JCC

A pure, no third-party dependencies C compiler

Why?!?

- Good re-introduction back into C after a long period of not using it
- Forces me to build many of my own data structures and designs – the standard library is extremely sparse
- C is simple enough a language that an end-goal of a bootstrapping compiler is viable

Design goals

- No third-party dependencies
  - With one caveat – **GraphViz** command line tool is used for visualizing parts of the parser and IR generator

Style & naming goals

- For this project, I've mainly adopted the **LLVM C-style**
- This is enforced by `clang-format`, but is quite a new code style to me and potentially also you
  - Avoids typedefs for aggregate types – you explicitly write `struct my_type` for every struct variable!
  - Fewer newlines than a lot of other style conventions…
  - Lots of top-level files!
- This style was chosen as something new, and as an experiment

High-level design

- Frontend – Parser + Lexer
- Middle sections – IR building, generation, and lowering
- Backend – machine code emitting, object file creation, and linking
- Major emphasis on reliability and memory-safety
  - Having worked extensively with Rust recently, I am trying to pull across many of the design patterns around ownership to help minimize the landmine that is C memory management

Frontend

- Compilation is on a per-file basis
- Preprocessor is work-in-progress!
  - Lower-priority as it is relatively simple ☺
- Handwritten lexer/parser pair which work in lockstep
  - Lexer provides token on-demand to the parser, rather than entirely tokenizing text
- Parser is a traditional recursive-descent parser with arbitrarily long lookup as is required by the C11 grammar
- The parser generates a top level `ast_translationunit` which contains the AST for the file in a tree-like data-structure
  - JCC is very heavy on intrusive trees and graphs which contain explicit links between structs – this is a natural consequence of using a language without any standard vector, tree, or graph types

IR Building

- The tree is walked downwards, statement-by-statement, and IR is generated as it goes
- The 3 key data structures for IR:
  - `ir_basicblock` – a set of instructions which will always execute together
    - Basicblock start/ends are either branches or branch targets, and so instructions before/after may not always execute together
  - `ir_stmt` - a set of instructions that have a sequence point after them
    - Operations in an `ir_stmt` can be arbitrarily reordered (such as argument evaluation in C), but cannot be reordered between `ir_stmt`s unless the compiler can definitively prove this is not visible to the program
  - `ir_op` - the actual type representing a single operation

ir_op

```c
enum ir_op_flags { IR_OP_FLAG_NONE = 0, IR_OP_FLAG_MUST_SPILL = 1 };

struct ir_op {
  size_t id;
  enum ir_op_ty ty;

  enum ir_op_flags flags;

  struct ir_op_var_ty var_ty;

  struct ir_op *pred;
  struct ir_op *succ;
  struct ir_stmt *stmt;
  union {
    struct ir_op_cnst cnst;
    struct ir_op_binary_op binary_op;
    struct ir_op_ret ret;
    struct ir_op_store_lcl store_lcl;
    struct ir_op_load_lcl load_lcl;
    struct ir_op_br_cond br_cond;
    /* br has no entry, as its target is on `ir_basicblock` and it has no
     * condition */
    struct ir_op_phi phi;
    struct ir_op_mov mov;
  };

  // only meaningful post register-allocation
  unsigned long reg;
  unsigned long lcl_idx;
  void *metadata;
};
```

