6.035 Project 5 Report

General Architecture
In the backend implementation of the compiler, the AST of the input program is first converted into control flow graphs for each function. A control flow graph is implemented as a graph of basic blocks (BasicBlock), where each basic block is backed by an ArrayList of Instructions. There are some constraints to a control flow graph, that every basic block can have 0, 1, or 2 outgoing connections. Additionally, every basic block also keeps track of all other blocks that go into itself. To perform any constant or mutating operation, such as printing the content of the CFG(a.k.a., dumping the assembly content), or performing a CSE pass, on a control flow graph, an instance implementing the class BasicBlockVisitor is used. BasicBlockVisitor is an abstract class that separates the algorithm for graph traversal from the logic that is applied per basic block. We will discuss about this later.

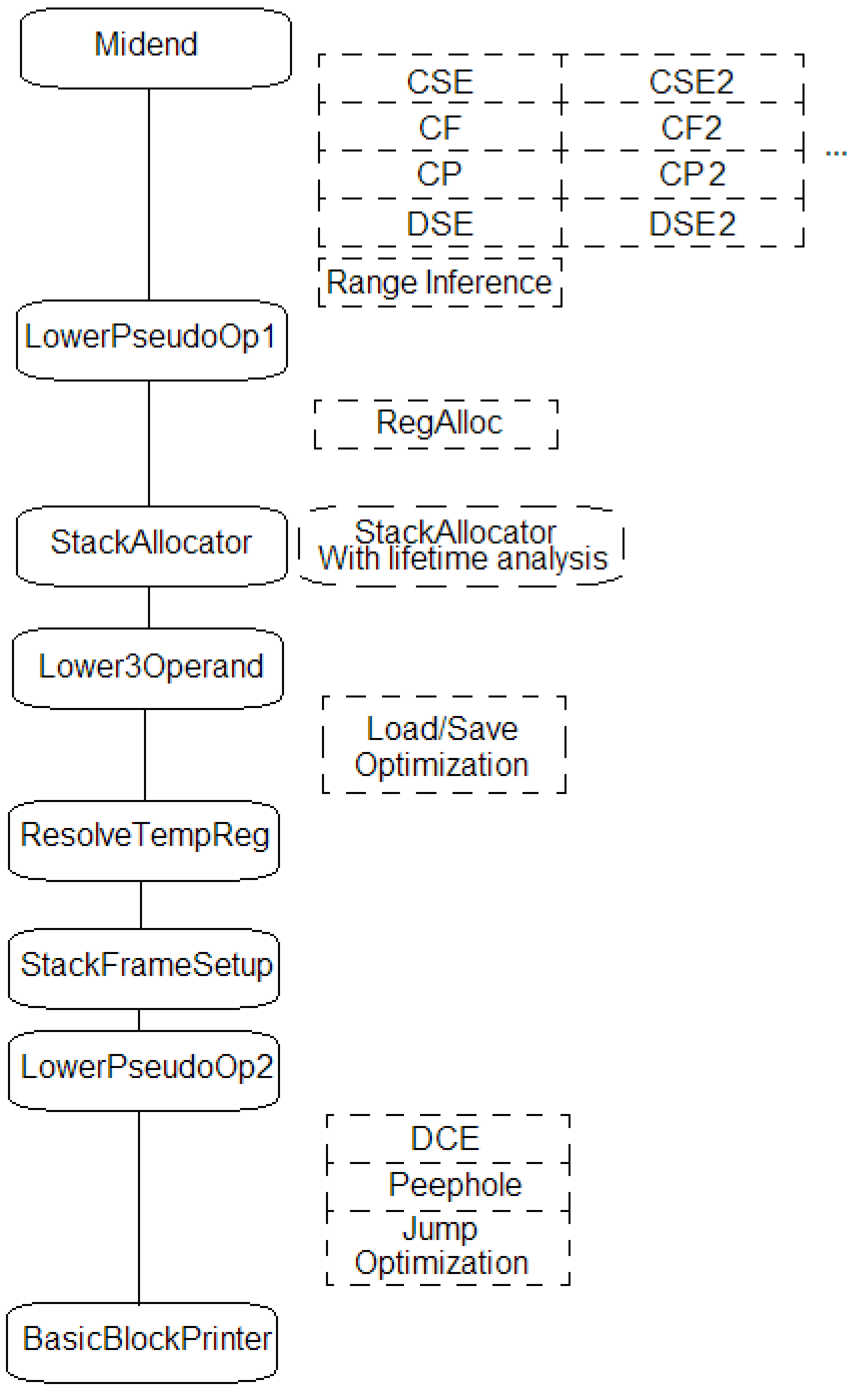
Low Level IR

The IR we use comes in two forms: ISA and pseudo op. In ISA (Instruction Set Architecture) form the instruction conforms to the x86 assembly specification, thus can be directly printed out as the final output. In pseudo op form the instruction has a more intuitive 3-operand representation (for example, `c = add a, b`), and the operands need not to conform to x86 specification, such as at most one memory operand per instruction. The generation of pseudo op is still checked to make sure that it logically makes sense (for example, having the correct number of operands). Most operations, in particular, arithmetic operations, can be represented in both forms. However, there are also some pure pseudo ops that don’t have corresponding assembly instructions. They are used to represent operations that are too complicated to represent using x86 ISA, and also they are intentionally inserted as hints to some optimizer stages. At the beginning when the CFG is created, most instructions are in pseudo op form, and there is a fixed stage pipeline that eventually transforms all pseudo ops into ISA instructions.

