Compilers 101

Backends - Part 1

Previously…

- Preprocessing
- Lexical analysis
- Syntax analysis
- Semantic analysis
- IR Generation
- IR Optimization
- Optimization
- Code generation

Frontend

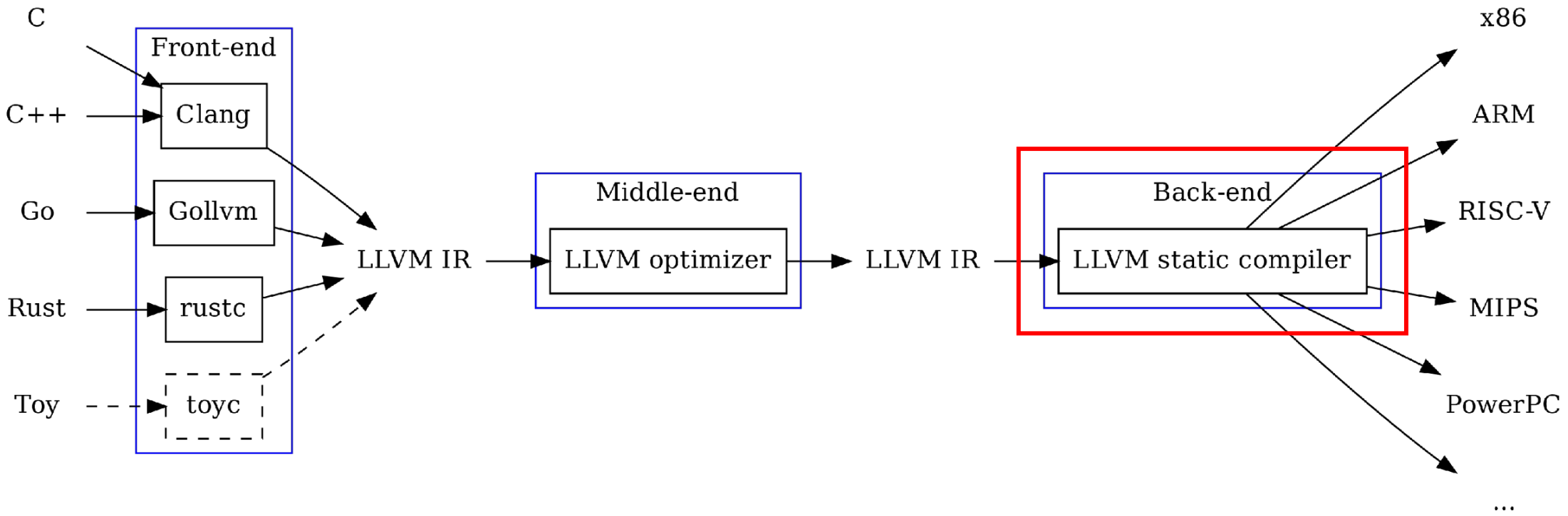
Middle-end

Backend

Today

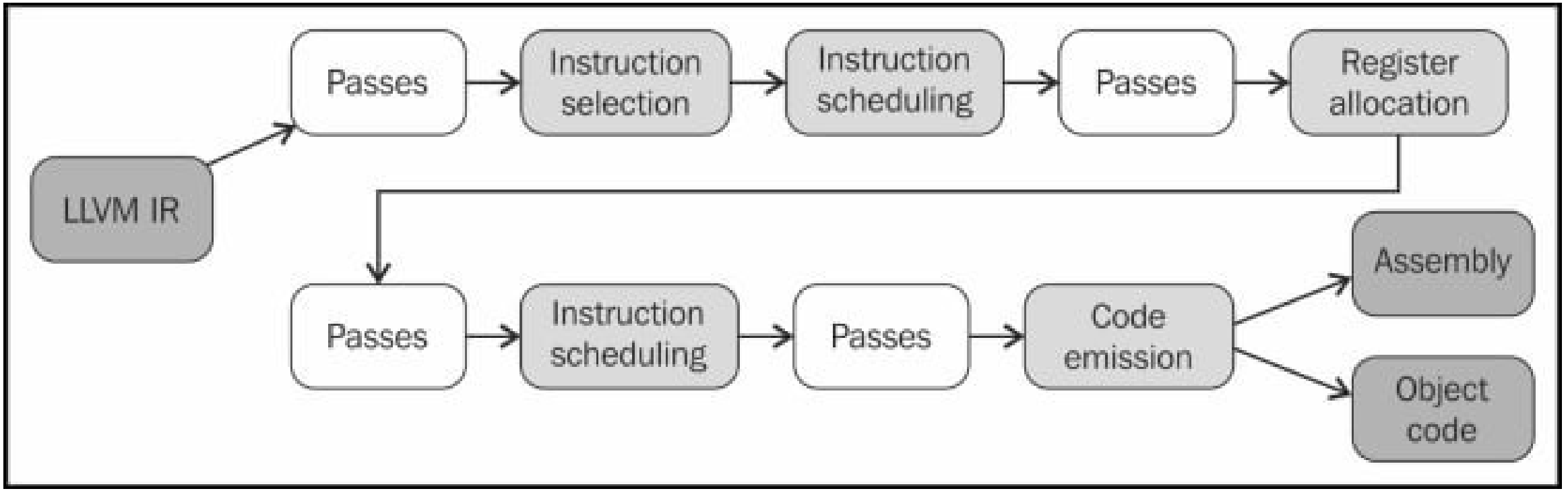
- Compiler backends
- Backend stages and algorithms behind

