COMPILER CONSTRUCTION · WEEK 5 · STATION II

Expressions

Code generation for literals, terms and arithmetic; registers as a stack; and encoding an instruction, which is Gödel numbering made mechanical.

THREE PROBLEMS

Where code goes. Where data goes. Which register.

CODE

Straight code generation: emit each instruction as soon as it is known, one after another, into `code_binary`; `code_size` grows by four per instruction. When a target address is not yet known, remember the spot and *fix it up* later. Next week.

DATA

Strings, big integers and global variables go into `data_binary`, in reverse: the first word is at offset −8 from `gp`, the next at −16. The data segment is static allocation at compile time.

REGISTERS

Seven temporaries, `t0` to `t6`, allocated as a stack at compile time: `talloc`, `tfree`, `current_temporary`, `previous_temporary`. Knowing the two most recent is enough for every situation selfie meets.

A stack allocator at compile time, for registers used at runtime. The two timelines again — and the assertions on `allocated_temporaries` in the comments are how the compiler keeps itself from leaking registers.

LITERALS

Emitting a literal: one `addi`, or a word in the data segment and its address.

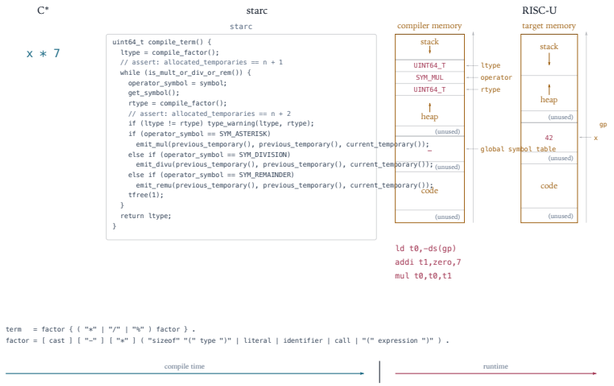
`load_integer` · `load_string`

A small integer fits in the 12-bit immediate of one `addi t0,zero,85`. A medium one needs a `lui` and an `addi`. A big one goes into the data segment like a string and is loaded from there.

A string is emitted into the data segment, entered into the symbol table so that it is reused if it occurs again, and its *address* is what the code loads: `addi t0,zero,−ds` then `add t0,gp,t0`. The value of a string literal is a pointer.

TERMS

Infix in, postfix out: both factors in temporaries, then one instruction.



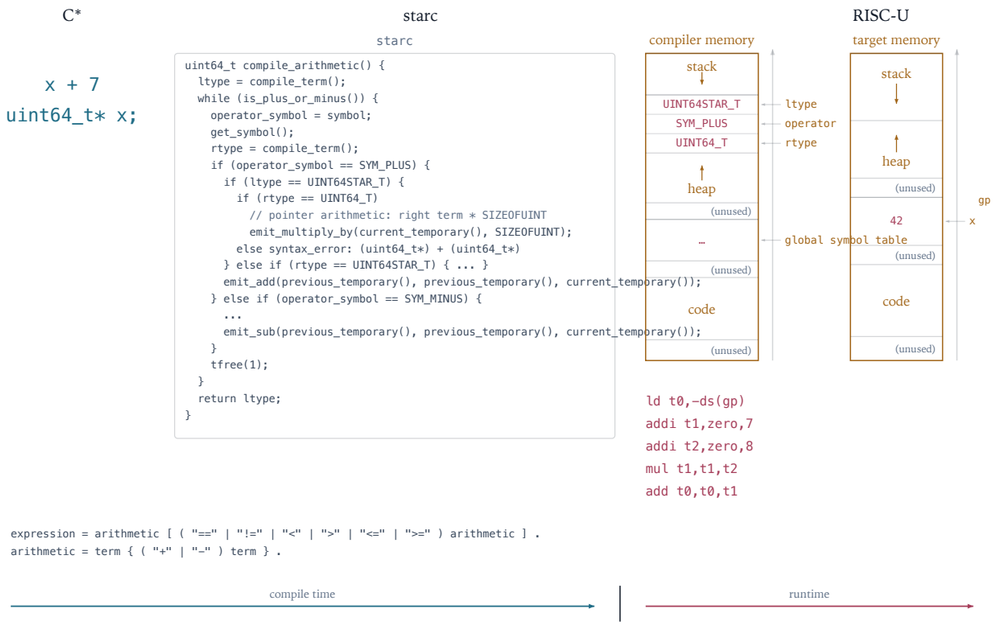
compile_term · x * 7

`compile_term` calls `compile_factor`, which leaves the value in a fresh temporary. Then while the symbol is `*`, `/` or `%`: remember the operator, get the next symbol, call `compile_factor` again, emit `mul`, `divu` or `remu` on the previous and current temporaries — and free one.

The infix operator is remembered across the second operand and emitted after it. That is the translation from the notation for humans to the notation for the machine, and the register stack is what makes it one instruction.

ARITHMETIC AND COMPARISON

Pointer arithmetic is scaled. A comparison is a subtraction.



x + 7 with x a pointer: multiply by 8 first

x == 7 as unsigned 7 - x < 1

ENCODING

An instruction is a number. Making it one is Gödel numbering, by hand.

```
uint64_t encode_i_format(uint64_t immediate, uint64_t r
uint64_t funct3, uint64_t rd, uint64_t opcode) {
// assert: -2^11 <= immediate < 2^11
check_immediate_range(immediate, 12);
immediate = sign_shrink(immediate, 12);
return left_shift(left_shift(left_shift(left_shift(
immediate, 5) + rs1, 3) + funct3, 5) + rd, 7) + opcode;
}
void emit_addi(uint64_t rd, uint64_t rs1, uint64_t imme
emit_instruction(encode_i_format(immediate, rs1, F3_ADD
}
```

Twelve bits of immediate, five of source register, three of function, five of destination, seven of opcode: shift and add, and `addi t1,zero,7` is `0x00700313`. The emulator's decoder does the same arithmetic backwards, and both are in one file, so the encoding is defined once.

This is the moment the compiler turns notation into a number that a machine can compute with. The introduction class called it Gödelisierung and made a theorem of it; here it is thirteen lines and a shift.

ELF

And the binary is a file that real hardware runs.

```
$ ./selfie -c selfie.c -o selfie.m && od -A d -t x1 sel
0000000 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
$ spike pk selfie.m
selfie.m { -c { source } | -o binary | … }
```

`-o` writes an ELF file: a header that says where code and data go, then `code_binary`, then `data_binary`. The same bytes load into mipster, into spike, and onto a RISC-V board.

Everything on this deck happened at compile time and produced a number. Everything that number does happens at runtime, on a machine that has never seen the source. That separation is the whole architecture of a compiler.

ASSIGNMENT

`bitwise-shift-execution`: `sll` and `srl`, in the ISA, the emulator, and the code generator.

1. Specify. Extend `riscu.md` with `sll rd,rs1,rs2` and `srl rd,rs1,rs2`: assembly, encoding as R-format instructions, and semantics in the style of the other lines.
2. Encode and decode. An `emit_sll` and `emit_srl` using `encode_r_format`, and the matching cases in the decoder and the disassembler.
3. Execute. The two instructions in mipster, using C*'s own << and >> — which do not exist in C* until your compiler from last week exists. Self-reference, as an assignment.
4. Generate. In last week's `compile_X`, emit them. Then `./grader/self.py bitwise-shift-execution` and `self-compile`.

RULES

Only riscu.md and selfie.c. No warnings. Note the bootstrap: to use shifts in selfie.c, selfie must first be compiled by a C compiler that has them, and then by itself.