Machine Problem 4

CS 426 — Compiler Construction

Handed out: Nov. 17, 2025; due: Dec. 8, 2025, 11:59 pm

In this assignment, you will implement a register allocator for LLVM, using a greedy and local allocation algorithm. Your register allocator should be independent of the target ISA, which is possible using the APIs provided by LLVM.

The objective of this MP is to help you (1) understand register allocation better, and (2) have some concrete experience of implementing backend code generation.

1 Getting Started

1. Download the llvm-project-19.1.6.src.tar.xz source code from https://github.com/llvm/llvm-project/releases/download/llvmorg-19.1.6.

2. Copy MP 4 handout files from `/class_cs426/public/mp4_handout` on your virtual machine. You should see

   - `RegAllocSimple.cpp`: a boilerplate file for a register allocator.
   - `simple-regalloc-19.1.6.patch`: a patch to set up the new register allocator.
   - `tests/`: some basic test cases.

3. Extract the LLVM source code to a directory, say, `llvm19`; then set up the new register allocator by

```
tar xvf llvm-project-19.1.6.src.tar.xz && mv llvm-project-19.1.6.src llvm19
    && cd llvm19
# ...or replace "../mp4_handout" with where you put your handout
patch -p1 < ../mp4_handout/simple-regalloc-19.1.6.patch
cp ../mp4_handout/RegAllocSimple.cpp llvm/lib/CodeGen
```

4. Configure and build LLVM. This will take around 10GB of disk space, and around 20 minutes (on your virtual machine).

```
mkdir build
cmake -B build/ -S llvm/ \
  -DCMAKE_C_COMPILER="clang-19" -DCMAKE_CXX_COMPILER="clang++-19" -
    DLLVM_USE_LINKER=lld \
  -DCMAKE_BUILD_TYPE="Debug" -DLLVM_TARGETS_TO_BUILD="X86" -
    DLLVM_INCLUDE_BENCHMARKS=Off
make -j4 llc  # only llc is needed
```

5. You should be able to run `bin/llc -regalloc=simple` on some LLVM IR file, and `llc` should print the message `simple regalloc not implemented` before aborting. You may consider adding the `bin` directory under `build/` to your `PATH` variable.

2 A Local Register Allocation Algorithm

In this MP, you will modify the template file `RegAllocSimple.cpp` to implement a register allocator. The register allocation algorithm you will implement is a local (single-basic-block) algorithm with trivial spill costs. You can find the pseudocode of this algorithm in Section 6.

For an intuition of the algorithm, consider a trivial, spill-all algorithm that stores all virtual registers on the stack: before each instruction, we reload all the uses from the stack to some available physical registers, and after each instruction, we spill all virtual register definitions back to the stack. As a result, no virtual register is live at the program point between two instructions.

The algorithm that you will implement is similar, except at the level of basic blocks instead of individual instructions. In other words, you will reload all virtual register uses of a basic block from stack slots before they are used, and spill all live virtual registers at the end of a basic block. This reduces the problem to allocating registers locally in a basic block, which is then solved by a greedy, linear algorithm that allocates a virtual register to the next available physical register.

To reduce the complexity, you can make the following assumptions:

- Only general purpose registers are used.
- No inline assembly.
- You don't need to handle debug information in the program.
- We will only test your code on the x86 backend. But you are still strongly encouraged to keep your allocator target-independent – and it is in fact easier this way.
- No Phi instructions: the SSA form has already been destructed before register allocation in LLVM.

However, you should handle these cases:

- Subregisters: you should not assign two different physical registers to different virtual registers at the same program point if they share a common subregister. LLVM provides a mechanism called *register unit* for you determine if two physical registers overlap, see Section 3.2.
- Simple function calls: you can assume that all function arguments are passed through registers. Besides usual defs and uses, a call instruction also have a `RegMask` operand, which you can check using `MachineOperand::isRegMask`. You can use `MachineOperand::getRegMask` and `MachineOperand::clobbersPhysReg` to check if a physical register may potentially be changed by the called function.
- Existing use of physical registers, such as in argument passing and return values.
- If a live virtual register is not dirty (i.e. not changed after reloading), you do not need to store it back to the stack slot when spilled.
- No need to insert spill instructions after return instructions (check with `MachineInstr::isReturn`).
- If a virtual register or physical register is killed at an instruction, i.e., it's not live after the instruction, then you do not need to spill it (check using `MachineOperand::isKilled`).