Codegen Pipeline

The code generation pipeline consists of a series of fixed stages, where each performs certain transformations to the CFG (mostly function-wise) and eventually converting all pseudo ops into actual assembly. Between some fixed stages, optional optimization stages can be inserted at some given point. Below is a diagram of the pipeline, which includes all fixed stages and optimization stages, including some that are to be implemented. Each stage is implemented with a specialized BasicBlockVisitor class. The abstract base class provides a generic algorithm to traverse a CFG from either direction, and the implementation needs to specify the following: (1) a State class, which carries information prior to the entry of a basic bock (for example, the IN(x) set for program analysis); (2) a merge operation for two State objects - This is used when a basic block has more than one predecessors, each predecessor provides an output state, and the merge operation decides what will be the combined state for the input of the current basic block; (3) and finally, a visit method, that is applied per basic block, given the input state, to provide the output state. When a merge between two incoming states occurs, the output of the merge operation is compared with the pre-merge inputs, if it is equal to either input state, it means the states converge and the outgoing block is not visited, otherwise, the outgoing block is visited again with the merged state. This process repeats indefinitely until all states converge.

There are two (abstract) specializations of the base visitor. The first one is a BasicBlockTraverser. For an operation on a basic block that is predecessor independent (doesn't care about which basic block it comes from), the state is just a generic object and the merge operation always returns the same object. This implies each basic block will only be visited exactly once. BasicBlockPrinter is the most typical application.



The second specialization is an Analyze-Transform pass. For some optimizations, the optimization action cannot be performed on the basic block on the first visit, since a second visit of the same block with a different input state may led to different actions. For example, dead store removal cannot be performed on first visit because it is not known whether a variable is dead until all path coming out from this block has been traversed and their states converged, then we can conclude that the variable is globally dead thus ok to remove it. The Analyze-Transform pattern provides this abstraction: it actually consists of two BasicBlockVisitors, the first one traverses the basic blocks while *simulating* the operation on each basic block until the states of all blocks converge, and the BasicBlockVisitors stores the input/output states for each block, and the second BasicBlockVisitors visits each basic block exactly once, and it uses the saved input/output states corresponding to the block to *replay* the operation, and then actually performs modifying operations on the basic block.

Fixed Pipeline Stages

All programs will go through these stages, regardless of optimization settings. However, some of these stages behave slightly different under some optimization settings.

As a convention here, register `rax` is always reserved as a real temporary register. Its lifetime is considered ended if it is not used consecutively at any point.

Midend

This is actually not a BasicBlockVisitor, but an IR Visitor instead. It generates pseudo op for the program and constructs a CFG. In this stage pseudo ops use virtual registers (Value objects) instead of ISA registers, and it is assumed that there are infinite registers. For real instructions that have implicit register operands (for example, `idiv`), it generates a high level pseudo op instead, so that the optimization stages can handle fewer exception cases.

LowerPseudoOp1

Some complex pseudo ops, such as `fake_div`, and `fake_call`, will be replaced by a sequence of simpler pseudo ops that have their corresponding ISA instructions.

StackAllocator

All virtual registers (Value objects) will be assigned to different stack locations using `rbp` addressing, if they haven't been assigned to a physical register[1] yet. By default none of the virtual register is assigned to a physical register, unless register allocation optimization is used, then this stage will pick up any virtual registers that have not been allocated with a physical register. For the basic StackAllocator, the lifetime of a value is not considered so different values will occupy a different stack offset.

[1]Physcial Register here means x86 logical registers, not the hardware physical registers.

Lower3Operand

After this stage all arithmetic pseudo ops will become ISA instructions. Instruction in 3-operand form (pseudo op) are converted into 2-operand form, with the use of `rax` temporary register. This stage performs a little optimization regardless of settings, which includes (non-propagating) constant folding, and some forms of peephole optimization. The reason is the limitation that only one temporary register is always available. For example, pseudo ops in the form of `reg = Op Imm64, Imm64` requires constant folding because it is not allowed to have two immediate operand in ISA, while pseudo op like this may drop to this stage since Constant Folding is not necessary enabled.

ResolveTempReg

This is an ad-hoc stage to deal with caller setup for function calls. Since the call instruction has too many side effects. It performs full register lifetime analysis and inserts instructions around call sites to handle caller-saved registers.

StackFrameSetup (**Affected by `--opt=omit_frame_pointer`**)

This stage calculates the amount of stack space needed for the function call, and inserts instructions at the entry and all exits of the function to adjust base and stack pointers. Stack pointer adjustment is coalesced into one subtraction/addition regardless of optimizations. Stack frame set up is deferred to this stage because the ResolveTempReg stage can introduce an indefinite amount of temporaries, so the size of the stack frame is not determined before that stage.

Optimization: if omit frame pointer (`omit_frame_pointer`) is enabled, base pointer is not saved/restored since it is used as a general purpose register. In addition, all base pointer indexing memory locations are converted to stack pointer indexing with a simple algebraic transformation. Note that this stage does not promote stack variable with the now general purpose `rbp` because register allocation should be done by RegAlloc optimization.

LowerPseudoOp2

Another ad-hoc stage to deal with any unconverted pseudo ops.

BasicBlockPrinter

Prints the final assembly output. Each basic block is assigned with a priority value upon creation in Midend, and their priority values are adjusted by the Midend such that relevant blocks (for example, sibling blocks in a conditional) are printed near each other using the default traversal algorithm.

