

Assembler Framework

Directive represents a language supported directive mnemonic and what the assembler should do when encountering that mnemonic/directive

```
class ASMDirective {
+ mnemonic: string
# decimalPattern: const regex {static}
# hexPattern: const regex {static}
# asciiPattern: const regex {static}
# stringPattern: const regex {static}

+ Implementation(const Workpiece* const&, const vector<string>& const: size_t {virtual})
# ASMDirective(string&&)

template<std::integral InputType, std::integral TargetType>
# SerializeToBuffer(const InputType&, Workpiece* const&): void {static} {inline}
}
```

Instruction represents a language supported instruction mnemonic and what the assembler should do when encountering that mnemonic/instruction

```
class ASMInstruction {
+ mnemonic: string
# registerPattern: const regex {static}
# decimalPattern: const regex {static}
# hexPattern: const regex {static}
# asciiPattern: const regex {static}

+ Implementation(vector<uint8_t>&, const Workpiece* const&, const vector<string>& const: size_t {virtual})
# ASMInstruction(string&&)

# GetLabelOffset(const string&, Workpiece* const&): uint64_t& {static} {inline}
# AddUnresolvedLabel(const string&, vector<uint8_t>&, Workpiece* const&): void {static} {inline}

template<std::integral TargetType, std::integral... InputType>
# GetRegisterID(const string&, const TargetType&, const TargetType&): TargetType {static} {inline}

template<std::integral TargetType>
# GetImmediateValue(const string&): TargetType {static} {inline}
}
```

Language() Directives and Instructions store instances of Directive and Instruction respectively. These hash tables are searched by a pass to allow for the machine code to be produced for a directive or instruction, or to verify the existence of a mnemonic

```
class LanguageDefinition {
# m_languageDirectives: const unordered_map<string, const ASMDirective*>
# m_languageInstructions: const unordered_map<string, const ASMInstruction*>
# m_reservedKeywords: const unordered_set<string>

# LanguageDefinition()
# ~LanguageDefinition()

# SetDirectives(const DerivedFromDirective auto&...): void
# SetInstructions(const DerivedFromInstruction auto&...): void
# SetKeywords(StringType auto&...): void

+ LanguageDefinition(const LanguageDefinition&) {delete}
+ operator=(const LanguageDefinition&) {delete}
+ LanguageDefinition(LanguageDefinition&&) {delete}
+ operator=(LanguageDefinition&&) {delete}
}
```

```
class SingletonWrapper {
# s_instance: shared_ptr<Derived> {static}
# _sharedMutex: shared_mutex {static}

+ GetInstance(): shared_ptr<Derived> {static}
+ Shutdown(): void {virtual}

+ SingletonWrapper(const SingletonWrapper&) {delete}
+ operator=(const SingletonWrapper&) {delete}
+ SingletonWrapper(SingletonWrapper&&) {delete}
+ operator=(SingletonWrapper&&) {delete}
}
```

Note: SetDirectives uses concept (c++ 20) called DerivedFromDirective to create a variadic parameter to allow passing in any number of Directive(s) as constructor arguments. DerivedFromDirective is defined as follows.

```
template<typename T>
concept DerivedFromDirective =
std::derived_from<T, ASMDirective>;
```

Note: SetInstructions uses concept (c++ 20) called DerivedFromInstruction to create a variadic parameter to allow passing in any number of Instruction(s) as constructor arguments. DerivedFromInstruction is defined as follows.

```
template<typename T>
concept DerivedFromInstruction =
std::derived_from<T, ASMInstruction>;
```

Note: SetKeywords uses concept (c++ 20) called StringType to create a variadic parameter to allow passing in any number of Instruction(s) as constructor arguments. StringType is defined as follows.

```
template<typename T>
concept StringType = std::is_same_v<T,
Instruction>;
```

All concrete "Instructions" are contained within *LanguageDefinition_4380*

All concrete "Directives" are contained within *LanguageDefinition_4380*

The Assembler can be thought of as an assembly line. It contains a collection of "Passes" that work upon the "Workpiece" until all the work needing to be done has been completed and the finished product has emerged

Note: Assembler derived types will be singleton, deriving also from SingletonWrapper

```
class Assembler {
# m_passes: const vector<Pass*>
# m_workpiece: Workpiece*
# m_filePath: const filesystem::path
# _languageSpec: const LanguageDefinition*

# Assembler(Workpiece*, const LanguageDefinition*, const DerivedFromPass auto*...)
# ~Assembler()

+ ProcessASM(const char*): void {virtual}

+ Assembler(const Assembler&) {delete}
+ operator=(const Assembler&) {delete}
+ Assembler(Assembler&&) {delete}
+ operator=(Assembler&&) {delete}
}
```