- For grading (and also debugging) purposes, you need to increment the variables `NumStores` and `NumLoads` to record the number of stores (spills) and loads (reloads) inserted when compiling a program. You can print them using the `-stats` flag in `llc`.

Note that for simplicity, the pseudocode in Section 6 makes more assumptions than you should. See Section 3 for how you can bridge the gap using the LLVM APIs.

3 Useful LLVM APIs

It may sound overwhelming to build a register allocator for the complex LLVM framework, so this section introduces some notions and resources to help you to get started.

Before reading the following sections, please read these documentations first to get a general idea of how passes in LLVM work and the use of `llc`: Writing an LLVM Pass, `llc` - LLVM static compiler.

In particular, the register allocation pass that you will implement is a `MachineFunctionPass` used via `llc`. It will be called on a `MachineFunction` and should modify this function to eliminate virtual registers.

The following sections introduce some basic notions involved in register allocation (Sections 3.1 and 3.2) and useful APIs (Section 3.3).

3.1 Machine IR

Machine IR is an intermediate representation in LLVM for code generation passes after instruction selection and before the actual machine code emission. In a nutshell, Machine IR

- looks similar to assembly code and contains target-specific instructions and registers, and
- still contains control-flow information and can have Phi instructions to optionally stay in SSA form.

For example, if you run `llc -print-after-all` on some LLVM IR (which prints out the program after each backend pass), most of the code printed would be in Machine IR.

The main classes of Machine IR that you will deal with in this assignment are

- `llvm::MachineFunction`
- `llvm::MachineBasicBlock`
- `llvm::MachineInstr`
- `llvm::MachineOperand`

These classes all implement convenient iterators. For example, given some `MachineFunction MF`, you can iterate through all machine operands in the function by

```
for (MachineBasicBlock &MBB : MF) {
  for (MachineInstr &MI : MBB) {
    for (MachineOperand &MO : MI.operands()) {
      ...
    }
  }
}
```

You are encouraged to check out the header files in the LLVM source code for these classes to see what methods they have.

3.2 Registers

Registers in the Machine IR will be the primary objects that you operate on. The main class for registers is `llvm::Register`, which can represent both physical and virtual registers. For example, in the x86 backend, `AH`, `EAX`, `%1` are all different registers (`%1` denotes a virtual register). Distinct virtual registers are simply represented by distinct integer numbers.

A useful notion in LLVM for register allocation is *register units*. In LLVM, every physical register is associated with a set of register units (specified by the target), and they can be used to determine if assigning to one register interferes the other. Specifically, two physical registers overlap iff they have a common register unit. For instance, below are some mappings of register to register units:

$$\begin{aligned}
\texttt{RAX} &\longmapsto \{\texttt{AH}, \texttt{AL}\} \\
\texttt{EAX} &\longmapsto \{\texttt{AH}, \texttt{AL}\} \\
\texttt{AX} &\longmapsto \{\texttt{AH}, \texttt{AL}\} \\
\texttt{AH} &\longmapsto \{\texttt{AH}\} \\
\texttt{AL} &\longmapsto \{\texttt{AL}\}
\end{aligned}$$

Therefore, your allocator can tell that `AL` overlaps with `EAX` but not with `AH`, without knowing details of the target ISA. You can get the register units of a physical register using `llvm::MCRegUnitIterator`.

3.3 A List of Classes/Methods Useful for Register Allocation

Classes:

- `MachineFunction`, `MachineBasicBlock`, `MachineInstr`, `MachineOperand`
- `Register`, `MCRegister`, `MCPhysReg`
- `TargetRegisterClass` describes properties of a set of registers (e.g. general purpose registers of size 32), which can be obtained using `MachineRegisterInfo::getRegClass`. Also see `llvm/lib/Target/X86/X86RegisterInfo.td` for a full list of register classes for x86.

Methods:

- `outs`, `dbgs` are output streams useful for debugging.

