



Understanding
TableGen'erated files in
LLVM Backend

Prerona Chaudhuri,
GPU Compiler Backend Engineer

Who Am I ?

- GPU Compiler engineer for past 5 years at **NVIDIA** and
- A user of LLVM TableGen for past 3 years, who has used **TableGen for the LLVM backend.** 😊



This talk is NOT

Tutorial on how to write
TableGen code for backend

This talk is...

A beginner's guide, on how to navigate
through some of the
common & important C++ files
(.inc) generated from .td

Disclaimer

- Using **AArch64** backend as a reference/examples
- You might need to revisit the slides few times to appreciate what I mean, if you are ,especially new to TableGen and LLVM Backend. 😊

Motivation

- Helpful in debugging
 - Encoding & Decoding errors
 - Assembly printer errors
 - Isel errors (why a pattern not matching or empty HW type)
 - What files to search for specific information
- Helps one understand
 - Concrete C++ code generated from target description (TD).
 - Interaction/connection between TD and target-independent code generator APIs.

Introduction

- TableGen is a Domain Specific Language heavily used by the code-generators/target backend to represent variety of information in target backend like

IR
Encoding, asm string,
Operands(registers, predicates)

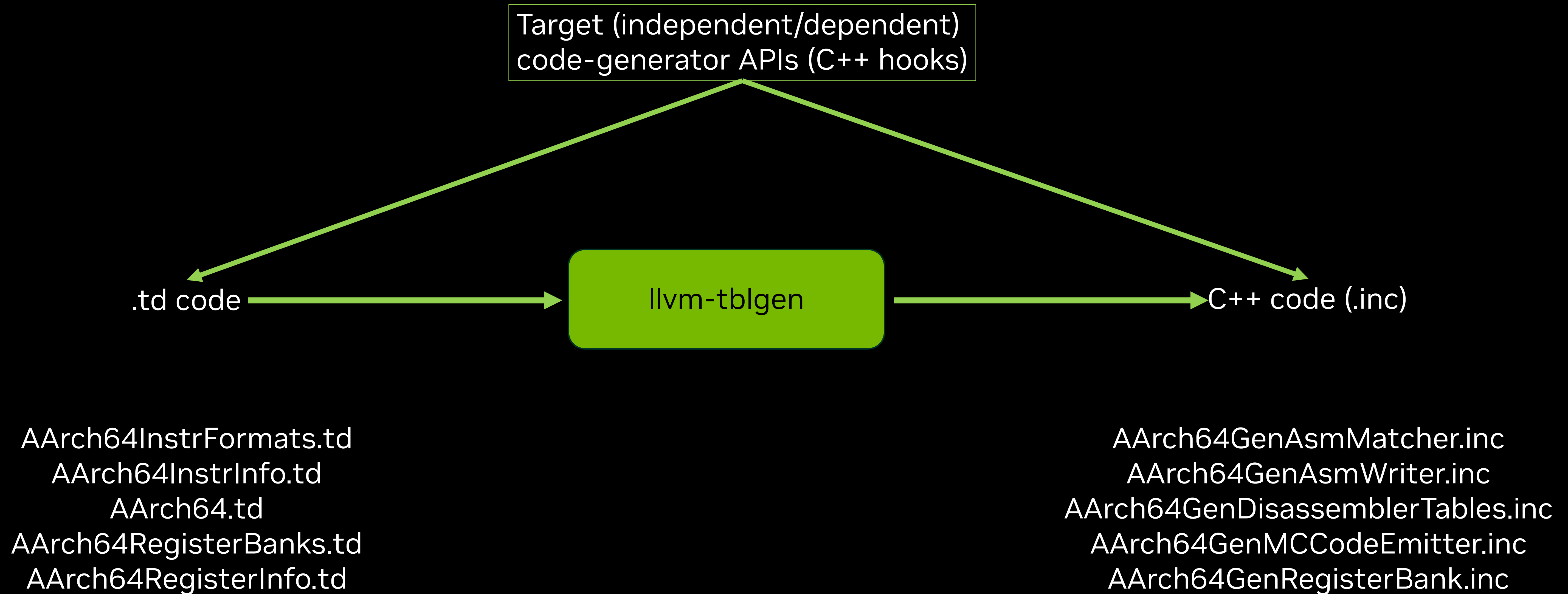
Optimize/Transform
Combiners to perform peephole,
convert GMIR/Selection Dag to
MIR

Target
Architecture, Features applicable
to target

- It offers the feature of generating **voluminous**, repetitive, tedious C++ code by writing relatively lesser and non-duplicated .td code.