Backend in LLVM



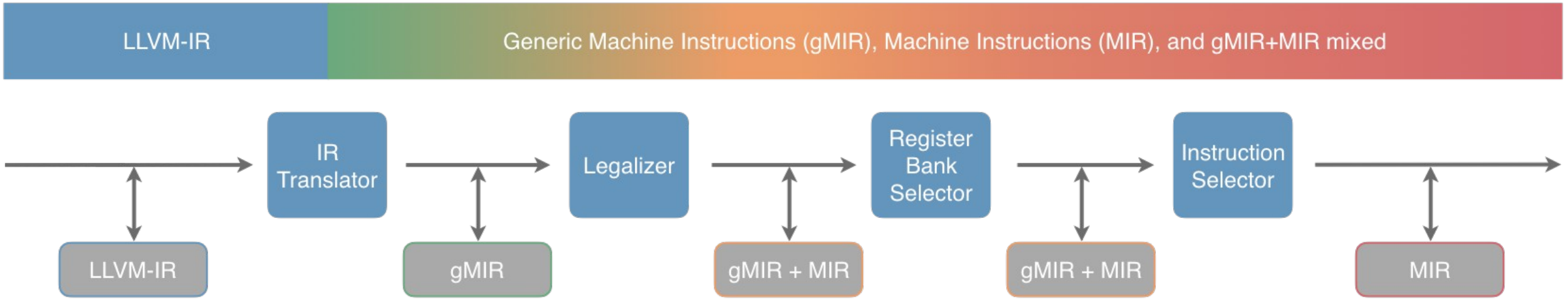
https://wr.informatik.uni-hamburg.de/_media/teaching/wintersemester_2020_2021/ep-2021_hosseini_llvm.pdf

LLVM backend pipeline (overview)



https://getting-started-with-llvm-core-libraries-zh-cn.readthedocs.io/zh-cn/latest/ch06.html

Modern LLVM backend: detailed view



https://llvm.org/docs/GlobalISel/Pipeline.html

Details about LLVM backend are going to be covered next time…

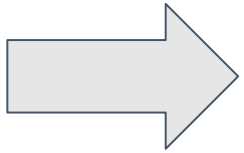
Instruction selection

- We now have an optimized IR code, but it's not consumable by HW
- Instruction selection is a stage of compiler, that transforms IR into low-level IR

In case of LLVM compiler: LLVM IR → MIR (Machine IR)

LLVM IR

```
source_filename = "factorial_module.ll"

define i32 @factorial(i32 %n) {
entry:
  %cmp = icmp sle i32 %n, 1
  br i1 %cmp, label %base, label %recursive

base:
  ret i32 1

recursive:
  %n_minus_one = sub i32 %n, 1
  %rec_call = call i32 @factorial(i32 %n_minus_one)
  %result = mul i32 %n, %rec_call
  ret i32 %result
}
```

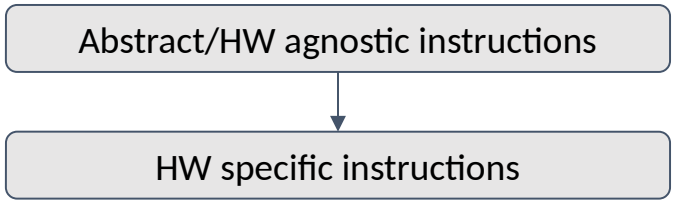


MIR

```
body:
  bb.0.entry:
    successors: %bb.1(0x40000000), %bb.2(0x40000000)
    liveins: $edi, $rbx

    CMP32ri $edi, 1, implicit-def $eflags
    JCC_1 %bb.2, 15, implicit killed $eflags

  bb.1.base:
    liveins: $rbx

    $eax = MOV32ri 1
    RET64 $eax

  bb.2.recursive:
    liveins: $edi, $rbx

    frame-setup PUSH64r killed $rbx, implicit-def $rsp, implicit $rsp
    frame-setup CFI_INSTRUCTION def_cfa_offset 16
    CFI_INSTRUCTION offset $rbx, -16
    $ebx = MOV32rr $edi, implicit-def $rbx
    renamable $edi = LEA64_32r renamable $rbx, 1, $noreg, -1, $noreg
    CALL64pcrel32 target-flags(x86-plt) @factorial, csr_64, implicit $rsp, implic
    renamable $eax = IMUL32rr killed renamable $eax, renamable $ebx, implicit-def
    $rbx = frame-destroy POP64r implicit-def $rsp, implicit $rsp
    frame-destroy CFI_INSTRUCTION def_cfa_offset 8
    RET64 $eax
...
```