ir_stmt

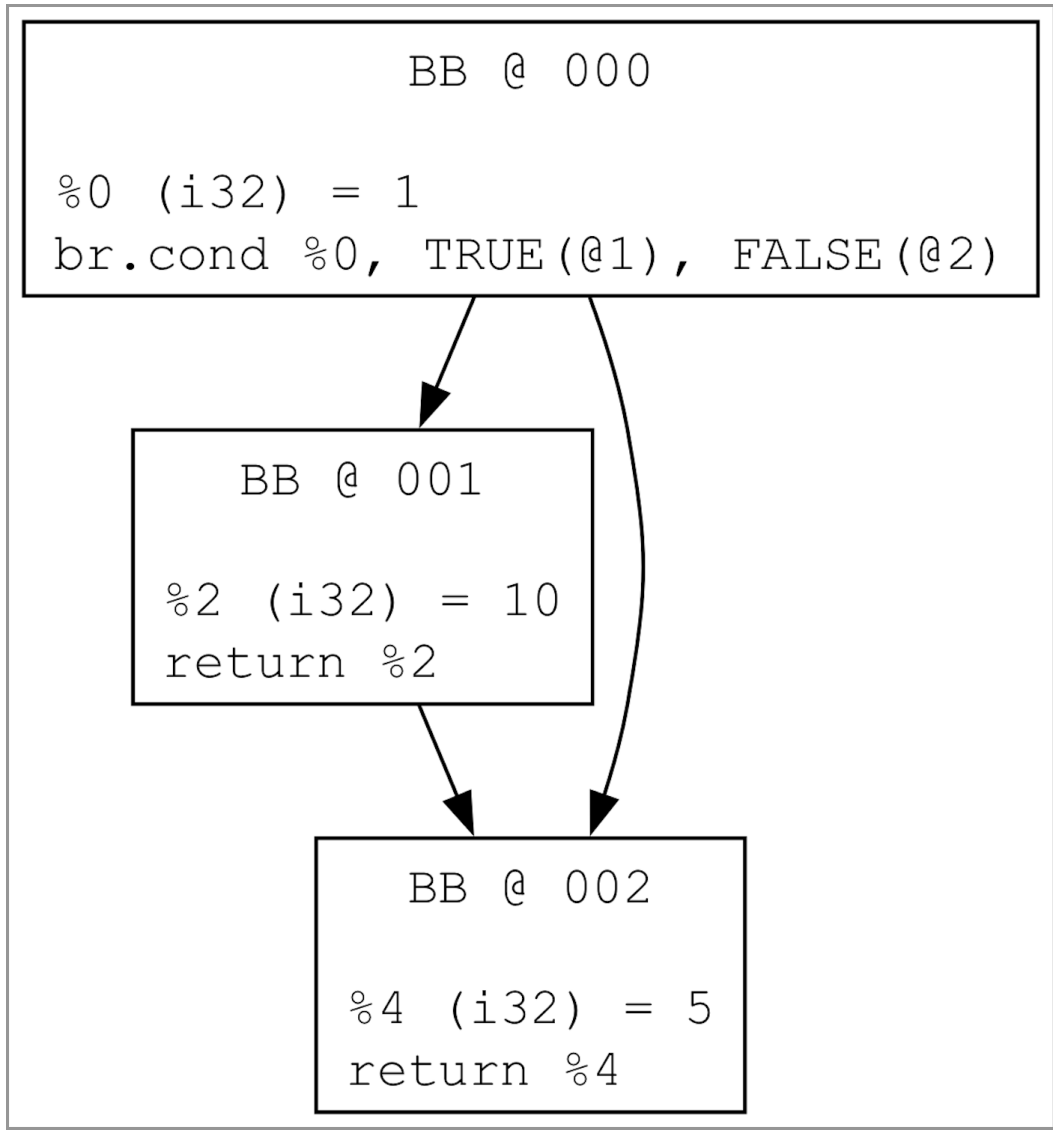
```c
// set of ops with no SEQ_POINTs
struct ir_stmt {
  size_t id;

  // a NULL bb means a pruned stmt
  struct ir_basicblock *basicblock;

  struct ir_stmt *pred;
  struct ir_stmt *succ;

  // the links between ops (`pred` & `succ`) have no significance to the
  // compilation and are just for traversal. meaningful links between operations
  // are with in the op data, such as `ir_op->ret.value`, which points to the op
  // whos result is returned
  struct ir_op *first;

  // last is the dominating op of the statement, and can be used as its "root".
  // all other ops in the statement are reachable from it
  struct ir_op *last;
};
```