The Flow

llvm-tblgen

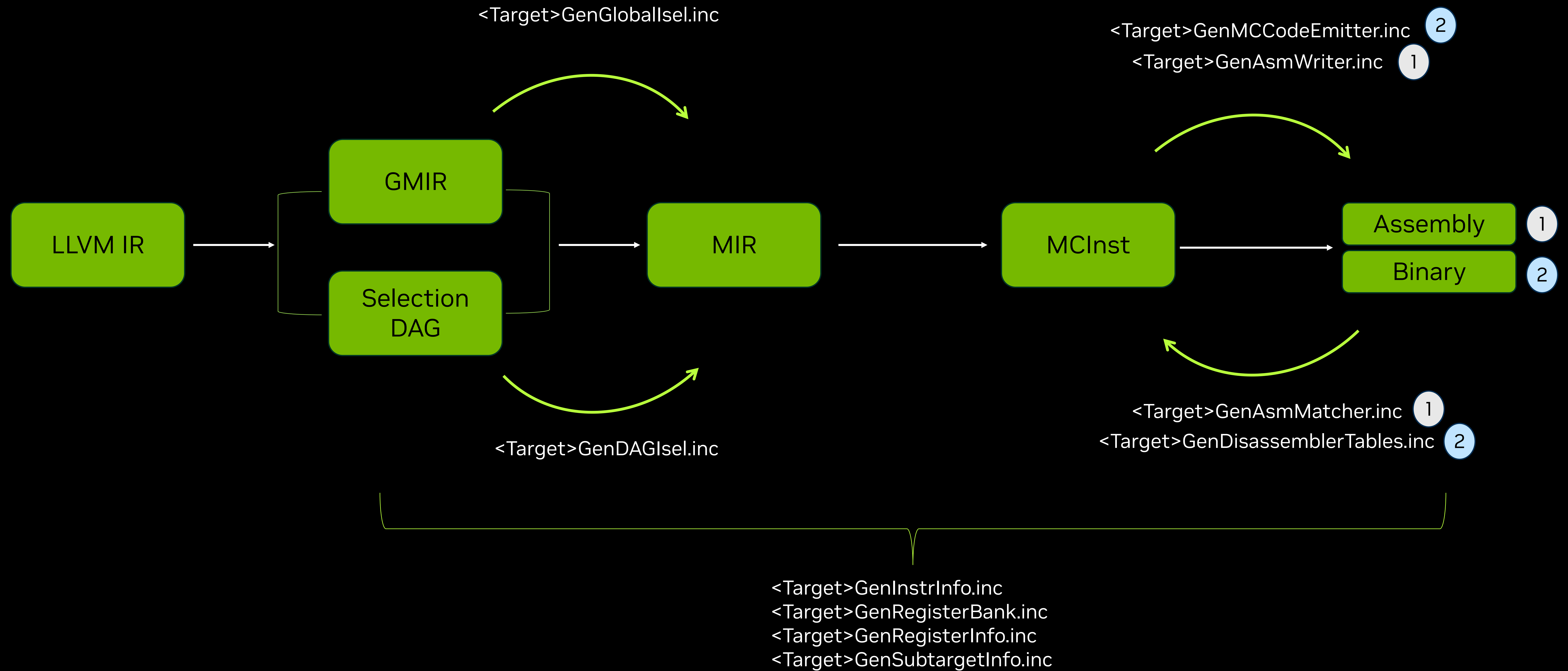


Path : llvm-project/llvm/lib/Target/AArch64

Path : \$build/lib/Target/AArch64

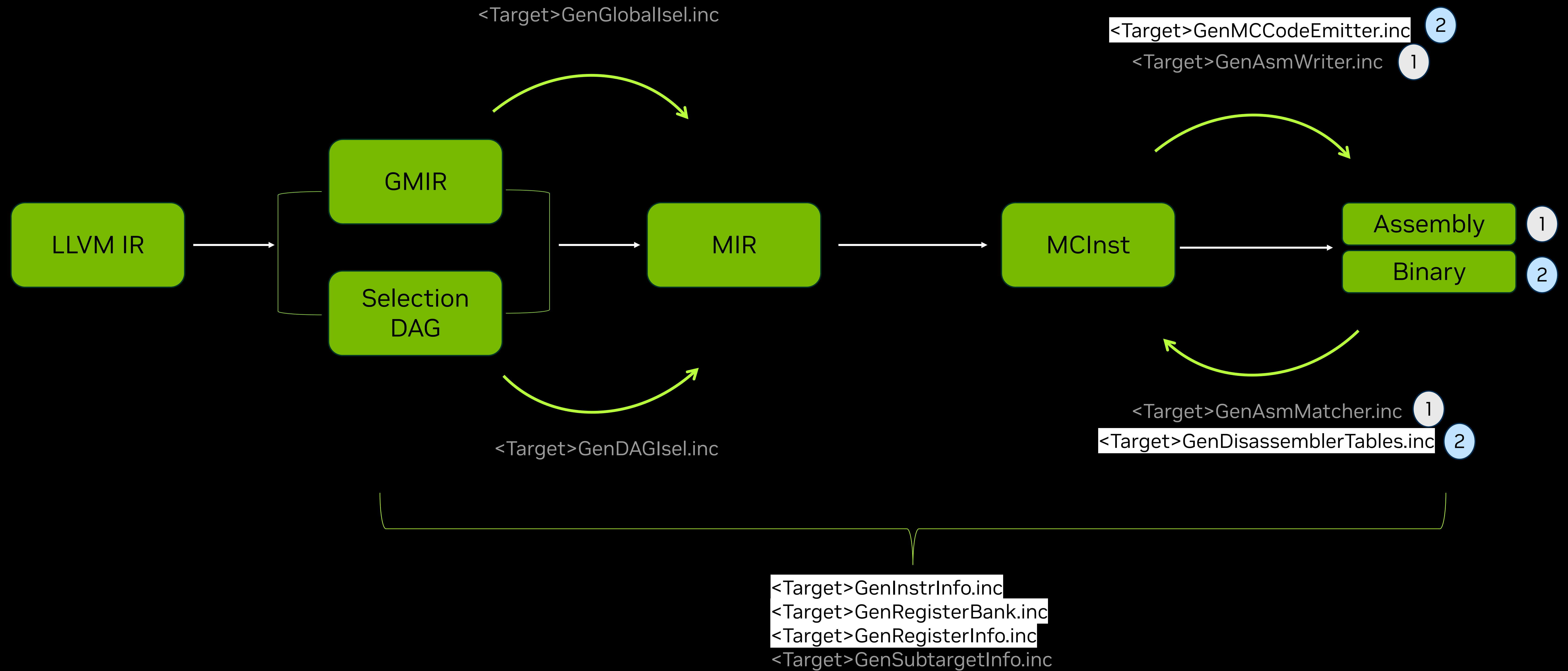
The Flow

IR in the LLVM Backend, how the .inc files are used



The Scope

.inc files which will be covered



An Example

32b add instruction with #imm operand getting converted to ADDWri in AArch64

```
def Wri :AddSubImmShift<isSub,  
0, GPR32sp, GPR32sp, addsub_shifted_imm32,  
mnemonic,Opcode> {  
    let Inst{31} = 0;  
}
```

LLVM IR

```
%result = add i32 %a, 42
```

GMIR

```
%2:gpr(s32) = G_CONSTANT i32 42  
%3:gpr32(s32) = G_ADD %1:gpr, %2:gpr
```

MIR

```
%3:gpr32common = ADDWri %1:gpr32sp, 42, 0
```

MCInst

```
<MCInst 1655 <MCOperand Reg:216>  
<MCOperand Reg:209>  
<MCOperand Imm:42>  
<MCOperand Imm:0>>
```