Optimization Stages

CSE (enabled with `--opt=cse`)

CSE optimization pass is performed on pure pseudo ops. It is implemented using the Analyze-Transform pattern. It detects common subexpressions consisting two operands and replaces instructions that repeat the computation with temporaries holding the subexpressions. It is able to perform these non-obvious CSE:

(1) Combining division and modular with the same operands: this is due to the fact that `idiv` instruction gives both the quotient and the remainder simultaneously, and the instruction is also very expensive. An ad-hoc operand class that stores two operands (as the results) is created to handle this.
(2) CSE on Range checking: range checking is represented by a pseudo op that takes an index operand, and an immediate operand, and it behaves like an arithmetic operation without giving a result, therefore CSE also applies to it.
(3) CSE on loads: loads are represented by a pseudo op that takes an index and a base operand and returns a value, therefore CSE also applies.
(4) CSE on operations involving globals: this is valid as long as the globals are not overwritten, which may happen in a call instruction, in this case all recorded common subexpressions involving any globals are invalidated. As a design decision, I chose not to perform globals usage analysis because that requires a callgraph analysis, and it takes too much to implement.

CSE doesn't improve performance, because it introduces a lot of temporaries. It should be combined with CP in order to gain performance.

CP (enabled with `--opt=cp`)

CP performs copy propagation. It replaces all copies of a local variable with the variable itself as long as the variable is still alive. It doesn't perform on global variables because of potential side effects of function calls.

CP definitely improves performance because it strictly reduces the number of instruction being executed, and it also reduces the number of temporaries globally, which decreases on register/stack usage.

Register Allocation (implemented but contains bugs, therefore not available `--opt=regalloc`)

Also affected by `--opt=omit_frame_pointer`

This pass performs heuristic register allocation. In the analysis phase, for each virtual register (Value object) in the function, a number indicating the value's usage frequency is assigned. This value is a heuristic in order to determine the most frequently used virtual registers. Each use of a virtual register counts as 1 point, self-reassignment (`a = a + b`) counts as 2 points, and loop variable counts as 3 times the number of instructions in the loop body. Then all virtual registers are sorted based on their heuristics. From the highest heuristic value to the lowest, there will be attempts to assign an ISA physical register to each virtual register, and the attempts repeat until either a register is successfully assigned, or there is no such assignment possible, in this case that virtual register will stay on the stack. If the optimization `omit_frame_pointer` is enabled, then `rbp` will also be used as a general purpose register in the assignment process.

This optimization will improve the performance the most, because an arithmetic operation on all register operands is generally about 3 times faster than the same operation with one memory operand. Furthermore, register allocation reduces the stack usage, therefore indirectly increases cache coherency for stack variables. As a design choice, a simple greedy algorithm is used to determine register assignment instead of using dynamic programming to determine the optimal register allocation, due to the fact that most temporaries are actually intermediate results, which are values that are used only once and for all. Greedy algorithm is very efficient (and near optimal) on assigning registers to short-lived temporaries.

Register allocation was taken offline for the checkpoint because it has bugs.

Jump Optimization (Not yet implemented for checkpoint, will implement for derby, **`--opt=branch`**)

This is a micro-optimization in late stage to remove unnecessary jump instructions (for example, jumping to the immediate following instruction), and it also removes empty (but reachable) blocks.

This optimization is useful because after constant folding for conditionals there are unreachable blocks generated, and eliminating unnecessary jumps increases performance because non-predictable jump is a relatively expensive instruction.

Peephole Optimization (Not yet implemented for checkpoint, will implement for derby **`--opt=peephole`**)

This consists of multiple instruction level optimizations. They are theoretically beneficial but the effect may be insignificant, so this optimization is placed in a low development priority. The optimizations contains the following:

- Arithmetic strength reduction: replacing expensive operations with cheap operations
- Zeroing a register: use `xor reg, reg` instead of `mov $0, reg`
- Excessive Comparison removal: a `cmp`/`test` can be removed, if there is a succeeding instruction that modifies flags without testing flags, or if the instruction is a self test (`cmp $0, x`, or `test x, x`) and there is a preceding flag-modifying instruction that creates `x`.

Advanced Stack Allocator (Considered, not yet implemented **`--opt=stackalloc`**)

This stack allocator will reuse the same stack location once the lifetime of a stack variable ends. It is similar to register allocator, and it benefits performance by furthermore reducing stack usage therefore increasing cache coherence.

Optimizations that will not be implemented

Data parallelization

Based on GCC's optimized output for most programs, data parallelization is not that commonly used. In addition, the SSE instruction itself is not free – it takes additional instructions to move data between general purpose registers and SSE registers.

List scheduling

Modern CPU has hardware instruction rescheduling functionality, so I doubt if this optimization can give too much performance.