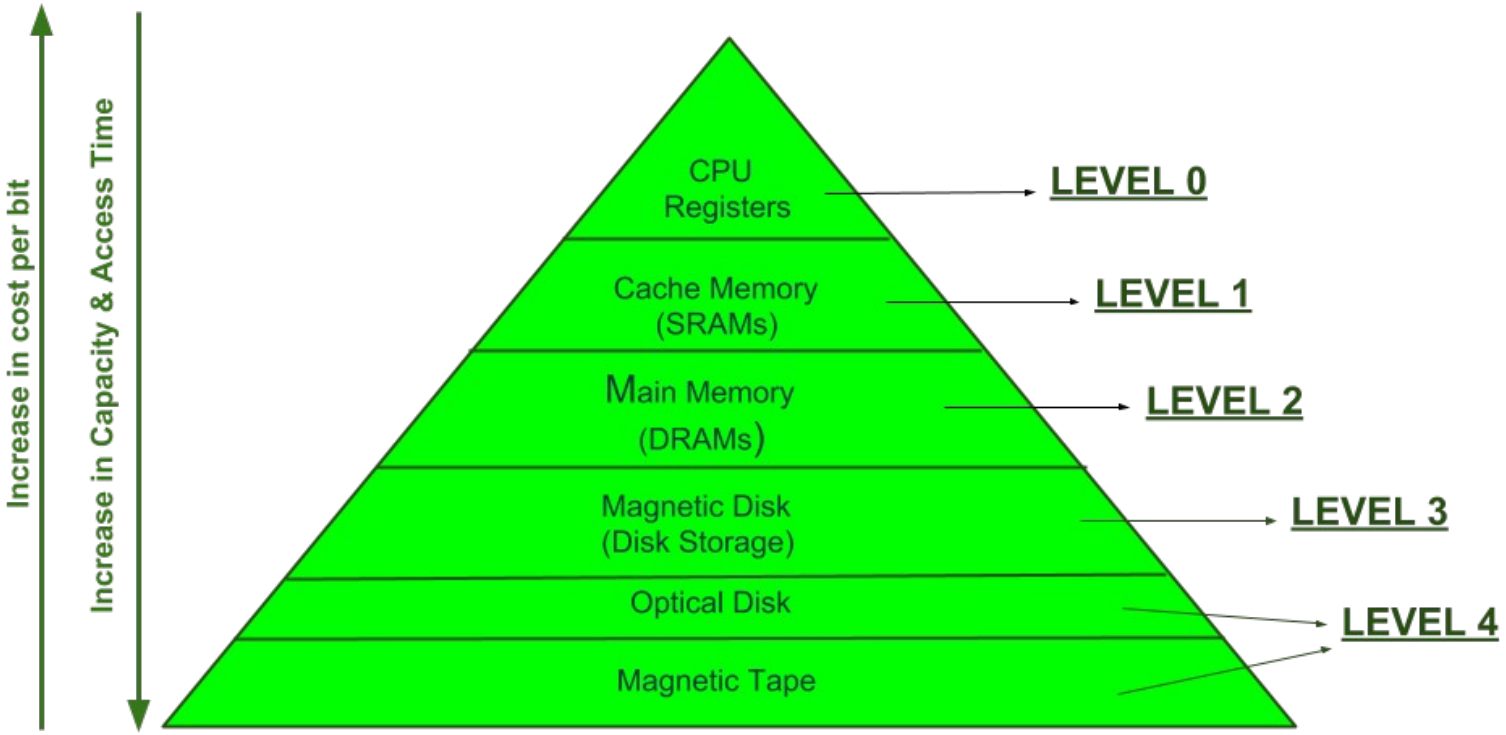
https://godbolt.org/z/d33c1P99M

Instruction selection methods



- Macro expansion
  - Replace all IR instructions by matching templates
- Graph covering
  - Transform IR into graph
  - Cover graph with patterns – templates that match a portion of a graph
- Lowest common denominator strategy
  - Attempt to select instructions that would allow execution on the widest range of hardware