Assembly/Binary

```
add    w8, w1, #42  
[0x28,0xa8,0x00,0x11]
```


An Example

The **abstract, concrete records (.td)** invoked to convert **MIR -> Assembly/Binary**

```
def Wri :AddSubImmShift< isSub,  
0, GPR32sp, GPR32sp, addsub_shifted_imm32  
mnemonic,Opcode> {  
  let Inst{31} = 0;  
}
```

```
def addsub_shifted_imm32 : addsub_shifted_imm<i32>;
```

```
class addsub_shifted_imm<ValueType Ty>  
  : Operand<Ty>, ComplexPattern<Ty,  
2,"SelectArithImmed", [imm]> {  
  let PrintMethod = "printAddSubImm";  
  let EncoderMethod = "getAddSubImmOpValue";  
  let ParserMatchClass = AddSubImmOperand;  
  let MIOperandInfo = (ops i32imm, i32imm);  
}
```

```
class AddSubImmShift<bit isSub, bit setFlags,  
RegisterClass dstRegtype,RegisterClass srcRegtype,  
addsub_shifted_imm immtype,string asm_inst,  
SDPatternOperator OpNode>:  
BaseAddSubImm<isSub, setFlags, dstRegtype,  
asm_inst,"\t$Rd, $Rn, $imm", (ins srcRegtype:$Rn,  
immtype:$imm), (set dstRegtype:$Rd, (OpNode  
srcRegtype:$Rn, immtype:$imm))> {  
  bits<14> imm;  
  let Inst{23-22} = imm{13-12}; // '00' => lsl #0,  
'01' => lsl #12  
  let Inst{21-10} = imm{11-0};  
  let DecoderMethod = "DecodeAddSubImmShift";  
  let hasPostISelHook = 1;  
}
```

```
class BaseAddSubImm<bit isSub, bit setFlags,  
RegisterClass dstRegtype,string asm_inst,string  
asm_ops,dag inputs, dag pattern> : I<(outs  
dstRegtype:$Rd), inputs,asm_inst, asm_ops, "",  
[pattern]>, Sched<[WriteI, ReadI]> {  
  bits<5> Rd;  
  bits<5> Rn;  
  let Inst{30} = isSub;  
  let Inst{29} = setFlags;  
  let Inst{28-24} = 0b10001;  
  let Inst{9-5} = Rn;  
  let Inst{4-0} = Rd;  
}
```


An Example

Relevant Information from the .td for this talk

```
def Wri :AddSubImmShift< isSub,  
0, GPR32sp, GPR32sp, addsub_shifted_imm32,  
mnemonic,Opcode> {  
  let Inst{31} = 0;  
}
```

3 Target Register class

```
class AddSubImmShift<bit isSub, bit setFlags,  
RegisterClass dstRegtype,RegisterClass srcRegtype,  
addsub_shifted_imm immtype,string asm_inst,  
SDPatternOperator OpNode>:  
BaseAddSubImm<isSub, setFlags, dstRegtype,  
asm_inst,"\\t$Rd, $Rn, $imm", (ins srcRegtype:$Rn,  
immtype:$imm), (set dstRegtype:$Rd, (OpNode  
srcRegtype:$Rn, immtype:$imm))> {  
  bits<14> imm;  
  let Inst{23-22} = imm{13-12}; // '00' => lsl #0,  
  '01' => lsl #12  
  let Inst{21-10} = imm{11-0};  
  let DecoderMethod = "DecodeAddSubImmShift";  
  let hasPostISelHook = 1;  
}
```

4 Decoder Method

```
def addsub_shifted_imm32 : addsub_shifted_imm<i32>;
```

```
class addsub_shifted_imm<ValueType Ty>  
  : Operand<Ty>, ComplexPattern<Ty,  
2,"SelectArithImmed", [imm]> {  
  let PrintMethod = "printAddSubImm";  
  let EncoderMethod = "getAddSubImmOpValue";  
  let ParserMatchClass = AddSubImmOp;  
  let MIOperandInfo = (ops i32imm, i32imm);  
}
```

5 Encoder Method

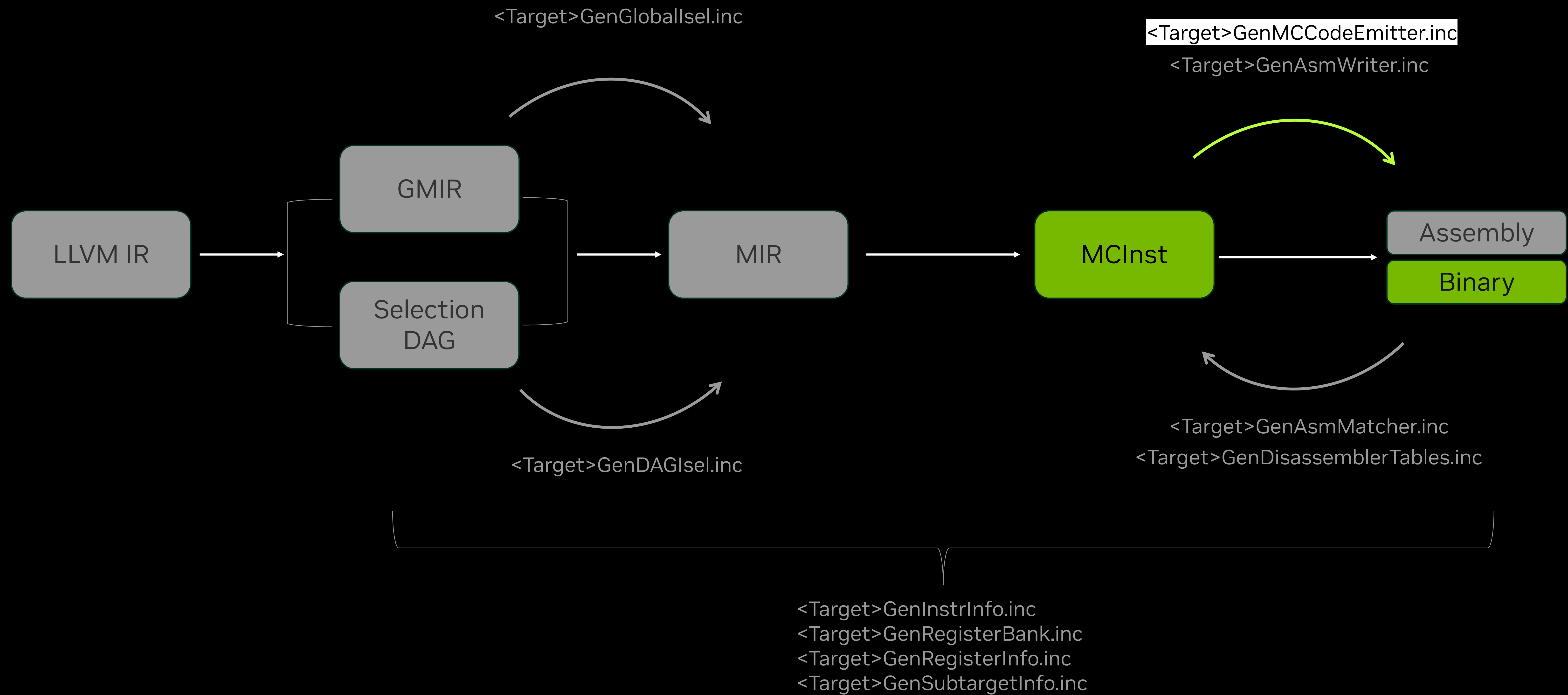
```
class BaseAddSubImm<bit isSub, bit setFlags,  
RegisterClass dstRegtype,string asm_inst,string  
asm_ops,dag inputs, dag pattern> : I<(outs  
dstRegtype:$Rd), inputs,asm_inst, asm_ops, "",  
[pattern]>, Sched<[WriteI, ReadI]> {  
  bits<5> Rd;  
  bits<5> Rn;  
  let Inst{30} = isSub;  
  let Inst{29} = setFlags;  
  let Inst{28-24} = 0b10001;  
  let Inst{9-5} = Rn;  
  let Inst{4-0} = Rd;  
}
```