Sample Output at Different Stages

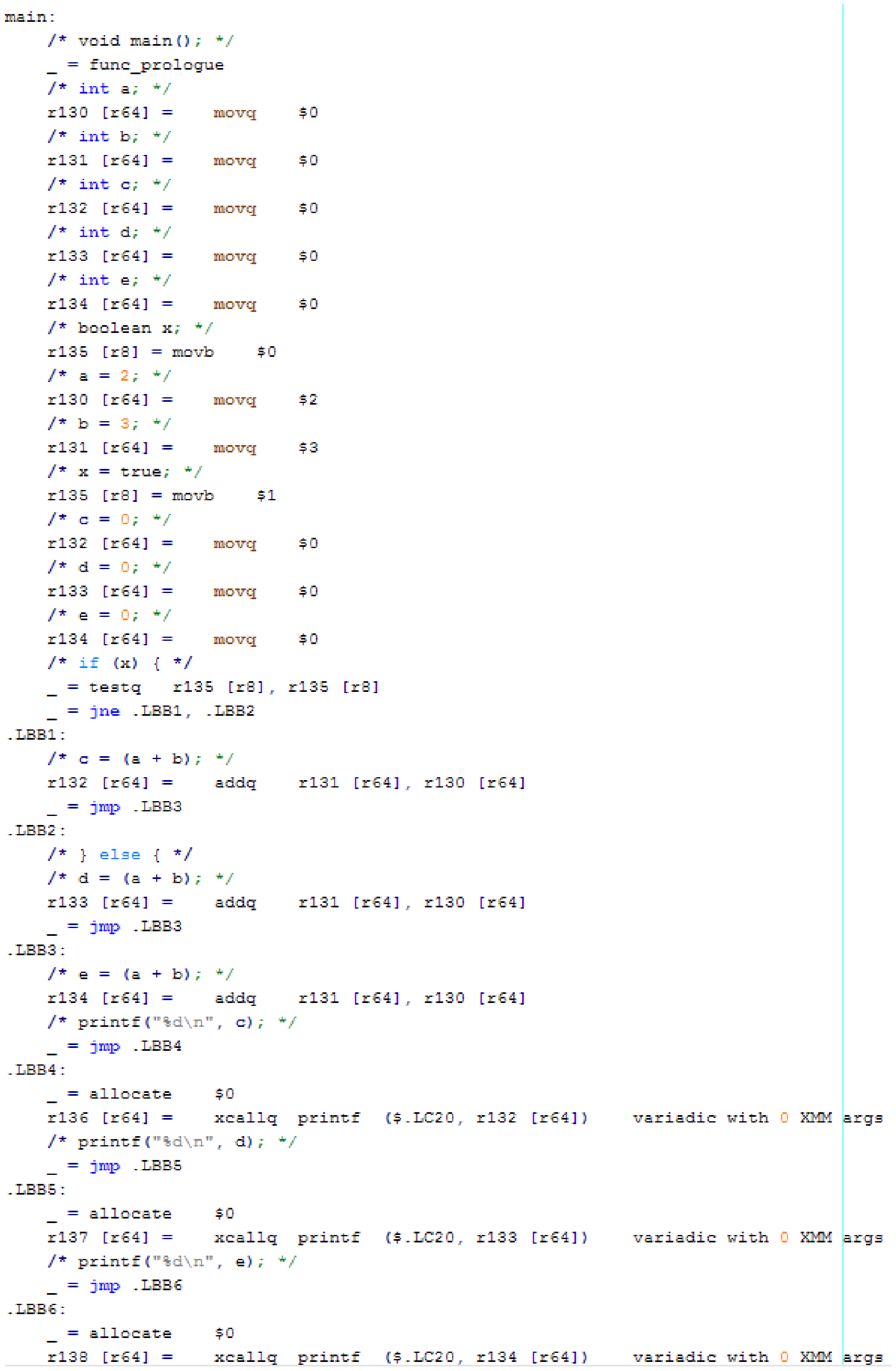
For low level IR and assembly, only the main function is shown since the complete code is really long.

Input program

```
void main ( ) {
  int a, b, c, d, e;
  boolean x;
  a = 2;
  b = 3;
  x = true;
  c = 0;
  d = 0;
  e = 0;
  if ( x ) {
    c = a + b;
  }
  else {
    d = a + b;
  }
  e = a + b;
  printf ( "%d\n", c );
  printf ( "%d\n", d );
  printf ( "%d\n", e );
}
```

