \dots) \\ \phi_4 &= \phi(\phi_1, \dots)\end{aligned}$$



$$\begin{aligned}\phi_1 &= \phi(\phi_2, \dots) \\ \phi_2 &= \phi(\phi_3, \dots) \\ \phi_3 &= \phi(\phi_1, \dots) \\ \phi_4 &= \phi(\phi_1, \dots)\end{aligned}$$

# Handling PHI Cycles

1. Compute number of other  $\phi$  nodes reading other  $\phi$  on same edge
2. For each  $\phi$  with 0 readers: handle node/chain
  - ▶ No readers  $\rightsquigarrow$  start of chain
  - ▶ Handling node may unblock next element in chain
3. For all remaining  $\phi$ -nodes: must be cycles, reader count always 1
  - ▶ Choose arbitrary node, load to temporary register, unblock value
  - ▶ Handle just-created chain
  - ▶ Write temporary register to target

---

Resolving  $\phi$  cycles requires an extra register (or stack slot)

## Regalloc Example 3 – Attempt 2

Edge %1 → %2 Edge %7 → %2

Critical  $\phi$ :

- ▶ %4 #readers: 10 – broken
- ▶ %5 #readers: 10

Action: break %4

```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0
  %13 = LDRXi %spills, 0 : X0; STRXi %13, %spills, 8
  %c0 = MOVXconst 1 : X0; STRXi %c0, %spills, 16
  %c1 = MOVXconst 0 : X0; STRXi %c1, %spills, 16
  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %10 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%10)
  br %6, %7, %9
7:%11 = LDRXi %spills, 8 : X0
  %8 = SUBXri %12, 1 : X0; STRXi %8, %spills, 32
  %14 = LDRXi %spills, 40 : X0; STRXi %14, %spills, 8
  %15 = LDRXi %spills, 24 : X1
  %16 = LDRXi %spills, 16 : X0; STRXi %16, %spills, 24
  STRXi %15, %spills, 16
  br %2
9:%12 = LDRXi %spills, 24 : X0
  ret %12
```

# SSA Destruction

```
fn(%0, %1) {  
  b1:  
    %2 = add %0, %1  
    br %b2  
  b2:  
    %3 = phi [%b1: %1], [%b3: %4]  
    %4 = phi [%b1: %0], [%b3: %3]  
    %5 = phi [%b1: %2], [%b3: %3]  
    %6 = phi [%b1: 0], [%b3: %8]  
    %7 = icmp lt %3, %6  
    br %7, %b3, %b4  
  b3:  
    %8 = add %6, 1  
    %9 = icmp gt %8, %1  
    br %9, %b4, %b2  
  b4:  
    %10 = phi [%b2: %4], [%b3: %3]  
    %11 = phi [%b2: %5], [%b3: %8]  
    %12 = add %10, %11  
    ret %12  
}
```

1. Dependencies between  $\phi$ -nodes?
2. Critical Edges? (Draw CFG)
3. Destruct SSA into form with unlimited registers.
  - 3.1 ... by breaking critical edges
  - 3.2 ... by copying used values

# Better Register Allocation

- ▶ Goal: keep as many values in registers as possible
  - ▶ Less stack spilling  $\Rightarrow$  better performance
- ▶ Problem: register count (severely) limited
- ↪ Are there enough registers? (otherwise: spilling)
- ↪ Which register to choose?
- ↪ Which register to kill and put on the stack?
- ▶ Needs information when value is actually needed



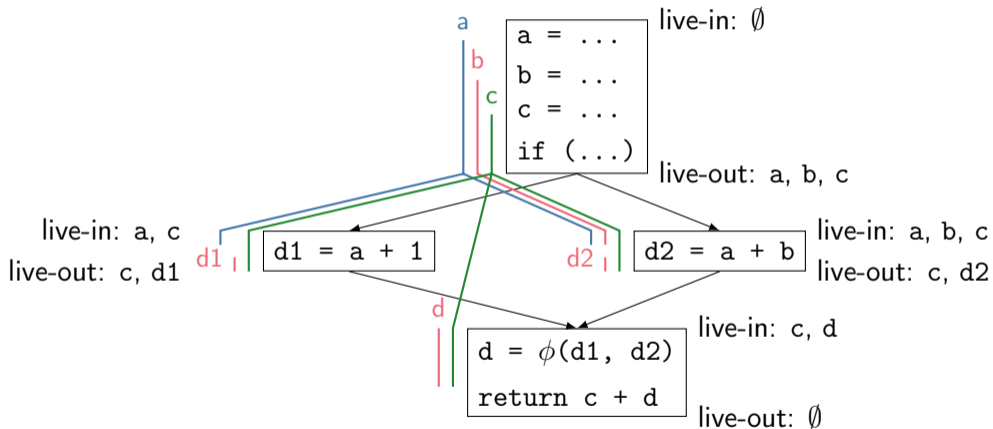
## Interlude: Register Allocation Research – Executive Summary