2 Sched Models

1 Encoding

Coming up...

<Target>GenMCCodeEmitter.inc – Streams MCIInst as Binary



<Target>GenMCCodeEmitter.inc

(eg: AArch64GenMCCodeEmitter.inc)

- Consists the code to stream MCInst to binary i.e (32b/64b) unsigned value of the instruction.

MCInst
<MCInst 1655 <MCOperand Reg:216>
<MCOperand Reg:209>
<MCOperand Imm:42>
<MCOperand Imm:0>>

//AArch64MCCodeEmitter.cpp
encodeInstruction()

//AArch64GenMCCodeEmitter.inc
getBinaryCodeForInstr()

Binary
//add w8, w1, #42
[0x28, 0xa8, 0x00, 0x11]

<Target>GenMCCodeEmitter.inc

.td -> .inc

```
class BaseAddSubImm<bit isSub, bit setFlags,
RegisterClass dstRegtype,string asm_inst,string
asm_ops,dag inputs, dag pattern> : I<(outs
dstRegtype:$Rd), inputs,asm_inst, asm_ops, "",
[pattern]>, Sched<[WriteI, ReadI]> {
    bits<5> Rd;
    bits<5> Rn;
    let Inst{30}      = isSub;
    let Inst{29}      = setFlags;
    let Inst{28-24}   = 0b10001;
    let Inst{9-5}     = Rn;
    let Inst{4-0}     = Rd;
}
```

```
case AArch64::ADDWri: {
    // op: Rd
    op = getMachineOpValue(MI, MI.getOperand(0), Fixups, STI);
    op &= UINT64_C(31);
    Value |= op;
    // op: Rn
    op = getMachineOpValue(MI, MI.getOperand(1), Fixups, STI);
    op &= UINT64_C(31);
    op <= 5;
    Value |= op;
    // op: imm
    op = getAddSubImmOpValue(MI, 2, Fixups, STI);
    op &= UINT64_C(16383);
    op <= 10;
    Value |= op;
    break;}
}
```

MCInst

```
<MCInst 1655 <MCOperand Reg:216>
<MCOperand Reg:209>
<MCOperand Imm:42>
<MCOperand Imm:0>>
```

↓

```
//AArch64MCCodeEmitter.cpp
encodeInstruction()
```

↓

```
//AArch64GenMCCodeEmitter.inc
getBinaryCodeForInstr()
```

Binary

↓

```
//add w8, w1, #42
[0x28,0xa8,0x00,0x11]
```


<Target>GenMCCodeEmitter.inc

.td -> .inc

```
class BaseAddSubImm<bit isSub, bit setFlags,
RegisterClass dstRegtype,string asm_inst,string
asm_ops,dag inputs, dag pattern> : I<(outs
dstRegtype:$Rd), inputs,asm_inst, asm_ops, "",
[pattern]>, Sched<[WriteI, ReadI]> {
    bits<5> Rd;
    bits<5> Rn;
    let Inst{30}      = isSub;
    let Inst{29}      = setFlags;
    let Inst{28-24}   = 0b10001;
    let Inst{9-5}     = Rn;
    let Inst{4-0}     = Rd;
}
```

```
case AArch64::ADDWri: {
    // op: Rd
    op = getMachineOpValue(MI, MI.getOperand(0), Fixups, STI);
    op &= UINT64_C(31);
    Value |= op;
    // op: Rn
    op = getMachineOpValue(MI, MI.getOperand(1), Fixups, STI);
    op &= UINT64_C(31);
    op <= 5;
    Value |= op;
    // op: imm
    op = getAddSubImmOpValue(MI, 2, Fixups, STI);
    op &= UINT64_C(16383);
    op <= 10;
    Value |= op;
    break;}
}
```

MCInst

```
<MCInst 1655 <MCOperand Reg:216>
<MCOperand Reg:209>
<MCOperand Imm:42>
<MCOperand Imm:0>>
```