- `MachineOperand::getReg`, `MachineOperand::setReg`
- `MachineOperand::isDef`, `MachineOperand::isUse`
- `Register::isPhysical`, `Register::isVirtual`
- `MachineBasicBlock::liveins` is an iterator for live variables at the entry of a block.
- `MachineOperand::isKill`, `MachineOperand::setIsKill`, an operand is marked `kill` if it's the last use of a register.
- `MachineOperand::getSubReg`, `MachineOperand::setSubReg`. Virtual register uses may be marked a *subregister index*, but when you replace it with a physical register, you need to get the corresponding physical subregister using `TargetRegisterInfo::getSubReg` and set the operand's register to that (using `MachineOperand::setReg` and setting the subregister index to 0 using `setSubReg`).
- `RegisterClassInfo::getOrder` returns a preferred order of allocation for a specific register class.
- `TargetInstrInfo::loadRegFromStackSlot` is used for reloading.
- `TargetInstrInfo::storeRegToStackSlot` is used for spilling.
- `TargetRegisterInfo::getSpillSize` computes spill size for a `TargetRegisterClass`.
- `TargetRegisterInfo::getSpillAlign` computes spill alignment for a `TargetRegisterClass`.
- `MachineFrameInfo::CreateSpillStackObject` allocates a stack slot for spilling.
- `printRegUnit` converts a register to something printable to an output stream.

4 Tips for Implementation and Testing

- Understand the notion of an LLVM pass. The document Writing an LLVM Pass contains some useful information. In particular, the pass that you will need to implement is a `MachineFunctionPass`.
- Understand where your register allocator pass lies in the entire backend pipeline of LLVM. You can use the `-print-after-all` flag in `llc` to print the pass names and Machine IR after each backend pass, and see how the function is being transformed in each pass.
- Use the flag `-verify-machineinstrs` when testing. This will check for some obvious issues in the machine code generated, such as if all virtual registers are eliminated or if a physical register is used without first defined.
- Use the flag `-stats` to print out the statistics in your register allocator.
- Write your debug output to the stream `dbgs()`, and then you can enable the debug output by

  - `-debug` to print all debug output (including other passes)
  - `-debug-only=regalloc` to print only the debug output in register allocator

- If you find you are writing code for doing something basic within LLVM, it is quite likely that the code for this exists already.
- Try using the LLVM built-in data structures in `llvm/include/llvm/ADT/` – they offer a similar API to the C++ STL interface you're likely familiar with, there's a wider range of data structures to choose from, and they're often more efficient than STL in LLVM-specific scenarios. See LLVM Programmers Manual for how to choose and use them.

5 Submission and Grading

For grading, you only need to hand in your modified `RegAllocSimple.cpp`. Please do not change any other source files. We will test your submission against programs similar to the tests provided. Your submission will be evaluated on:

- (70%) Whether the tests are successfully compiled and executed with the expected output using the following flags to `llc`:
  - `-verify-machineinstrs`
  - `-regalloc=simple`
  - `-O0`, `-O1`, `-O2`
- (30%) If the statistics recorded in `NumStores` and `NumLoads` are comparable or better than a reference implementation, which should be easy to achieve if you implement the cases mentioned in Section 2. For the provided tests, the reference stores and loads at `-O0` are

Test	Stores	Loads
even.ll	9	4
fib.ll	7	4
max.ll	8	3
rotate.ll	4	0

  We will provide additional test cases and reference counts for them closer to the due date.

To hand in your solution, run the program with 2 arguments – the name of the MP and the directory containing `RegAllocSimple.cpp`:

```
/class_cs426/public/cs426_handin mp4 <path_to_dir>
```

If you used LLM-based code-generating tool in your submission, follow the same rules as in MP1:

- If you used a prompted code generator, such as GPT-4, annotate every block of code that was assisted by the code generator (even if you further edited the code generator's output). You should label and number each such code block with comments (e.g., `/* LLM Block 1 */`) and clearly indicate the start/end line of the block. Comment at the end of the file on (1) the tool you used (GPT-3.5, GPT-4, etc.) and (2) the full prompt you gave to the tool, for each code block.
- If you used an unprompted code generator, such as GitHub Copilot, comment at the beginning of each file the name of the tool you used, and indicate any large block of code (≥ 5 lines) you got from the tool.
- We will manually check these comments and may try to reproduce some of the code generation results with your prompts.