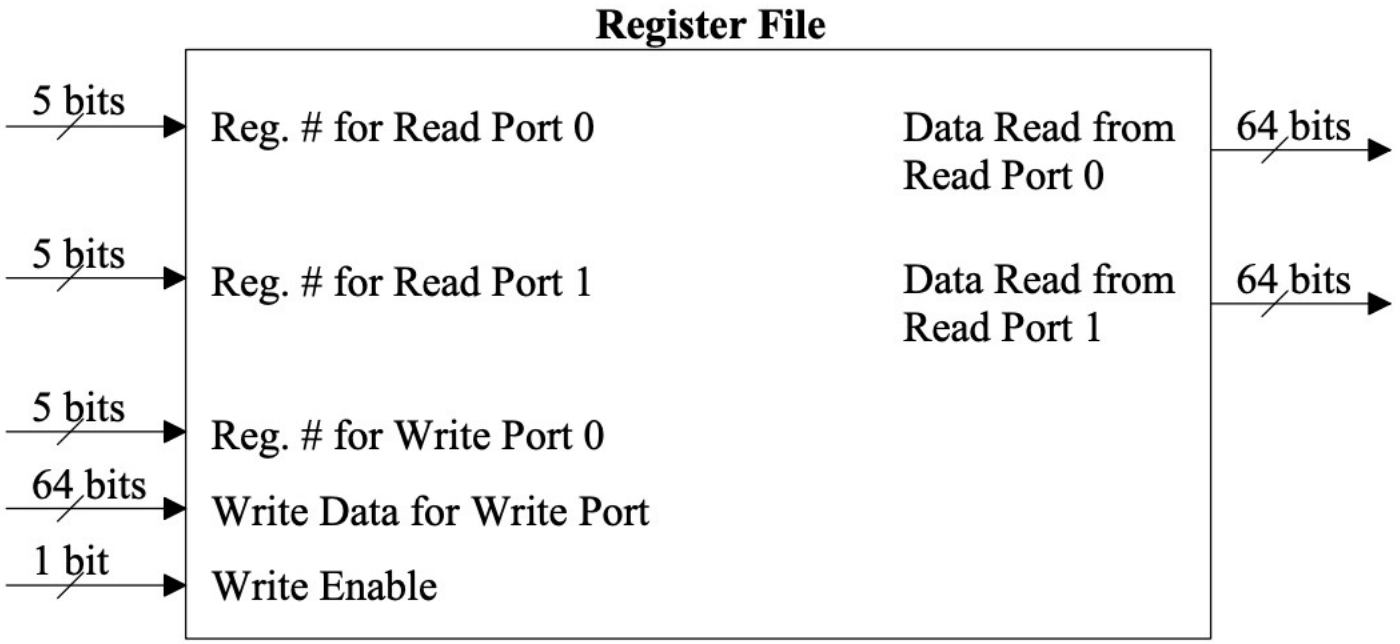
Memory hierarchy



MEMORY HIERARCHY DESIGN

https://www.geeksforgeeks.org/memory-hierarchy-design-and-its-characteristics/

Registers



https://www.cs.uni.edu/~fienup/cs041s08/lectures/lec13_reg_file.pdf

Registers in modern CPUs

Different number of scalar registers in the most common architectures

Architecture	32 bits	64 bits
ARM	15	31
Intel x86	8	16
MIPS	32	32
POWER/PowerPC	32	32
RISC-V	16/32	32
SPARC	31	31