↓

```
//AArch64MCCodeEmitter.cpp
encodeInstruction()
```

↓

```
//AArch64GenMCCodeEmitter.inc
getBinaryCodeForInstr()
```

Binary

↓

```
//add w8, w1, #42
[0x28,0xa8,0x00,0x11]
```


<Target>GenMCCodeEmitter.inc

.td -> .inc

```
class addsub_shifted_imm<ValueType Ty>
    : Operand<Ty>, ComplexPattern<Ty,
2,"SelectArithImmed", [imm]> {
    let PrintMethod = "printAddSubImm";
    let EncoderMethod = "getAddSubImmOpValue";
    let ParserMatchClass = AddSubImmOperand;
    let MIOperandInfo = (ops i32imm, i32imm);
}
```

```
case AArch64::ADDWri: {
    // op: Rd
    op = getMachineOpValue(MI, MI.getOperand(0), Fixups, STI);
    op &= UINT64_C(31);
    Value |= op;
    // op: Rn
    op = getMachineOpValue(MI, MI.getOperand(1), Fixups, STI);
    op &= UINT64_C(31);
    op <= 5;
    Value |= op;
    // op: imm
    op = getAddSubImmOpValue(MI, 2, Fixups, STI);
    op &= UINT64_C(16383);
    op <= 10;
    Value |= op;
    break;}
}
```

MCInst

```
<MCInst 1655 <MCOperand Reg:216>
<MCOperand Reg:209>
<MCOperand Imm:42>
<MCOperand Imm:0>>
```

//AArch64MCCodeEmitter.cpp
encodeInstruction()

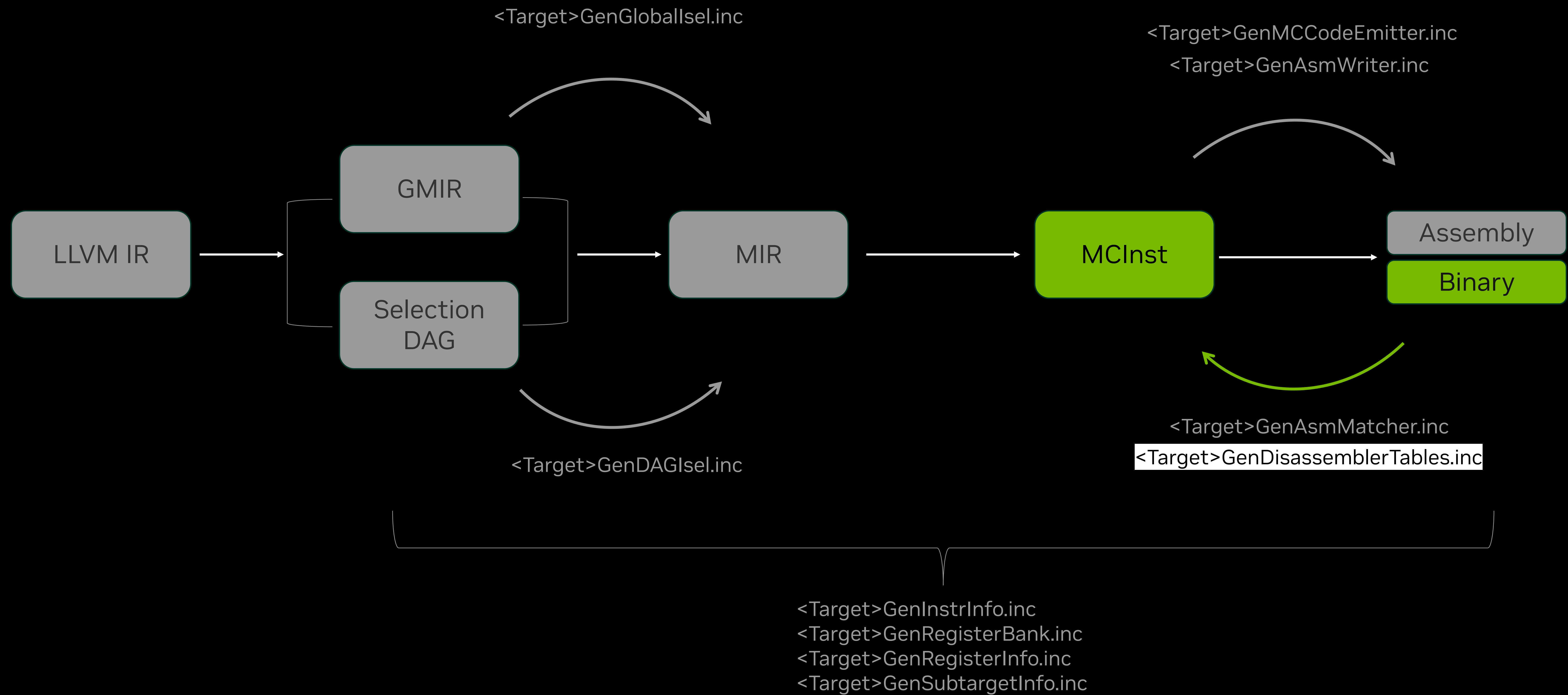
//AArch64GenMCCodeEmitter.inc
getBinaryCodeForInstr()

Binary

```
//add w8, w1, #42
[0x28,0xa8,0x00,0x11]
```


Coming up...

