TableGen Formatter

Extending Clang-Format Capabilities

Himanshu Shishir Shah
Venkat Nikhil Tatavarthy
Introduction

- The TableGen framework plays a crucial role in both LLVM and MLIR.
Need for TableGen Formatter

- TableGen currently lacks a dedicated code formatting tool, which poses challenges for maintaining and reading the code.
- A code formatter ensures consistent and readable code, fostering collaboration and ease of maintenance.
Need for TableGen Formatter

- TableGen currently lacks a dedicated code formatting tool, which poses challenges for maintaining and reading the code.

- A code formatter ensures consistent and readable code, fostering collaboration and ease of maintenance.

```python
def FeatureFMV : SubtargetFeature<"fmv", "HasFMV", "true", "Enable Function Multi Versioning support.">;

def FeatureZCRegMove : SubtargetFeature<"zcm", "HasZeroCycleRegMove", "true", "Has zero-cycle register moves">;
```

Inconsistent def record formatting
Need for TableGen Formatter

- TableGen currently lacks a dedicated code formatting tool, which poses challenges for maintaining and reading the code.

- A code formatter ensures consistent and readable code, fostering collaboration and ease of maintenance.

```perl
foreach vt = [v2f16, v2bf16] in {
    def: Pat<(vt (ProxyReg vt:$src)), (ProxyRegI32 Int32Regs:$src)>;
}

foreach mma = !listconcat(MMAs, WMMAs, MMA_LDSTs, LDMATRIXs) in
    def : MMA_PAT<mma>;
```

Inconsistent foreach loop formatting
Considered Approaches

- Approach 1: Pull out relevant libraries from Clang-Format
  - Can be built independently as a new tool.
  - Difficult to extract common files to be shared between Clang-Format and TableGen Formatter.
Considered Approaches

■ Approach 1: Pull out relevant libraries from Clang-Format
  ❑ Can be built independently as a new tool.
  ❑ Difficult to extract common files to be shared between Clang-Format and TableGen Formatter.

■ Approach 2: Build everything from scratch
  ❑ Can be built independently as a new tool.
  ❑ This introduces code duplication and additional effort to maintain both tools.
Considered Approaches

■ Approach 1: Pull out relevant libraries from Clang-Format
  - Can be built independently as a new tool.
  - Difficult to extract common files to be shared between Clang-Format and TableGen Formatter.

■ Approach 2: Build everything from scratch
  - Can be built independently as a new tool.
  - This introduces code duplication and additional effort to maintain both tools.

■ Approach 3: Adding TableGen support in Clang-Format
  - Reusing current infrastructure as there are similarities between C++ and TableGen, needing to only focus on the differences.
Considered Approaches

- **Approach 1: Pull out relevant libraries from Clang-Format**
  - Can be built independently as a new tool.
  - Difficult to extract common files to be shared between Clang-Format and TableGen Formatter.

- **Approach 2: Build everything from scratch**
  - Can be built independently as a new tool.
  - This introduces code duplication and additional effort to maintain both tools.

- **Approach 3: Adding TableGen support in Clang-Format**
  - Reusing current infrastructure as there are similarities between C++ and TableGen, needing to only focus on the differences.
Implementation
Implementation
Implementation

```
Clang-Format

Lexer
  - Format Tokens
  - Format Token Lexer
  - Merge Tokens
    Merging tokens to create new tokens like ladd, lif, etc.

TableGen Keywords

Unwrapped Line Parser

Token Annotator

Token Analyzer

Unwrapped Line Formatter

Replacements

Spacing Manager
  - Whitespace Manager
  - Definition Block Separator
  - Continuation Indenter
```
Implementation
Implementation

Clang-Format

Lexer
- Format Tokens
- Format Token Lexer
- Merge Tokens

TableGen Keywords

Unwrapped Line Parser
- Unwrapped Line Formatter

Token Annotator
- Determine Formatting rule

Token Analyzer
- Replacements

Spacing Manager
- Whitespace Manager
- Definition Block Separator
- Continuation Indenter

Determining the formatting rules to apply based on the token semantics (e.g., using 'if' as a concat operator)
Implementation
Implementation

Clang-Format

Lexer
- Format Tokens
- Format Token Lexer
- Merge Tokens

TableGen Keywords

Unwrapped Line Parser

Token Annotator
- Determine Formatting rule

Token Analyzer

Unwrapped Line Formatter
- Replacements

Clang-Format style options

Spacing Manager
- Whitespace Manager
- Definition Block Separator
- Continuation Indenter