6.035 Project 5 Report

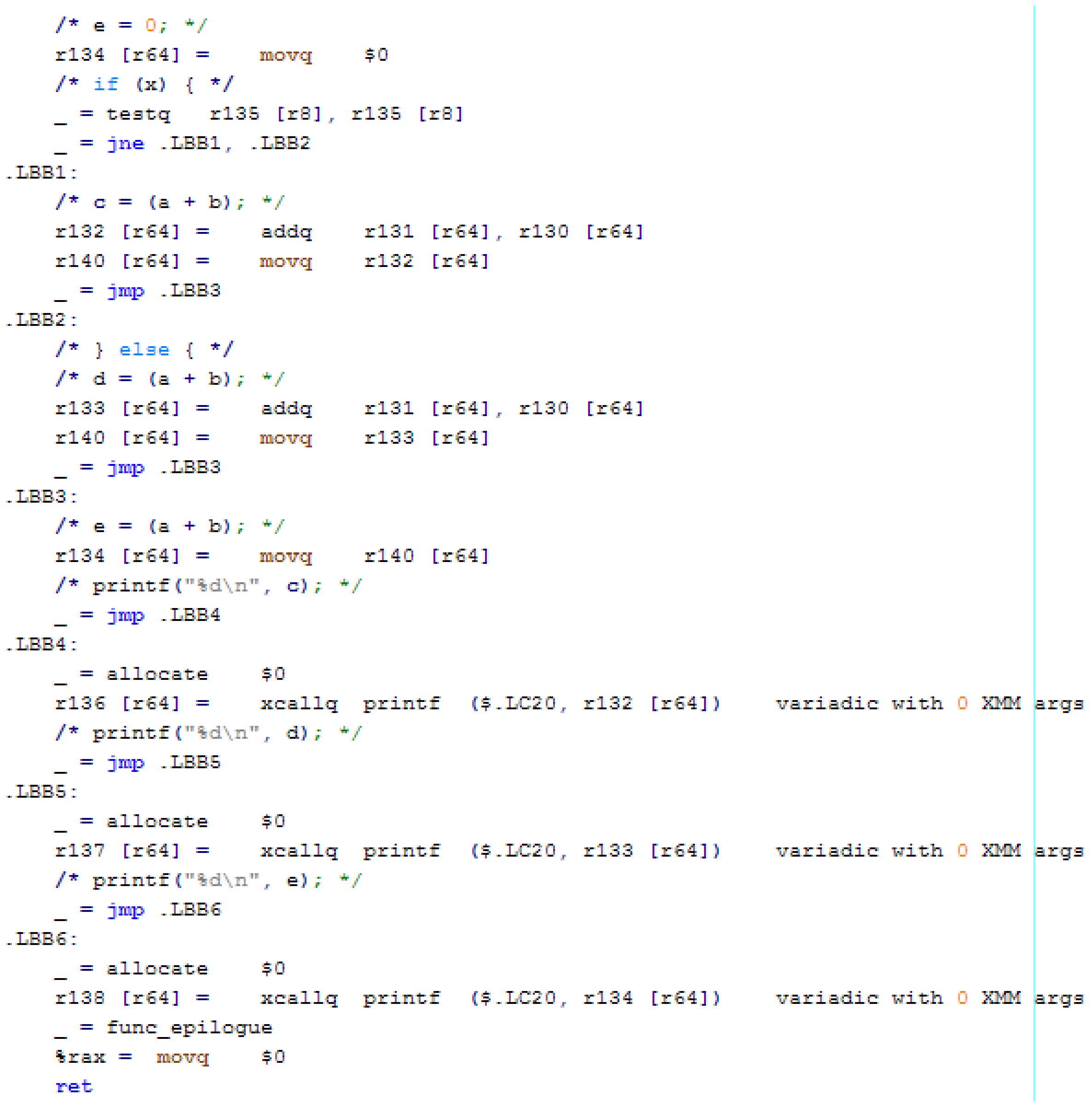
Low Level IR (main function only)



```
main:
    /* void main(); */
    _ = func_prologue
    /* int a; */
    r130 [r64] =    movq    $0
    /* int b; */
    r131 [r64] =    movq    $0
    /* int c; */
    r132 [r64] =    movq    $0
    /* int d; */
    r133 [r64] =    movq    $0
    /* int e; */
    r134 [r64] =    movq    $0
    /* boolean x; */
    r135 [r8] = movb    $0
    /* a = 2; */
    r130 [r64] =    movq    $2
    /* b = 3; */
    r131 [r64] =    movq    $3
    /* x = true; */
    r135 [r8] = movb    $1
    /* c = 0; */
    r132 [r64] =    movq    $0
    /* d = 0; */
    r133 [r64] =    movq    $0
    /* e = 0; */
    r134 [r64] =    movq    $0
    /* if (x) { */
    _ = testq   r135 [r8], r135 [r8]
    _ = jne .LBB1, .LBB2
.LBB1:
    /* c = (a + b); */
    r132 [r64] =    addq    r131 [r64], r130 [r64]
    _ = jmp .LBB3
.LBB2:
    /* } else { */
    /* d = (a + b); */
    r133 [r64] =    addq    r131 [r64], r130 [r64]
    _ = jmp .LBB3
.LBB3:
    /* e = (a + b); */
    r134 [r64] =    addq    r131 [r64], r130 [r64]
    /* printf("%d\n", c); */
    _ = jmp .LBB4
.LBB4:
    _ = allocate    $0
    r136 [r64] =    xcallq  printf  ($.LC20, r132 [r64])    variadic with 0 XMM args
    /* printf("%d\n", d); */
    _ = jmp .LBB5
.LBB5:
    _ = allocate    $0
    r137 [r64] =    xcallq  printf  ($.LC20, r133 [r64])    variadic with 0 XMM args
    /* printf("%d\n", e); */
    _ = jmp .LBB6
.LBB6:
    _ = allocate    $0
    r138 [r64] =    xcallq  printf  ($.LC20, r134 [r64])    variadic with 0 XMM args
```