ir_basicblock

```c
struct ir_basicblock {
  size_t id;

  // `value` contains a `struct vector *` containing the last op(s) that wrote
  // to this variable or NULL if it is not yet written to
  struct var_table var_table;

  // a NULL irb means a pruned basicblock
  struct ir_builder *irb;

  // these are creation order traversal methods, and do not signify edges
  // between BBs
  struct ir_basicblock *pred;
  struct ir_basicblock *succ;

  struct ir_stmt *first;
  struct ir_stmt *last;

  struct ir_basicblock **preds;
  size_t num_preds;

  enum ir_basicblock_ty ty;
  union {
    struct ir_basicblock_merge merge;
    struct ir_basicblock_split split;
  };

  // how many ops are before this block in the function
  size_t function_offset;

  void *metadata;
};
```

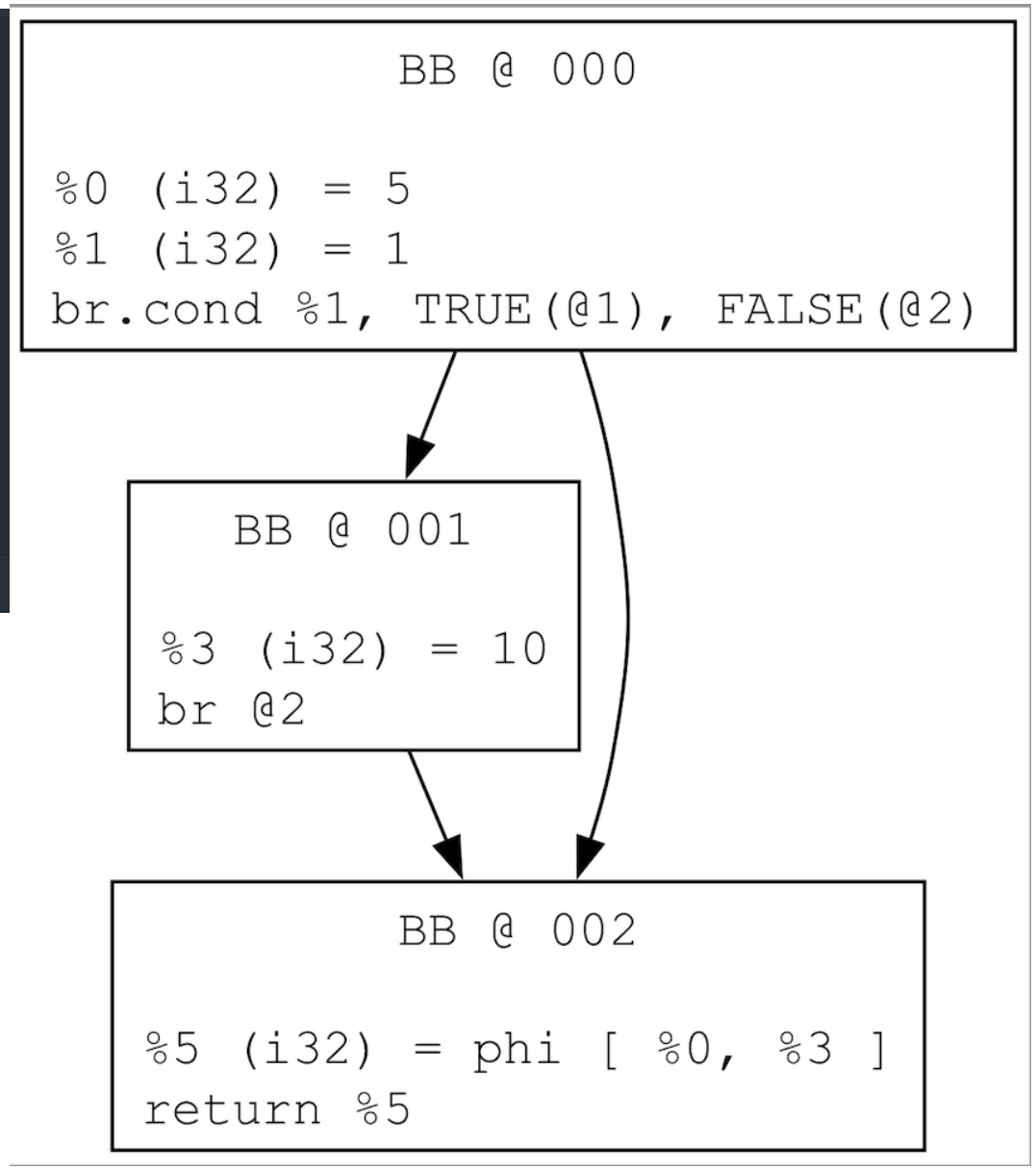
GraphViz

```
1  // expected value: 10
2  int main() {
3    if (1) {
4      return 10;
5    }
6
7    return 5;
8  }
~
```