- ▶ *Tons* of papers exist
- ▶ Papers often skip over important details
  - ▶ E.g., when spilling – using the value needs another register
  - ▶ E.g., temporary register for shuffling values
- ▶ Additional (ISA) constraints in practice: (incomplete list)
  - ▶ 2-address instructions with destructive source
  - ▶ Fixed registers for specific instructions
  - ▶ Computing the stack address may need yet another register
  - ▶ Different register classes, often just handled independently
- ▶ Implementations even of simple algorithms tend to be large and complex

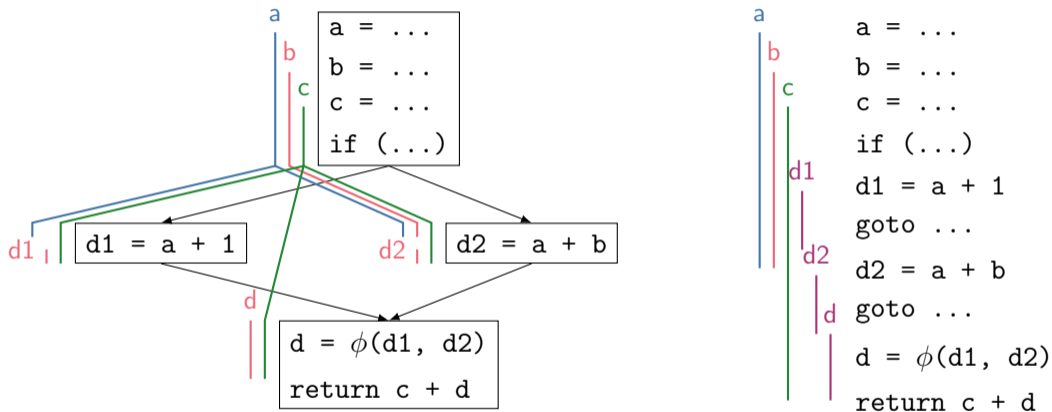
# Liveness Analysis – Definitions

- ▶ *Live*: value still used afterwards
  - ▶ After last (possible) use in program flow, the value becomes dead
- ▶ *Live ranges*: set of ranges in program where value is live
  - ▶ Not necessarily contiguous, e.g. in case of branches
- ▶ *Live interval*: over-approximation of live ranges without holes
  - ▶ Depends on block order, reverse post-order often a good choice
- ▶ *Live-in/Live-out*: values live at begin/end of basic block
  - ▶ For  $\phi$  nodes:  $\phi$  is live-in, operands are live-out in predecessors  
(Note: different literature uses different definitions)

# Liveness Analysis – Example



# Liveness Analysis – Example – Live Ranges vs. Live Intervals



- ▶ Live intervals are substantially worse, but easier to compute

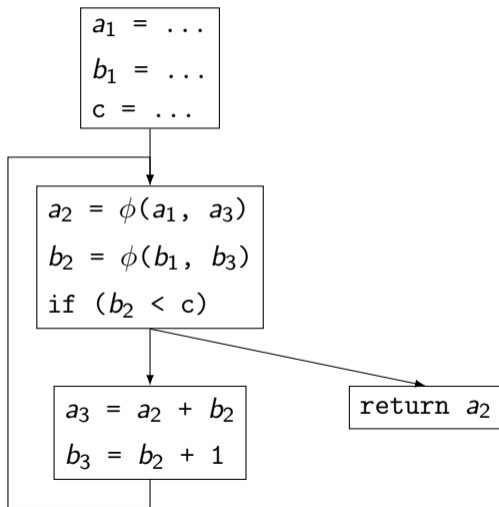
# Liveness Analysis – Algorithm<sup>39</sup>