https://en.wikipedia.org/wiki/Register_allocation

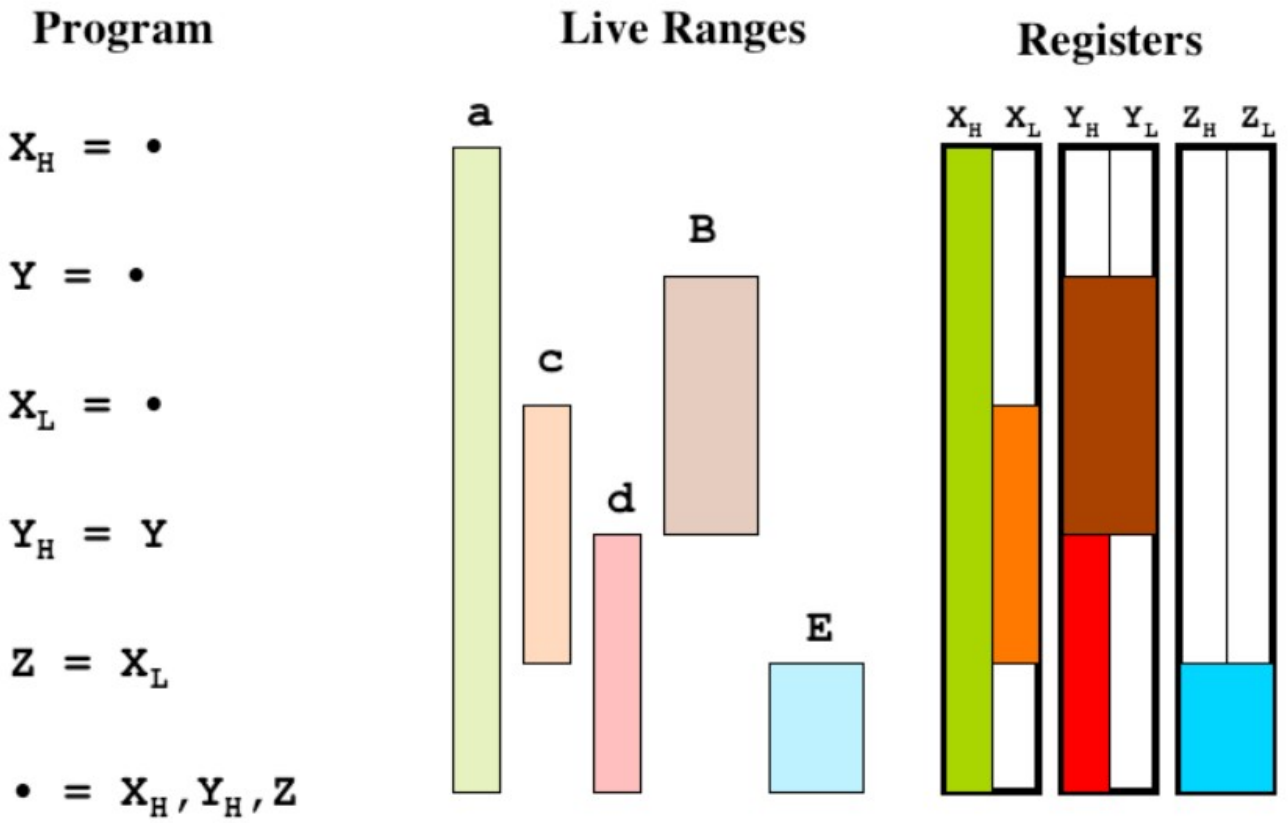
SSA recap

- Registers are only assigned once
- Infinite amount of registers

Challenges

- Registers are scarce
  - In most of the cases, we deal with dozens of registers
- Registers are complicated
  - Registers can be made of smaller registers
  - Some registers may be reserved
  - Some instructions must store results to certain registers
  - Some registers are part of ABI

Challenges



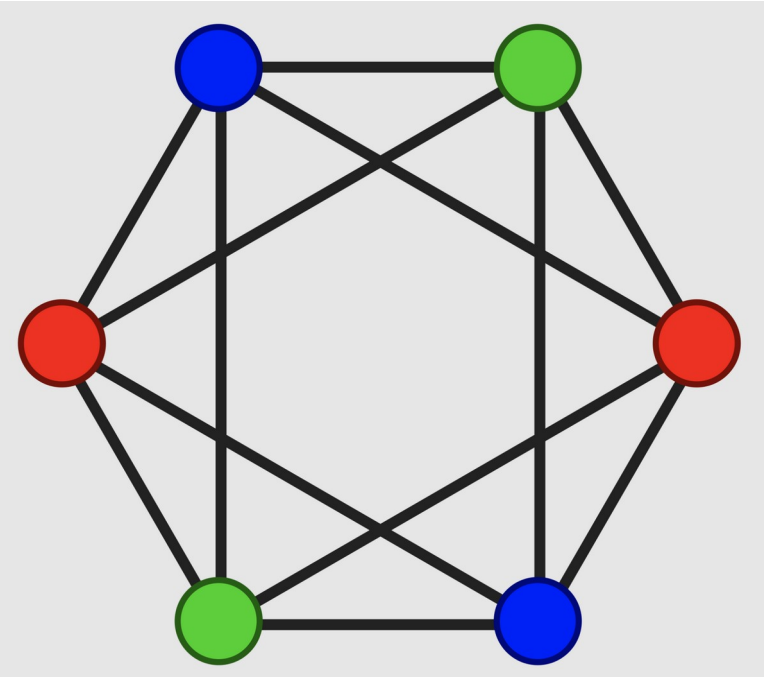
https://llvm.org/pubs/2008-06-PLDI-PuzzleSolving.pdf

Graph theory to the rescue!

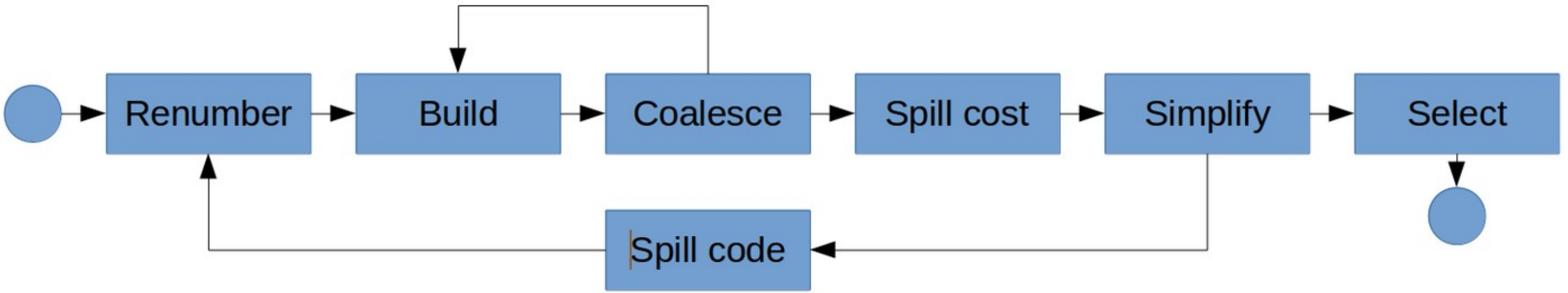
- Each register is a node in a graph
- Edges are interfering ranges (i.e., registers that live together)