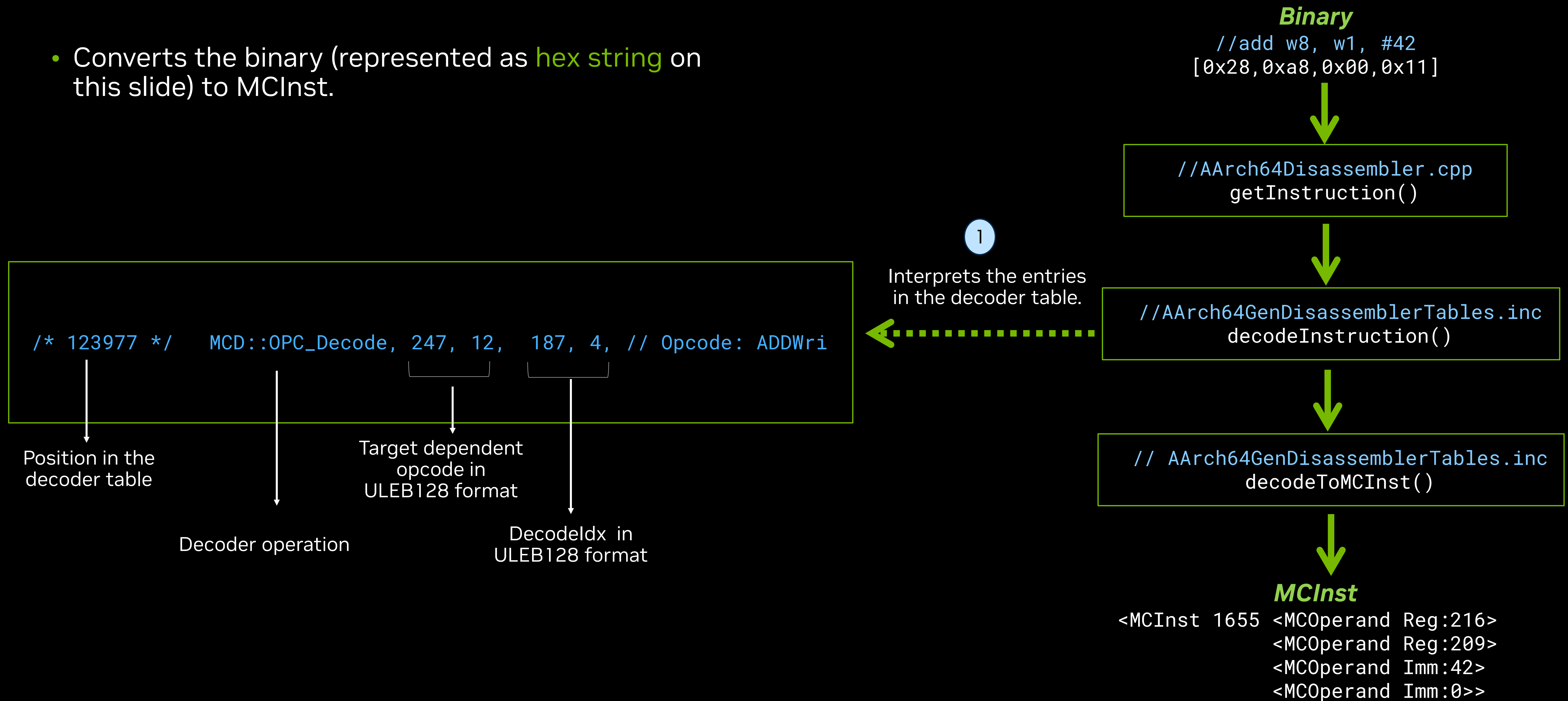
<Target>GenDisassemblerTables.inc – Disassembles Binary back to MCInst



<Target>GenDisassemblerTables.inc

Eg: AArch64GenDisassemblerTables.inc

- Converts the binary (represented as **hex string** on this slide) to MCInst.



<Target>GenDisassemblerTables.inc

decodeIdx calculation & .td -> .inc

```
/* 123977 */ MCD::OPC_Decode, 247, 12, 187, 4, // Opcode: ADDWri
Decodeldx in ULEB128 format
```

2 Calculate **decodeIdx** which is case label for the opcode in **decodeToMCInst**

decodeIdx calculation:
187 – 10111011 – drop 1st significant bit – 00111011 = 59
4 << 7 = 512
dIdx = 512 + 59 = **571**

```
case 571:
    if (!Check(S, DecodeAddSubImmShift(MI, insn, Address, Decoder))) {
        return MCDisassembler::Fail;
    }
    return S;
```

```
class AddSubImmShift<arguments>:
BaseAddSubImm<arguments> {
    ...
    let DecoderMethod = "DecodeAddSubImmShift";
    let hasPostISelHook = 1;
}
```

Binary

```
//add w8, w1, #42
[0x28,0xa8,0x00,0x11]
```

```
//AArch64Disassembler.cpp
getInstruction()
```

```
//AArch64GenDisassemblerTables.inc
decodeInstruction()
```

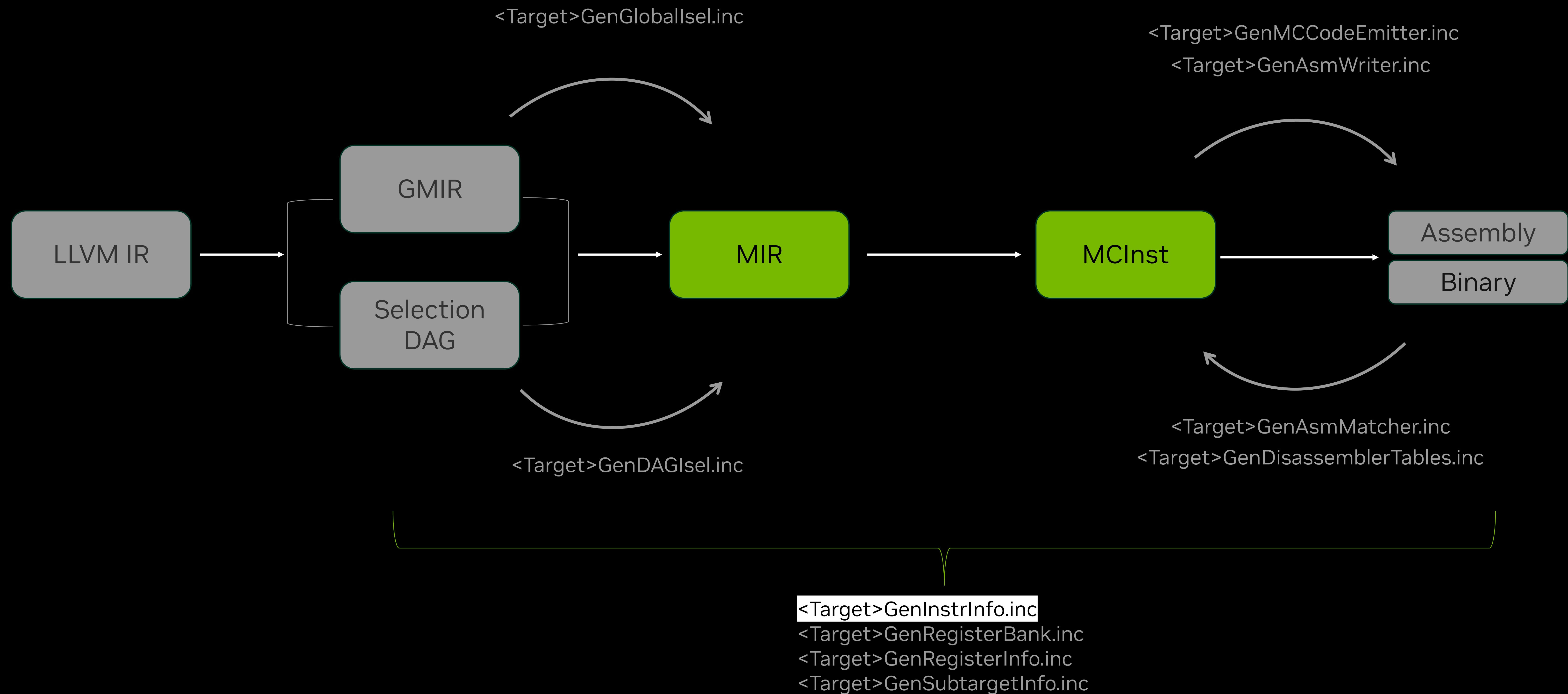
```
// AArch64GenDisassemblerTables.inc
decodeToMCInst()
```

