Where we are

Time	Details	Type
9:00 - 9:05	Introduction, welcome and objectives	Presentation
9:05 - 9:25	An overview of MLIR and LLVM	Presentation
9:25 - 9:55	The xDSL framework	Presentation
9:55 - 10:00	Introduction to the hands-on activity	Presentation
10:00 - 10:15	Logging into ARCHER2 and start hands-on practical activity	Hands-on
10:15 - 10:30	Morning break	
10:30 - 10:35	Welcome back and overview of second part	Presentation
10:35 - 12:10	Continue hands-on practical activity	Hands-on
12:10 - 12:20	Wash up from practical activities, highlighting key take-away points	Presentation
12:20 - 12:30	Conclusions & next steps to continue working with the technologies	Presentation

Wrap up of exercise one

- Hopefully you have managed to run exercise one by now and have an executable that runs on ARCHER2

1. The decorator *@python_compile in our python_compiler* module drives translation of the Python code into our *tiny py* dialect
   - This is the entry point into MLIR
2. We run the *tiny-py-to-standard* transformation to lower this into the standard dialects
3. This IR is passed to mlir-opt which converts it into LLVM IR
4. LLVM IR is compiled by clang into an executable

Single Static Assignment (SSA) form

Retrieves the address and pointer of our string

Calls the printf function with our string as an argument

```
"builtin.module"() ({
  "func.func"() ({
    %0 = "llvm.mlir.addressof"() {"global_name" = @str0} : () -> !llvm.ptr<!llvm.array<13 x i8>>
    %1 = "llvm.getelementptr"(%0) {"rawConstantIndices" = array<i32: 0, 0>} : (!llvm.ptr<!llvm.array<13 x i8>>) -> !llvm.ptr<i8>
    "func.call"(%1) {"callee" = @printf} : (!llvm.ptr<i8>) -> ()
    "func.return"() : () -> ()
  }) {"sym_name" = "main", "function_type" = () -> (), "sym_visibility" = "public"} : () -> ()
  "llvm.mlir.global"() ({
  }) {"global_type" = !llvm.array<13 x i8>, "sym_name" = "str0", "linkage" = #llvm.linkage<"internal">, "addr_space" = 0 : i32,
"constant", "value" = "Hello world!\n", "unnamed_addr" = 0 : i64} : () -> ()
  "func.func"() ({
  }) {"sym_name" = "printf", "function_type" = (!llvm.ptr<i8>) -> (), "sym_visibility" = "private"} : () -> ()
}) : () -> ()
```

Signature definition of printf function

The string that we will print

xDSL

Passing to MLIR and LLVM

mlir-opt --convert-func-to-llvm ex_one.mlir | mlir-translate -mlir-to-llvmir | clang -x ir -o test -

- The *mlir-opt* tool will apply MLIR transformations
  - Here we need to lower the *func* dialect to the *llvm* dialect
- The *mlir-translate* tool will generate LLVM-IR from MLIR
  - But this MLIR needs to be fairly low level, for instance the *llvm* dialect is supported whereas the *func* dialect is not and hence we need to lower before passing to this tool
    - In exercise two we will need to undertake additional transformations
- Passing the generated to LLVM-IR to *clang* which will produce the executable

Going forwards

- If there are any questions or issues with the first exercise then let us know
  - It is no problem if you are still working on this

- Exercises two and three will explore the concepts in more detail
  - In exercise two we extend the subset of Python that we support by making changes to our dialect and the *tiny-py-to-standard* transformation
  - In exercise three we will target parallelism by developing a transformation pass that replaces our loop of exercise two with a parallel loop and then use MLIR transforms to lower this to OpenMP and the vector dialect