Graph coloring

- Process vertices in the given order
- Assign colors to each vertex
- Use the smallest color number that is not already in use



Register allocation mechanism



Register allocation mechanism stages

- Renumber
  - assign/verify unique virtual register IDs
- Build
  - build a graph: liveness analysis
- Coalesce
  - merge nodes that represent copy-related variables (reduce need of COPY)
- Spill Cost
  - calculate a spill cost for each variable
- Simplify
  - simplify a graph (e.g., drop low-degree nodes and push them to the stack)
- Select
  - compiler pops nodes from the stack and tries to assign them to registers (color them)
  - if a node cannot be assigned a register (e.g., it conflicts with all available registers), the compiler designates it as “spilled”
- Insert spill code
  - compiler inserts extra load/store instructions (the spill code) to move that variable between memory and registers (if needed)

Pros and cons

- Good known algorithm

- NP-hard problem
- Evicted variables are spilled everywhere
- Variable, that is not spilled, is kept in the register throughout its whole lifetime

Linear scan to the rescue!

- Instead of building a graph, all the variables are linearly scanned to determine their live range
- All ranges are sorted and traversed chronologically.
- Registers are allocated in a greedy way

Pros and cons

- Fast algorithm

- Lifetime holes
- Spilled variable will stay spilled for its entire lifetime

CPU scheduler and pipeline

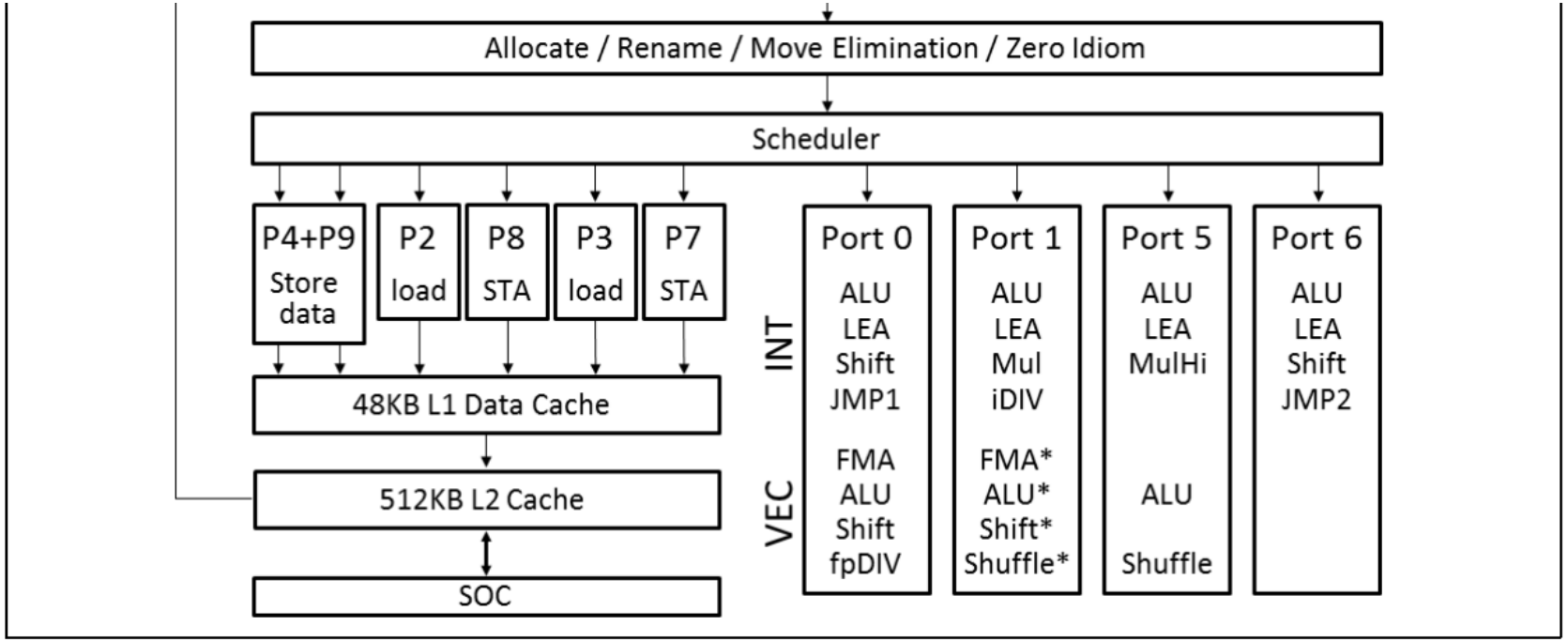


Figure 2-1. Processor Core Pipeline Functionality of the Ice Lake Client Microarchitecture[1]