MCInst

```
<MCInst 1655 <MCOperand Reg:216>
<MCOperand Reg:209>
<MCOperand Imm:42>
<MCOperand Imm:0>>
```


Coming up...

<Target>GenInstrInfo.inc – Information about the instruction set (ISA) which is used in various IRs



<Target>GenInstrInfo.inc

Eg: AArch64GenInstrInfo.inc

Enums for target

- independent (eg: GMIR opcode **G_ADD**)
- dependent (eg: MIR opcode **ADDWri**)

opcodes

```
// defined under
// GET_INSTRINFO_ENUM
G_ADD = 53,
. . .
ADDWri = 1655,
```

Enums for scheduling models applicable to **operands (defs, uses)** of the instruction

```
// .td code
class BaseAddSubImm<args>
:
I<args>,
Sched<[WriteI, ReadI]> {
}
```



```
// defined under
// GET_INSTRINFO_SCHED_ENUM

WriteI_ReadI = 4
```

Target specific properties applicable to the instruction

Defined using the .td
constructs

TIIPredicate,
MCSchedPredicate

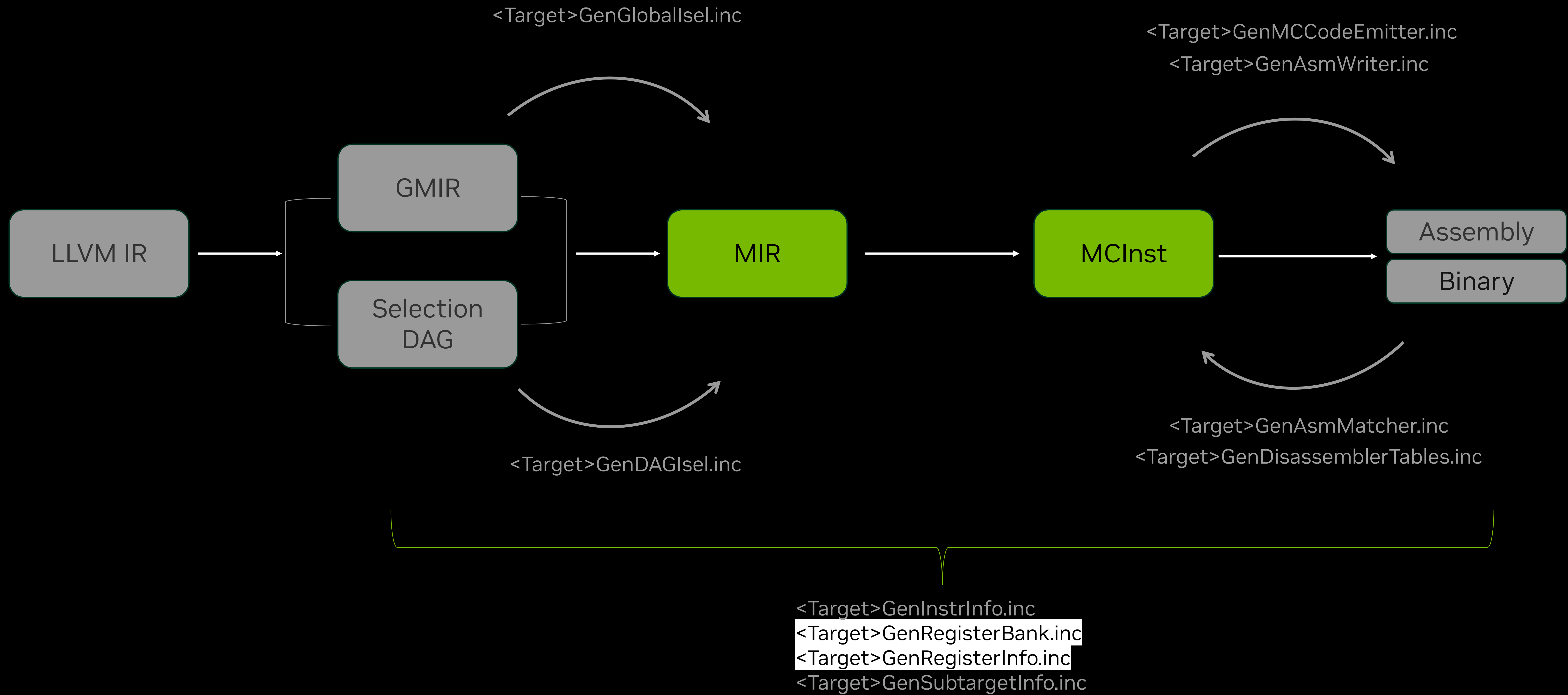
```
// defined under
// GET_INSTRINFO_HELPER_DECLS
// GET_INSTRINFO_HELPERS
```

isExynosArithFast()

Additionally, this file also consists the code to initialize the MCInst layer.

Coming up...