Something a bit more interesting…

```c
// expected value: 10
int main() {
  int a = 5;

  if (1) {
    a = 10;
  }

  return a;
}
```

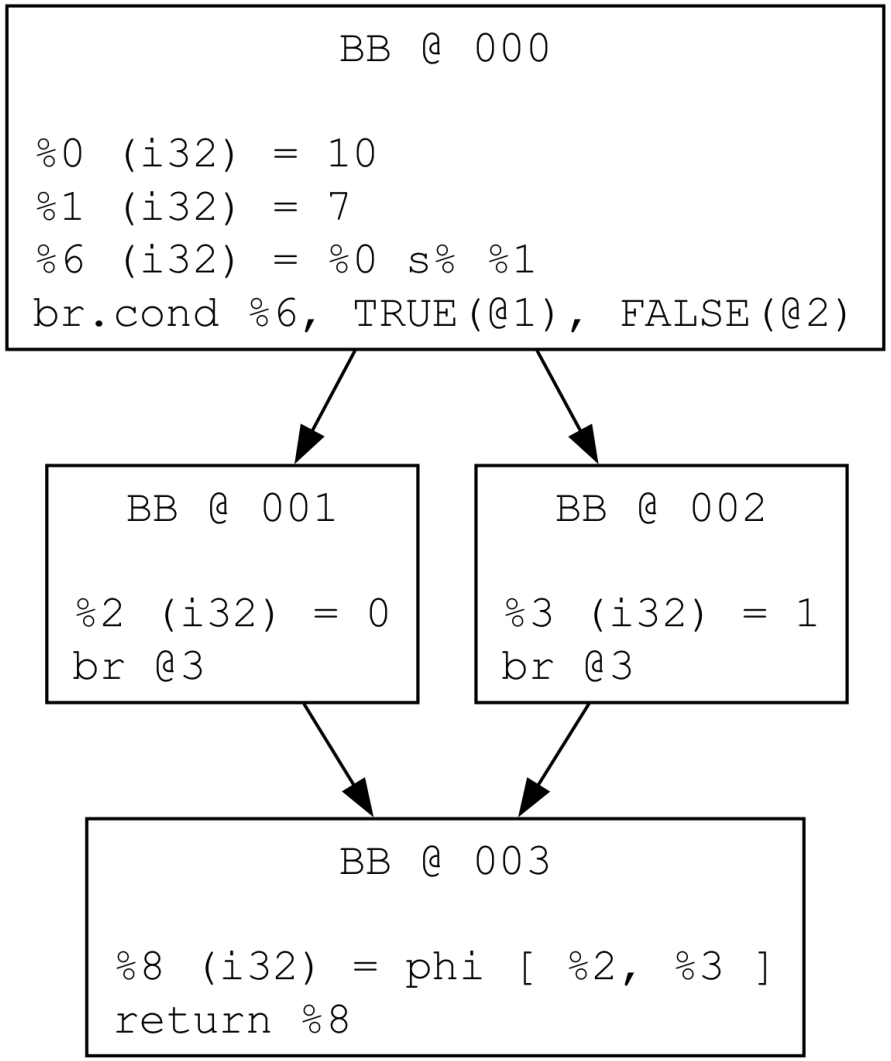


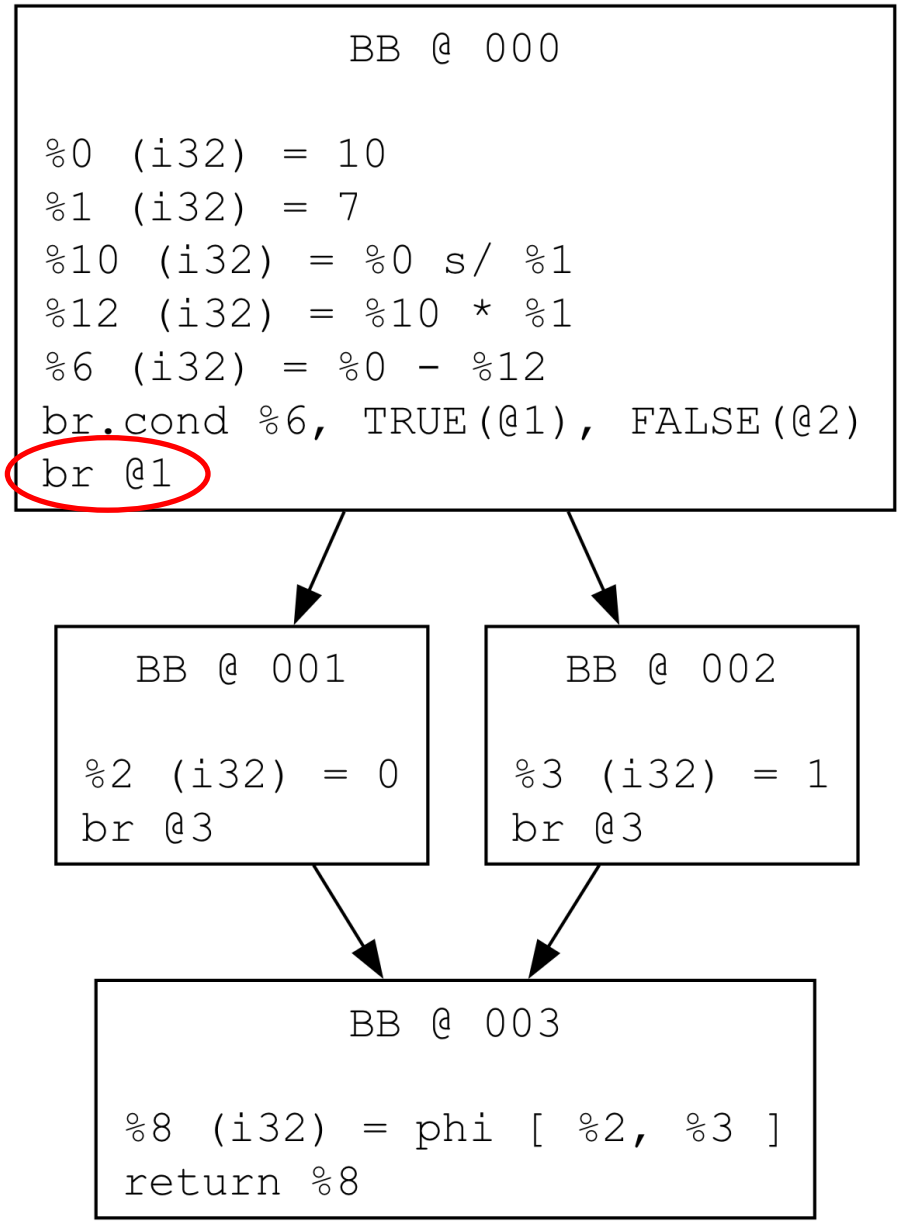
Lowering

- Performs platform-specific changes required before we can emit
  - In the future, also optimisations
- Still generates platform-independent IR
- This allows lowering to occur and for us to *then* re-run optimization passes
  - Ideal for performance
- Currently only a supports AArch64 platform
  - (Others in the works)