https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

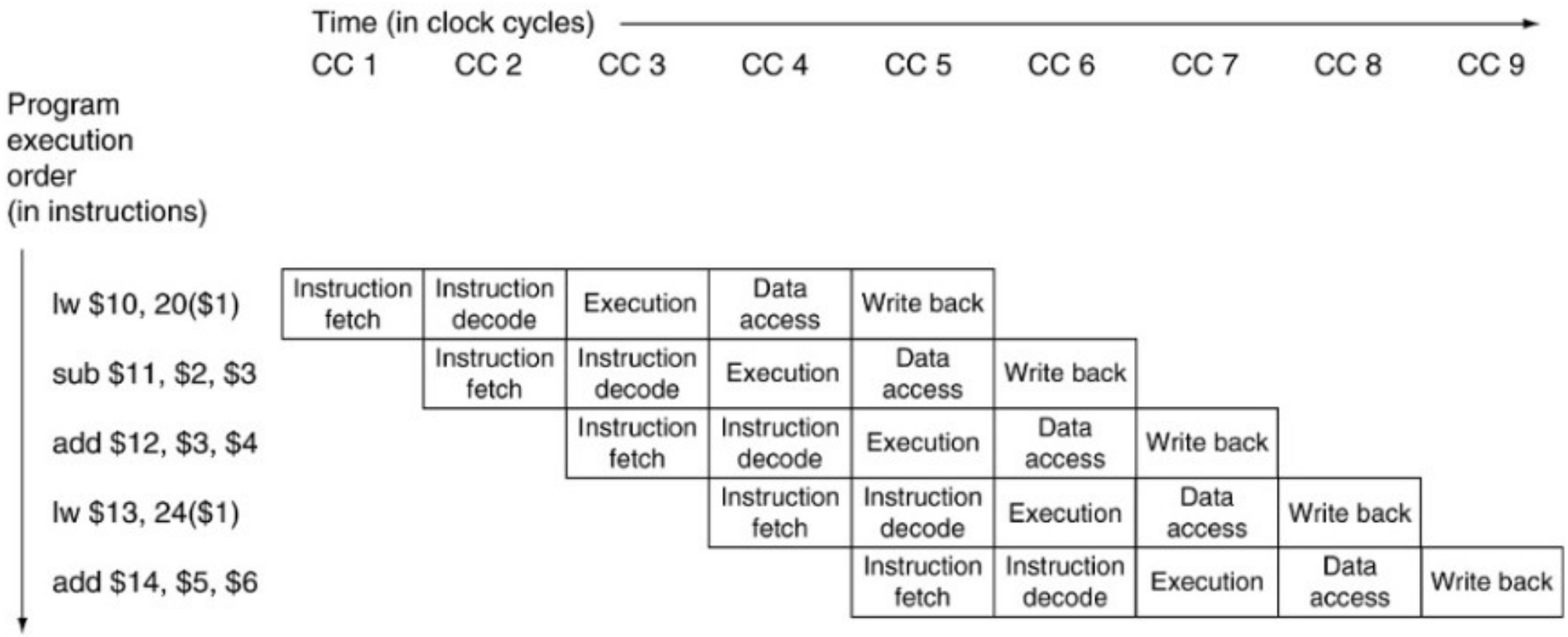
Instruction scheduling

- Instruction scheduling is a compiler optimization used to improve ILP
- Goals:
  - Avoid pipeline stalls
  - Avoid illegal or semantically ambiguous operations

Instruction scheduling types

- Static instruction scheduling:
  - during compile time
  - reordering instructions
  - compiler has less execution context
  - more time for instruction rearrangement at compile time
  - complicates compiler
- Dynamic instruction scheduling:
  - during execution time on a HW
  - Selectively issuing instructions to execution units
  - HW (e.g. CPU) has more execution context
  - less time for instruction rearrangement at execution time
  - complicates HW

Pipeline execution



Data hazards

- Read after Write
  - Instruction 1 writes a value used later by instruction 2. I1 must come first.
- Write after Read
  - Instruction 1 reads a location that is later overwritten by Instruction 2. Instruction 1 must come first, or it will read the new value instead of the old
- Write after Write
  - Two instructions both write the same location. They must occur in their original order

Scheduling algorithm inputs

Hardware details like:

- Micro-ops
- Latencies
- Resource cycles

Scheduling algorithm

- Build dependency graph
- Apply topology sort
- Use target-specific heuristics to schedule instructions