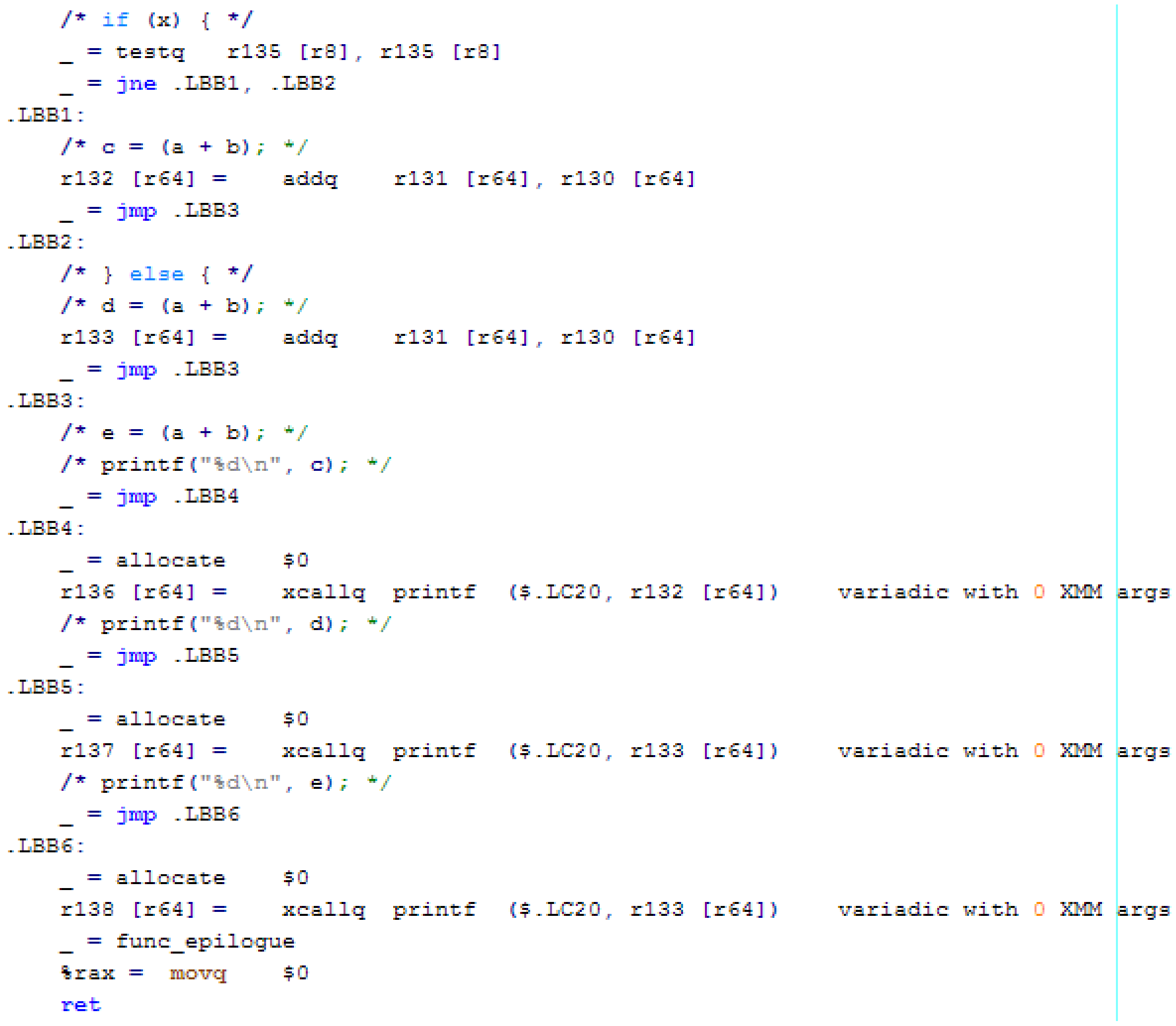
After CSE

(Initial assignments are skipped to reduce length)



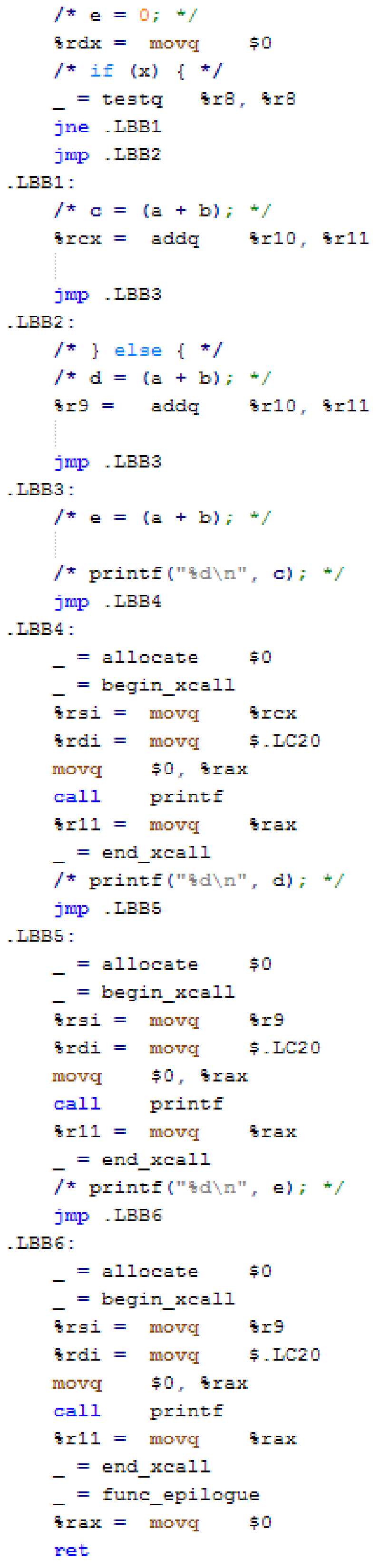
```
    /* e = 0; */
    r134 [r64] =    movq    $0
    /* if (x) { */
    _ = testq   r135 [r8], r135 [r8]
    _ = jne .LBB1, .LBB2
.LBB1:
    /* c = (a + b); */
    r132 [r64] =    addq    r131 [r64], r130 [r64]
    r140 [r64] =    movq    r132 [r64]
    _ = jmp .LBB3
.LBB2:
    /* } else { */
    /* d = (a + b); */
    r133 [r64] =    addq    r131 [r64], r130 [r64]
    r140 [r64] =    movq    r133 [r64]
    _ = jmp .LBB3
.LBB3:
    /* e = (a + b); */
    r134 [r64] =    movq    r140 [r64]
    /* printf("%d\n", c); */
    _ = jmp .LBB4
.LBB4:
    _ = allocate    $0
    r136 [r64] =    xcallq  printf  ($.LC20, r132 [r64])    variadic with 0 XMM args
    /* printf("%d\n", d); */
    _ = jmp .LBB5
.LBB5:
    _ = allocate    $0
    r137 [r64] =    xcallq  printf  ($.LC20, r133 [r64])    variadic with 0 XMM args
    /* printf("%d\n", e); */
    _ = jmp .LBB6
.LBB6:
    _ = allocate    $0
    r138 [r64] =    xcallq  printf  ($.LC20, r134 [r64])    variadic with 0 XMM args
    _ = func_epilogue
    %rax =  movq    $0
    ret
```