- ▶ Iterate over blocks in post-order
  - ▶  $live \leftarrow \cup s.liveIn \setminus s.phis, s \in b.successors$
  - ▶  $live \leftarrow live \cup \{\phi.input(b) \mid \phi \in b.successors.phis\}$
  - ▶  $b.liveOut \leftarrow live$
  - ▶  $\forall v \in live : ranges[v].add(b.start, b.end)$
  - ▶ For each non- $\phi$  instruction  $inst$  in reverse order
    - ▶  $live \leftarrow (live \cup inst.ops) \setminus \{inst\}$
    - ▶  $ranges[inst].setStart(inst)$
    - ▶  $\forall op \in inst.ops : ranges[op].add(b.start, inst)$
  - ▶  $b.liveIn \leftarrow live \cup b.phis$
- ▶ Repeat until convergence<sup>38</sup>

<sup>38</sup>Reducible graphs: expanding  $liveIn$  of loop headers to the entire loop suffices

<sup>39</sup>Adapted from C Wimmer and M Franz. “Linear scan register allocation on SSA form”. In: *CGO*. 2010, pp. 170–179.

## Liveness Analysis – Example



# Liveness Analysis – Example

```
fn(%0, %1) {  
  b1:  
    %2 = add %0, %1  
    br %b2  
  b2:  
    %3 = phi [%b1: %1], [%b3: %4]  
    %4 = phi [%b1: %0], [%b3: %3]  
    %5 = phi [%b1: %2], [%b3: %3]  
    %6 = phi [%b1: 0], [%b3: %8]  
    %7 = icmp lt %3, %6  
    br %7, %b3, %b4  
  b3:  
    %8 = add %6, 1  
    %9 = icmp gt %8, %1  
    br %9, %b4, %b2  
  b4:  
    %10 = phi [%b2: %4], [%b3: %3]  
    %11 = phi [%b2: %5], [%b3: %8]  
    %12 = add %10, %11  
    %13 = add %12, %2  
    ret %13  
}
```

1. Compute live ranges of every SSA value

# Register Allocation Decisions (Outline)

- ▶ Question: are there enough registers for all values?
  - ▶ *Register pressure* = number of values live at some point
  - ▶ Register pressure  $>$  #registers  $\Rightarrow$  move some values to stack (spilling)
- ▶ Question: when spilling, which values and where to store/reload?
  - ▶ Spilling is expensive, so avoid spilling frequently used values
- ▶ Question: for unspilled values, which register to assign?
  - ▶ Also: respect register constraints, etc.



# Register Allocation Strategies

## Scan-based

- ▶ Iterate over the program
  - ▶ Decide locally what to do
  - ▶ Greedily assign registers
- 
- + Fast, good for straight code
  - Code quality often bad
  - ▶ Used for -O0 and JIT comp.

## Graph-based


- ▶ Compute *interference graph*
    - ▶ Nodes are values
    - ▶ Edge  $\Rightarrow$  live ranges overlap
  - ▶ Holistic approach
- 
- + Often generate good code
  - Expensive, superlinear run-time
  - ▶ Used for optimized code

# Linear Scan Register Allocation<sup>40</sup>

- ▶ Idea: treat whole function as single block
  - ▶ Block order affects quality (but not correctness)
  - ▶ Only consider live intervals without holes
- ▶ Iterate over instructions from top to bottom
- ▶ For operands of instruction in their last use: mark register as free
- ▶ Assign instruction result to new free register
  - ▶ If no free register available: move some value to the stack
  - ▶ Heuristic: value whose liveness ends furthest in future

# Linear Scan Register Allocation

- + low compile-time, simple
- very suboptimal code, live intervals grossly over-approximated
- ▶ What's missing?
  - ▶ Registers to load spilled values
  - ▶ Shuffling of values between blocks
  - ▶ Register constraints (e.g., for instructions or function calls)
- ▶ Other disadvantage: once a value is spilled, it is spilled everywhere
  - ▶ Some other approaches based on lifetime splitting<sup>41</sup>
- ▶ Function calls: clobber lots of registers

<sup>41</sup>O Traub, G Holloway, and MD Smith. "Quality and speed in linear-scan register allocation". In: *SIGPLAN* 33.5 (1998), pp. 142–151. 

# Scan-based Register Allocation<sup>44</sup>

Iterate over basic blocks<sup>42</sup>

- ▶ Start with register assignment from predecessor
  - ▶ Multiple predecessors: choose assignment from any one
  - ▶  $\phi$ -nodes can either reside in registers or on the stack
- ▶ Iterate over instructions top-down
  - ▶ Ensure all instruction operands are in registers
    - ▶ When out of registers: move any value to stack
  - ▶ For operands in their last use: mark register as free
  - ▶ Assign instruction result to new free register
- ▶ Shuffle values back into registers where successor expects them<sup>43</sup>

<sup>42</sup>Typically: reverse post-order, so most predecessors are seen before successors, except for loops.