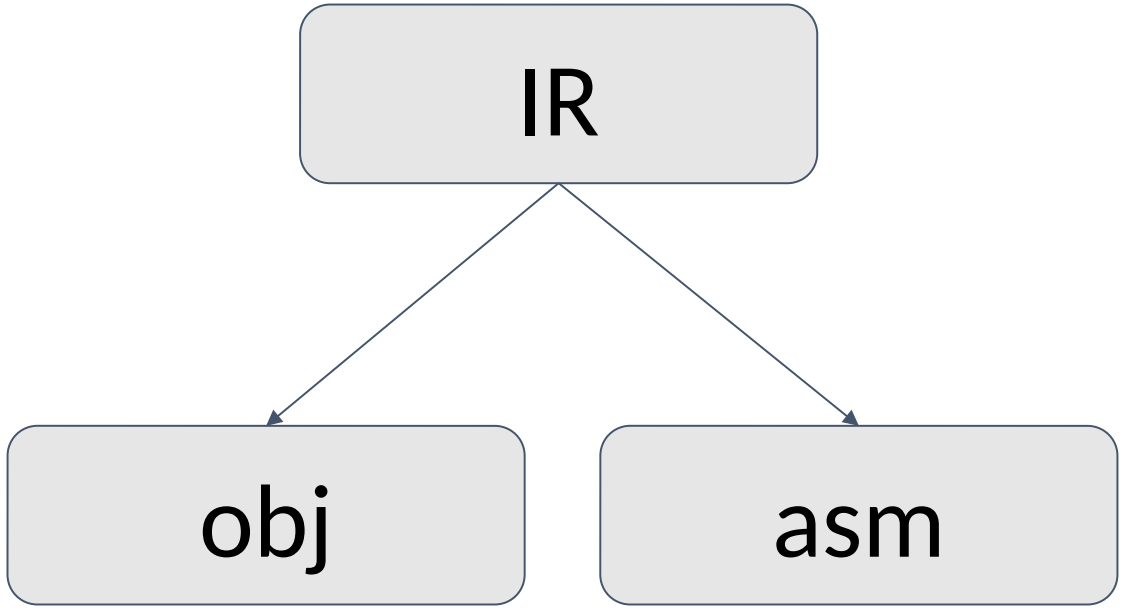
List scheduling algorithm

- Input: list of jobs that should be executed on a set of m machines

- Take first job in the list
- Find a machine that is available for executing job
  - If a machine is found, schedule the job
  - Otherwise, select the next job

Code emission

- asm emission
- object file generation




```
792415C0      55            push ebp
792415C1      89E5          mov ebp, esp
792415C3      8B45 08       mov eax, [ebp+0x08]
792415C6      DB28          fld tword [eax]
792415C8      8B4D 0C       mov ecx, [ebp+0x0C]
792415CB      DB29          fld tword [ecx]
792415CD      DEC1          faddp
792415CF      8B55 10       mov edx, [ebp+0x10]
792415D2      DB3A          fstp tword [edx]
792415D4      DB68 0A       fld tword [eax+0x0A]
792415D7      DB69 0A       fld tword [ecx+0x0A]
792415DA      DEC1          faddp
792415DC      DB7A 0A       fstp tword [edx+0x0A]
792415DF      5D            pop ebp
792415E0      C2 0C00       ret 0x000C
```

Target specific passes

- Instruction selection
- Register allocation
- Peephole optimizations
- Loop optimizations
- Vectorizer
- Machine Code optimizations
  - Fine-tunes machine code, including adjusting alignment, optimizing branch instructions, and applying target-specific tweaks to increase efficiency.
- Link-Time optimizations (LTO)
  - Performs optimizations across multiple compilation units or modules at the link stage, enabling more global optimizations that are not possible when compiling files separately
- Profile-Guided Optimization (PGO)
  - Uses runtime profiling information to guide optimization decisions, such as which branches to prioritize for speed or which loops to optimize for unrolling

Next time…

- LLVM backend overview

Test

https://forms.gle/LZZZYDejbnUV19zY7

Submission time: **10 minutes**

Какие проблемы решают алгоритмы распределения/аллокации регистров (register allocation)? В чем состоит сам алгоритм?

Your answer

Зачем используется алгоритм планирования/упорядочивания инструкций (instruction scheduling)?

Your answer

Backup: me@gooddoog.ru

Extra materials

- LLVM Developers' Meeting: J. Bogner & A. Nandakumar & D. Sanders "Tutorial: GlobalISel " - https://www.youtube.com/watch?v=Zh4R40ZyJ2k
- Tutorial: Creating an LLVM Backend for the Cpu0 Architecture - https://jonathan2251.github.io/lbd/
- https://llvm.org/devmtg/2014-04/PDFs/Talks/Building%20an%20LLVM%20backend.pdf
- Writing an LLVM backend: https://llvm.org/docs/WritingAnLLVMBackend.html
- Welcome to the Back End: The LLVM Machine Representation: https://llvm.org/devmtg/2017-10/slides/Braun-Welcome%20to%20the%20Back%20End.pdf