6 A Local Register Allocation Algorithm: Pseudocode

```
// Assuming:
// - All registers have the same size
// - No existing use of physical registers
// - Different physical registers do not overlap
// - No function calls
// Note that you should NOT make some of these assumptions
// See Section 2 for details.
// We maintain the following state
SpillMap = {}     // virtual register -> stack slot
LiveVirtRegs = {} // virtual register -> physical register

// Allocates all virtual registers in F and spills/reloads as necessary
void allocate(MachineFunction F) {
  for (MachineBasicBlock BB: F) {
    this->LiveVirtRegs = {}
    for (MachineInstruction I: BB) {
      // physical registers used in the current instruction
      UsedInInstr = {}
      // Allocate uses first
      for (MachineOperand &MO : I.operands()) {
        if (MO.isUse()) {
          P = allocateVirtualRegister(MO, IsUse=true, UsedInInstr);
          UsedInInstr.add(P);
          setMachineOperandToPhysReg(MO, P);
        }
      }
      // Spill all clobbered registers before a call.
      // If the current instruction `I` is a call, one of the
      // arguments will be the special type `RegMask`.
      for (MachineOperand &MO : I.operands()) {
```

```
        if (MO.isRegMask()) {
          const uint32_t *Mask = MO.getRegMask();
          // For each live virtual register, check if
          // the phys register it's in is clobbered as indicated by the mask
          for (auto Pair : this->LiveVirtRegs) {
            if (MachineOperand::clobbersPhysReg(Mask, ...)) {
              Spill here
            }
          }
        }
      }
      // Allocate defs
      for (MachineOperand &MO : I.operands()) {
        if (MO.isDef()) {
          P = allocateVirtualRegister(MO, IsUse=false, UsedInInstr);
          UsedInInstr.add(P);
          setMachineOperandToPhysReg(MO, P);
        }
      }
    }
    Spill all registers in LiveVirtRegs
  }
}

// Input:
//  - R            Virtual register to be allocated
//  - IsUse        Indicating whether the operand is a use
//  - UsedInInstr  Physical registers used in the current instruction
// Output: An available physical register, possibly after spilling
MCPhysReg allocateVirtualRegister(MachineOperand R, bool IsUse, ... UsedInInstr) {
  if (R in this->LiveVirtRegs) {
    return this->LiveVirtRegs[R];
  }
  // Get a list of phys registers that can hold R in HW preference order
  auto &RC = *MRI->getRegClass(R);
  auto AllocationOrder = RegClassInfo.getOrder(&RC);
  MCPhysReg Found;
  for (MCPhysReg PhysReg : AllocationOrder) {
    if (PhysReg is unused) {
      Found = PhysReg;
    }
  }
  if (not Found) {
    // Find a physical register to spill
    for (MCPhysReg PhysReg : AllocationOrder) {
      if (PhysReg not in UsedInInstr and V = findVirtRegInPhysReg(PhysReg)) {
        Spill V
```

```
        Found = PhysReg;
      }
    }
  }
  // If we are using a virtual register and it's not already live,
  // it must have been spilled into the stack.
  // So we reload the value here.
  if (IsUse) {
    Slot = findOnStack(R);
    loadStackSlotInto(Slot, Found);
  }
  LiveVirtRegs[R] = Found;
  return P;
}
```

The appropriate LLVM API calls for binding a `MachineOperand` on a found physical register is given here – this is not pseudocode.

```
void setMachineOperandToPhysReg(MachineOperand &MO, Register PhysReg) {
  unsigned SubRegIdx = MO.getSubReg();
  if (SubRegIdx != 0) {
    PhysReg = TRI->getSubReg(PhysReg, SubRegIdx);
    MO.setSubReg(0);
  }
  MO.setReg(PhysReg);
  if (MO.isKill()) {
    MO.setIsKill(false);
  } else if (MO.isDead()) {
    MO.setIsDead(false);
  }
  MO.setIsRenamable();
}
```