ARM64 has no modulo instruction!

```
int a = 10, b = 7;

int answer;

if (a % b) {
  answer = 0;
} else {
  answer = 1;
}

return answer;
}
```





RegAlloc

- As we saw earlier, our SSA representation generates a new named-value for every single expression in the program
  - A given SSA variable is SDMU – Single Definition Multiple Use
- In real life, we do not have an infinite number of registers, so we must break this SSA form and assign each value a register from a fixed set
- Currently, we use a linear-scan approach to register interval
  - A graph-coloring based register allocator is my final plan and is currently in the works

RegAlloc (cont.)

- The register allocator works via a two-pass allocation approach:
  - This pass assigns "trivial" registers and marks which variables need spilling
- In first pass, we insert `storelcl` and `loadlcl` instructions with appropriate locals for variables marked as `REG_SPILLED`
- We then *re*-run allocation, which will assign all registers
- Because store/load pairs are always directly around their use-sites, we guarantee that the second allocation run will never generate new spills
- The register allocator is platform-agnostic
  - It has an upper-bound on how many registers it may allocate, and it is up to late stages of the pipeline to transform this index into an actual register
- Floating-point instructions are not yet generated by JCC, but they will be handled without major changes to the regalloc – a second pass will be run for instructions that need to read/write from FP registers

Emitting

- Once all lowering and register allocation has occurred, the program is ready to be emitted into machine code
- It is still in SSA, although a slightly weaker version of it often called *degenerate SSA*
  - It retains all "real" properties of SSA but there are now instructions such as moves & stores that cannot be moved around as they could in normal SSA, due to the register allocation
- We emit the IR directly into AArch64 machine code, rather than assembly
  - Relatively simple emitter as AArch64 is fixed size

Object-file & Linking

- Final stages – build an object file from your machine code generated by each function, and link them together
- Object file building is currently hand-written for the Mach-O object file format used by macOS
  - Work-in-progress on ELF (Linux) and PE (Windows) object files
- Mach-O object file builder:
  - Takes the machine code the compiled functions + a set of symbols to export (such as the `` `_main` `` function)
- Linking is handled by the system linker
  - Still executed by compiler

Lessons Learnt: Memory Management

```c
#ifndef ALLOC_H
#define ALLOC_H

#include <stdlib.h>

/* Allocator which progressively grows and is freed in one go */
struct arena_allocator;

void arena_allocator_create(struct arena_allocator **allocator);
void arena_allocator_free(struct arena_allocator **allocator);

/* Alloc word-aligned block in arena */
void *arena_alloc(struct arena_allocator *allocator, size_t size);

/* Alloc space necessary for `str` (including null-terminator), then copy it into that spa
   Returns the start of the new allocation */
void *arena_alloc_strcpy(struct arena_allocator *allocator, const char *str);

/* Try and expand the allocation at `ptr` to `size`.
     - If `ptr` is `NULL`, acts the same as calling `arena_alloc(allocator, size)`
     - If `size` is less than the allocation size, this method does nothing */
void *arena_realloc(struct arena_allocator *allocator, void *ptr, size_t size);

#endif
```

Lessons Learnt: Memory Management

- Using arena-style allocation has many advantages
  - Performance – prevents lots of `malloc`/`free` calls
  - Simplicity – allows tying lifetimes of allocations to the compiler and not having to worry about them
  - Robustness – lifetime tying means memory leaks are much harder to cause
- Has downsides too!
  - Principally, higher memory usage

Lessons Learnt: C-style encapsulation

- Create-free pattern with pseudo-instance methods
- Opaque pointer which consumer can only pass around
  - Helps keep everything together
  - Keeps all access to a given struct within one file