Note: Constructor uses concept (c++ 20) called DerivedFromPass to create a variadic parameter to allow passing in any number of Pass(s) as constructor arguments. DerivedFromPass is defined as follows.

```
template<typename T>
concept DerivedFromPass =
std::derived_from<T, Pass>;
```

The Workpiece is the program under assembly. It can be thought of as a "workpiece" traveling through an assembly line from start to finish. Each pass performing some work on the piece and passign it on to the next stage of the line

```
class Workpiece {
+ _sharedMutex: shared_mutex

+ _symbolTable: unordered_map<string, uint64_t>
+ _unresolvedLabels: unordered_map<string, vector<void*>>
+ _dataSegmentItems: vector<std::pair<string, DataSegmentItem>>
+ _dataSegmentBin: vector<uint8_t>
+ _codeSegmentItems: vector<std::pair<string, vector<CodeSegmentItem>>>
+ _codeSegmentBins: list<std::pair<string, vector<uint8_t>>>

+ _lineNum: size_t
+ _directive: const ASMDirective*
+ _dirArgs: vector<string>
+ _lineComment: string

+ _lineNum: size_t
+ _instruction: const ASMInstruction*
+ _instrArgs: vector<string>
+ _lineComment: string
}
```

A Pass should be thought of as a station, or section, of an assembly line to perform some work on the workpiece. This can be as granular as needed

```
class Pass {
+ Execute(Workpiece* const&, const filesystem::path&, const LanguageDefinition* const&) const: void {virtual}
}
```

All "Passes" are realizations of *Pass*

Passes

```
class PassOne_Tokenization {
+ Execute(Workpiece* const&, const filesystem::path&, const LanguageDefinition* const&) const: void {override}

- AddLabel(const string&, Workpiece* const&)const: void
- ProcessDataSegment(size_t&, Workpiece* const&, ifstream&, const LanguageDefinition* const&) const: void
- HandleDirective(const size_t&, string&&, const string&, const vector<string>&, string&&, Workpiece* const&, const LanguageDefinition* const&) const: void
- ProcessCodeSegment(size_t&, Workpiece* const&, ifstream&, const LanguageDefinition* const&) const: void
- HandleInstruction(const size_t&, const string&, const vector<string>&, string&&, vector<CodeSegmentItem>* const&, const LanguageDefinition* const&) const: void
- HandleCodeSegmentItem(const string&, const vector<CodeSegmentItem>&, Workpiece* const&) const: void
}
```

```
class PassTwo_Assemble {
- _fail: bool {mutable}

+ Execute(Workpiece* const&, const filesystem::path&, const LanguageDefinition* const&) const: void {override}

- ProcessDataSegment(size_t&, Workpiece* const&, const LanguageDefinition* const&) const: void
- ProcessCodeSegment(size_t&, Workpiece* const&, const LanguageDefinition* const&) const: void
- ProcessUnresolvedLabels(Workpiece* const&) const: void
}
```

```
class PassThree_Serialize {
+ Execute(Workpiece* const&, const filesystem::path&, const LanguageDefinition* const&) const: void {override}

- SerializeInitialPC(ofstream&, Workpiece* const&) const: size_t
- SerializeDataSegment(ofstream&, Workpiece* const&) const: size_t
- SerializeCodeSegment(ofstream&, Workpiece* const&) const: size_t
}
```

All concrete "Passes" are contained within *ASM4380*

ASM_4380

```
class LanguageDefinition_4380 {
- all directives, and all instructions
}
```

```
class ASM4380 {
- _passOne: PassOne_Tokenization
- _passTwo: PassTwo_Assemble
- _passThree: PassThree_Serialize

+ _workpiece: Workpiece
+ _languageDef: LanguageDefinition_4380

+ _trapZero: bool {static}

+ Shutdown: void {override}
+ ProcessASM(const char*): void {override}
}
```

Directives

BYT_DIR
CSTR_DIR
INT_DIR
STR_DIR

Instructions

ADD_INS	ADI_INS	DIVL_INS	DIV_INS	MULI_INS
MUL_INS	SUB_INS	CMPL_INS	CMP_INS	ALCL_INS
ALLC_INS	FREE_INS	BGT_INS	BLT_INS	BNZ_INS
BRZ_INS	JMP_INS	JMR_INS	AND_INS	NOT_INS
OR_INS	LDA_INS	LDB_INS	LDR_INS	MOVI_INS
MOV_INS	STB_INS	STR_INS	BLK_INS	END_INS
LCK_INS	RUN_INS	ULK_INS	TRP_INS	