<sup>43</sup>Without critical edges, only relevant for blocks with one successor — others are visited afterwards by RPO definition.

<sup>44</sup>Mostly following Go: <https://github.com/golang/go/blob/5f7abe/src/cmd/compile/internal/ssa/regalloc.go>

# Scan-based Register Allocation – Spilling

What to spill?

- ▶ Spill value with furthest use in future<sup>45</sup>
  - ▶ Frees register for longest time
  - ▶ Requires information on next use to be stored during analysis
  - ▶ But: avoid spilling values computed inside loops (esp. loop-carried dependencies), reloads are fine<sup>46</sup>
  - ▶ Downside: superlinear run-time

Where to store?

- ▶ Stack, period.
- ▶ Spilling to FP/vector registers. . . occasionally proposed, not used in practice

<sup>45</sup>C Wimmer and H Mössenböck. “Optimized interval splitting in a linear scan register allocator”. In: *VEE*. 2005, pp. 132–141.

<sup>46</sup>Intel Optimization Reference Manual (Aug. 2023), Assembly/Compiler Coding Rules 38 and 45

# Scan-based Register Allocation – Spilling

Where to insert store?

- ▶ Option 1: spill exactly where required
  - ▶ Downside: multiple spills of same value, many reloads
- ▶ Option 2: spill once, immediately after computation
  - ▶ Later “spills” to the stack are less costly
  - ▶ May lead to spills on code paths that don’t need it
- ▶ Option 3: compute best place using dominator tree
  - ▶ Spill store must dominate all subsequent loads

# Scan-based Register Allocation – Register Assignment

- ▶ Merge blocks: choose predecessor with most values in registers
  - ▶ High likelihood of reducing the number of stores
  - ▶ Re-loads are pushed into predecessors
- ▶ Propagate register constraints bottom-up as hints first
  - ▶ E.g.: call parameters, instruction constraints, assignment for merge block
  - ▶ Reduces number of moves

# Graph Coloring Approaches

- + Considerably better results than greedy algorithms
- High run-time, even with heuristics
- ▶ Graph coloring in general is  $\mathcal{NP}$ -complete
- ▶ Often used in compilers (e.g., GCC, WebKit)

AD IN2053 “Program Optimization” covers this more formally



# Stack Frame Allocation

- ▶ Optionally setup frame pointer
  - ▶ Required for variably-sized stack frame  
Otherwise: cannot access spilled variables or stack parameters
- ▶ Optionally re-align stack pointer
- ▶ Save callee-saved registers, maybe also link register
- ▶ Optionally add code for stack canary
- ▶ Compute stack frame size and adjust stack pointer
  - ▶ Mainly size of `alloca`s, but needs to respect alignment
  - ▶ Ensure sufficient space for parameters passed on the stack
  - ▶ Ensure stack pointer is sufficiently aligned
- ▶ Stack pointer adjustment *may* be omitted for leaf functions
  - ▶ Some ABIs guarantee a *red zone*

# Block Ordering

- ▶ Order blocks to make use of fall-through in machine code
- ▶ Avoid sequences of `b.cond; b`
  - ▶ Sometimes cannot be avoided: conditional branches often have shorter range
- ▶ Block ordering has implications for branch prediction
  - ▶ Forward branches default to not-taken, backward taken
  - ▶ Unlikely blocks placed “out of the way” of the main execution path
  - ▶ Indirect branches are predicted as fall-through

# Register Allocation – Summary

- ▶ Map unlimited virtual registers to restricted register set
- ▶ Responsible for:
  - ▶ Assigning registers to values
  - ▶ Deciding which registers to spill to stack
  - ▶ Deciding when to spill/unspill values
- ▶  $\phi$ -nodes require extra care, esp. for chains and cycles
- ▶ Liveness information is key information for register allocation
- ▶ Scan-based approaches are fast, but lead to suboptimal code
- ▶ Graph coloring yields better results, but is much slower
- ▶ Register allocation/spilling heavily relies on heuristics in practice

# Register Allocation – Questions

- ▶ Why is register allocation a difficult problem?
- ▶ How are  $\phi$ -nodes handled during register allocation?
- ▶ What are the two main problems when destructing  $\phi$ -nodes?
- ▶ Why are critical edges problematic and how to deal with them?
- ▶ What are practical constraints for register allocation?
- ▶ How to detect whether a value is still needed at some point?
- ▶ How to compute the live ranges of values in an SSA-based IR?
- ▶ What is the idea of linear scan and what are its practical problems?