After CP



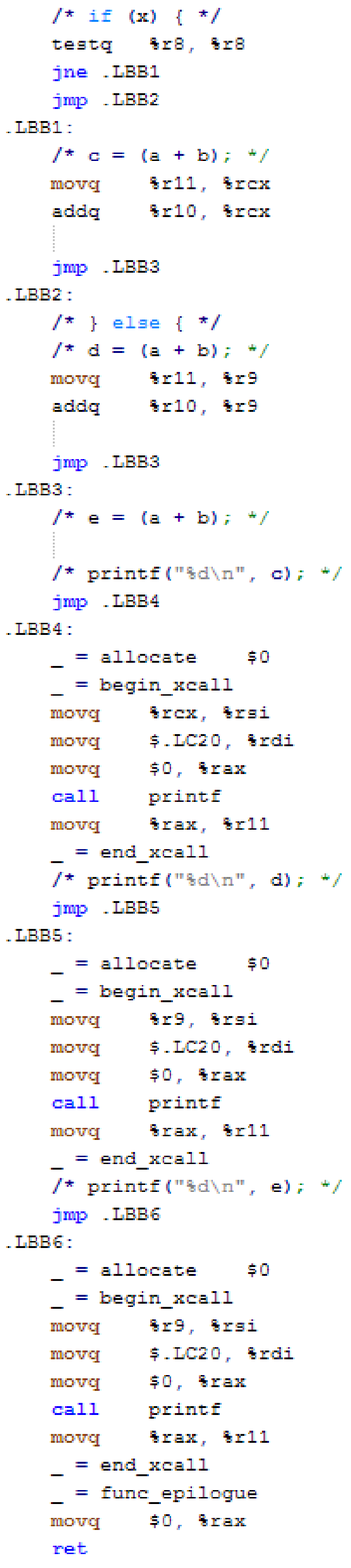
```
    /* if (x) { */
    _ = testq   r135 [r8], r135 [r8]
    _ = jne .LBB1, .LBB2
.LBB1:
    /* c = (a + b); */
    r132 [r64] =    addq    r131 [r64], r130 [r64]
    _ = jmp .LBB3
.LBB2:
    /* } else { */
    /* d = (a + b); */
    r133 [r64] =    addq    r131 [r64], r130 [r64]
    _ = jmp .LBB3
.LBB3:
    /* e = (a + b); */
    /* printf("%d\n", c); */
    _ = jmp .LBB4
.LBB4:
    _ = allocate    $0
    r136 [r64] =    xcallq  printf  ($.LC20, r132 [r64])    variadic with 0 XMM args
    /* printf("%d\n", d); */
    _ = jmp .LBB5
.LBB5:
    _ = allocate    $0
    r137 [r64] =    xcallq  printf  ($.LC20, r133 [r64])    variadic with 0 XMM args
    /* printf("%d\n", e); */
    _ = jmp .LBB6
.LBB6:
    _ = allocate    $0
    r138 [r64] =    xcallq  printf  ($.LC20, r133 [r64])    variadic with 0 XMM args
    _ = func_epilogue
    %rax =  movq    $0
    ret
```

After Register Allocation (with `omit_frame_pointer`)



```
    /* e = 0; */
    %rdx =  movq     $0
    /* if (x) { */
    _ = testq   %r8, %r8
    jne .LBB1
    jmp .LBB2
.LBB1:
    /* c = (a + b); */
    %rcx =  addq     %r10, %r11

    jmp .LBB3
.LBB2:
    /* } else { */
    /* d = (a + b); */
    %r9 =   addq     %r10, %r11

    jmp .LBB3
.LBB3:
    /* e = (a + b); */

    /* printf("%d\n", c); */
    jmp .LBB4
.LBB4:
    _ = allocate     $0
    _ = begin_xcall
    %rsi =  movq     %rcx
    %rdi =  movq     $.LC20
    movq     $0, %rax
    call     printf
    %r11 =  movq     %rax
    _ = end_xcall
    /* printf("%d\n", d); */
    jmp .LBB5
.LBB5:
    _ = allocate     $0
    _ = begin_xcall
    %rsi =  movq     %r9
    %rdi =  movq     $.LC20
    movq     $0, %rax
    call     printf
    %r11 =  movq     %rax
    _ = end_xcall
    /* printf("%d\n", e); */
    jmp .LBB6
.LBB6:
    _ = allocate     $0
    _ = begin_xcall
    %rsi =  movq     %r9
    %rdi =  movq     $.LC20
    movq     $0, %rax
    call     printf
    %r11 =  movq     %rax
    _ = end_xcall
    _ = func_epilogue
    %rax =  movq     $0
    ret
```