Indentation and spacing rules for TableGen constructs

loops, conditionals, def records, etc.
Implementation
Testing the Formatter

- We ran the formatter on all the TableGen files under LLVM project.
- Following the formatting process, we ran all the LLVM regression tests using the `check-llvm` target.
- Overall, we successfully formatted 802 files of TableGen code without breaking the build.
Example: Recognizing TableGen keywords

```python
def CARRY: SPR<1, "xer">, DwarfRegNum<76> {  
    let Aliases = [XER];
}
multiclass DIV_Common <InstR600 recip_ieee> {  
    def : R600Pat<  
        (fdiv f32:$src0, f32:$src1),  
        (MUL_IEEE $src0, (recip_ieee $src1))  
    >;
}

let Predicates = [HasSSE3] in  
    def rx : Instruction<opc, "rx">;
```
Example: Recognizing TableGen keywords

```python
def CARRY: SPR<1, "xer">, DwarfRegNum<76> {  
    let Aliases = [XER];
}

multiclass DIV_Common <InstR600 recip_ieee> {
    def : R600Pat<
        (fdiv f32:$src0, f32:$src1),
        (MUL_IEEE $src0, (recip_ieee $src1))
    >;
}

let Predicates = [HasSSE3] in {
    def rx : Instruction<opc, "rx">;
}
```
Example: Recognizing TableGen keywords

```cpp
def CARRY: SPR<1, "xer">, DwarfRegNum<76> {  
    let Aliases = [XER];
}

multiclass DIV_Common <InstR600 recip_ieee> {
    def : R600Pat<
        (fdiv f32:$src0, f32:$src1),
        (MUL_IEEE $src0, (recip_ieee $src1))
    >;
}

let Predicates = [HasSSE3] in 
    def rx : Instruction<opc, "rx">;

SeparateDefinitionBlocks: true

InsertBraces: true

```
Example: Parsing loops and conditional statements

```python
foreach Index = 32-63 in {
    def VSX : VSXReg<Index, "vs">;
}

if P.HasExtSDWA then {
    def : MnemonicAlias<opName # "_sdwa", opName>;
}
```

RemoveBracesLLVM: true

```python
foreach Index = 32 - 63 in
    def VSX : VSXReg<Index, "vs">;

if P.HasExtSDWA then
    def : MnemonicAlias<opName#"_sdwa", opName>;
```
Example: Support for existing Clang-Format options

```cpp
class C<int x> {
    int Y = x;
    int Yplus1 = add( Y, 1 );
    int xplus1 = add( x, 1 );
}

def imm16_31 : ImmLeaf<i32, [{
    int Y = x;
    int Yplus1 = Y + 1;
    int xplus1 = x + 1;
    return Yplus1 >= 0;
}];
```
Example: Support for existing Clang-Format options

```cpp
class C<int x> {
    int Y = x;
    int Yplus1 = !add( Y, 1 );
    int xplus1 = !add( x, 1 );
}

def imm16_31 : ImmLeaf<i32, [{
    int Y = x;
    int Yplus1 = Y + 1;
    int xplus1 = x + 1;
    return Yplus1 >= 0;
}]>;```

AlignConsecutiveAssignments: Consecutive

```cpp
class C<int x> {
    int Y = x;
    int Yplus1 = !add( Y, 1 );
    int xplus1 = !add( x, 1 );
}

def imm16_31 : ImmLeaf<i32, [{
    int Y = x;
    int Yplus1 = Y + 1;
    int xplus1 = x + 1;
    return Yplus1 >= 0;
}]>;```
Example: Support for existing Clang-Format options

class C<int x> { 
    int Y = x;
    int Yplus1 = !add( Y, 1 );
    int xplus1 = !add( x, 1 );
}

def imm16_31 : ImmLeaf<i32, [{{
    int Y = x;
    int Yplus1 = Y + 1;
    int xplus1 = x + 1;
    return Yplus1 >= 0;
}]}>

AlignConsecutiveAssignments: Consecutive

class C<int x> { 
    int Y = x;
    int Yplus1 = !add( Y, 1 );
    int xplus1 = !add( x, 1 );
}

def imm16_31 : ImmLeaf<i32, [{{
    int Y = x;
    int Yplus1 = Y + 1;
    int xplus1 = x + 1;
    return Yplus1 >= 0;
}]}>

AlignConsecutiveAssignments: Consecutive

X
Future Works

- Formatting of multi-line string literals between ‘[{“ and ‘}]’ (*TokCode*).
- Adding support for other TableGen keywords and constructs such as *defset*, *defm*, etc.
- Adding support for the remaining relevant Clang-Format Style Options.
Acknowledgement

- Dr. Min-Yih Hsu (@mshockwave) for providing guidance and support throughout the project.
Thank you