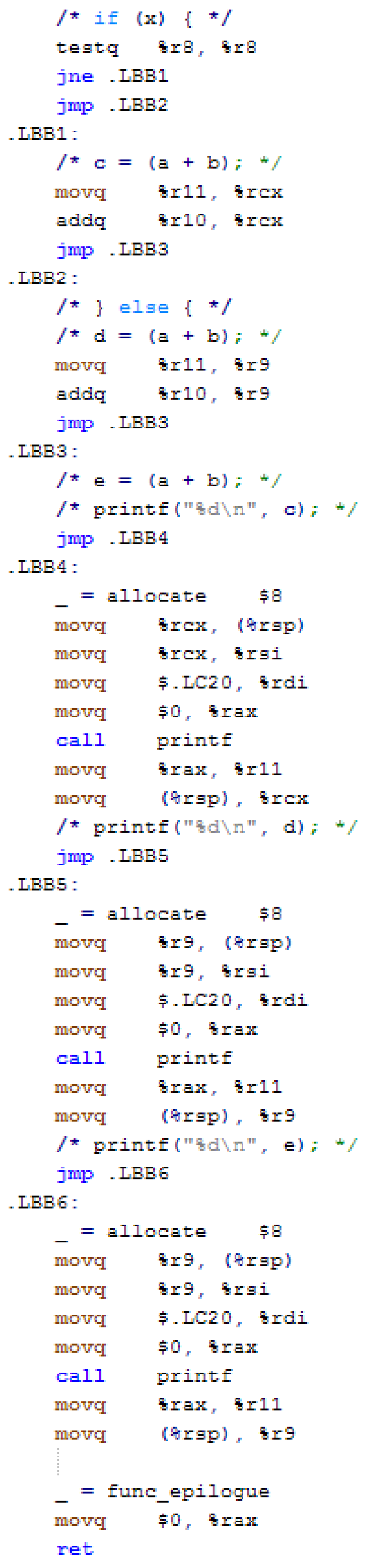
After Lower3Operand



```
    /* if (x) { */
    testq   %r8, %r8
    jne .LBB1
    jmp .LBB2
.LBB1:
    /* c = (a + b); */
    movq    %r11, %rcx
    addq    %r10, %rcx

    jmp .LBB3
.LBB2:
    /* } else { */
    /* d = (a + b); */
    movq    %r11, %r9
    addq    %r10, %r9

    jmp .LBB3
.LBB3:
    /* e = (a + b); */

    /* printf("%d\n", c); */
    jmp .LBB4
.LBB4:
    _ = allocate    $0
    _ = begin_xcall
    movq    %rcx, %rsi
    movq    $.LC20, %rdi
    movq    $0, %rax
    call    printf
    movq    %rax, %r11
    _ = end_xcall
    /* printf("%d\n", d); */
    jmp .LBB5
.LBB5:
    _ = allocate    $0
    _ = begin_xcall
    movq    %r9, %rsi
    movq    $.LC20, %rdi
    movq    $0, %rax
    call    printf
    movq    %rax, %r11
    _ = end_xcall
    /* printf("%d\n", e); */
    jmp .LBB6
.LBB6:
    _ = allocate    $0
    _ = begin_xcall
    movq    %r9, %rsi
    movq    $.LC20, %rdi
    movq    $0, %rax
    call    printf
    movq    %rax, %r11
    _ = end_xcall
    _ = func_epilogue
    movq    $0, %rax
    ret
```

An important stage, stack allocation, is omitted here because the register allocator successfully places all temporaries in registers, so no stack is needed for temporaries, and there is no different after that stage.

Final Output



```
    /* if (x) { */
    testq   %r8, %r8
    jne .LBB1
    jmp .LBB2
.LBB1:
    /* c = (a + b); */
    movq    %r11, %rcx
    addq    %r10, %rcx
    jmp .LBB3
.LBB2:
    /* } else { */
    /* d = (a + b); */
    movq    %r11, %r9
    addq    %r10, %r9
    jmp .LBB3
.LBB3:
    /* e = (a + b); */
    /* printf("%d\n", c); */
    jmp .LBB4
.LBB4:
    _ = allocate    $8
    movq    %rcx, (%rsp)
    movq    %rcx, %rsi
    movq    $.LC20, %rdi
    movq    $0, %rax
    call    printf
    movq    %rax, %r11
    movq    (%rsp), %rcx
    /* printf("%d\n", d); */
    jmp .LBB5
.LBB5:
    _ = allocate    $8
    movq    %r9, (%rsp)
    movq    %r9, %rsi
    movq    $.LC20, %rdi
    movq    $0, %rax
    call    printf
    movq    %rax, %r11
    movq    (%rsp), %r9
    /* printf("%d\n", e); */
    jmp .LBB6
.LBB6:
    _ = allocate    $8
    movq    %r9, (%rsp)
    movq    %r9, %rsi
    movq    $.LC20, %rdi
    movq    $0, %rax
    call    printf
    movq    %rax, %r11
    movq    (%rsp), %r9

    _ = func_epilogue
    movq    $0, %rax
    ret
```