<Target>GenRegisterInfo.inc/ <Target>GenRegisterBank.inc
Emits target register file & bank for code generator



<Target>RegisterInfo.inc

- defines <Target>GenRegisterInfo class with several utility functions like `getSubRegisterClass`
- Enumerates registers, register classes, subregister indices

<Target>RegisterBank.inc

- Enumerates the various banks provided by arch and implements function `getRegBankFromRegClass`

GMIR

```
%2:_(s32) = G_CONSTANT i32 42  
%3:_(s32) = G_ADD %1:_, %2:_
```

After RegBankSelect

GMIR

```
%2:gpr(s32) = G_CONSTANT i32 42  
%3:gpr32(s32) = G_ADD %1:gpr, %2:gpr
```

After Instruction Selection

MIR

```
%3:gpr32common = ADDWri %1:gpr32sp, 42, 0
```

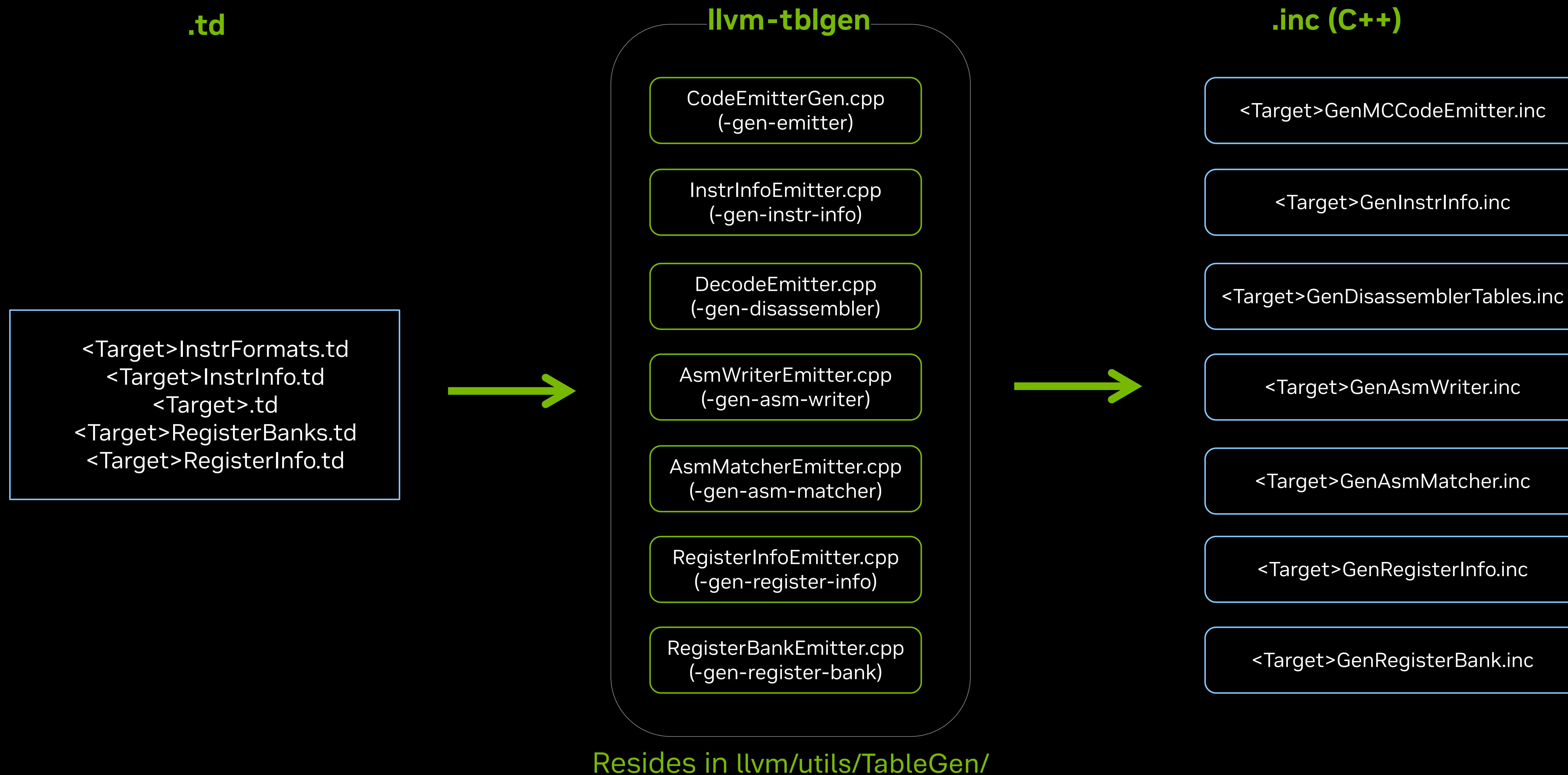

Some Common Errors

Not an exhaustive list!

| Failure Message | Possible Meaning | Possible Solution |
|---|---|--|
| Decoding Conflict: | <p>Can be seen while building llc.</p> <p>Static information is the same for multiple MIR opcodes. Possible case is when we create multiple MIR opcodes from the same abstract record.</p> | <p>Use DecoderNamespace while creating a new MIR opcode.</p> |
| <p>Type set is empty for each HW mode:</p> <p>possible type contradiction in the pattern below (use -print-records with llvm-tblgen to see all expanded records).</p> | <p>Can be seen while building llc.</p> <p>Generally related to ISel phase (when ISel patterns are written in .td). Could mean that operands have been assigned incorrect register classes i.e inconsistency with the InOperandList, OutOperandList.</p> | <p>There are two cases :</p> <ol style="list-style-type: none">1. The operands need to be assigned proper classes2. The generic SDnode or GMIR code is not in accordance with the InOperandList, OutOperandList specified by the assembly. And the InOperandList or OutOperandList need to be either modified or the patterns need to be converted to C++ code. |
| Undefined reference to record: <record_name> | <p>Error can be seen while building llc.</p> | <p>Means no concrete record for this variable</p> |

Wrapping up...

The various **backends** that **parse** the TableGen subDSLs and **generate** C++ code



Note : Depending on how we configure, **build.ninja** or **build.cmake** lists down all the commands used to build the particular .inc file using `llvm-tblgen`

References

- LLVM documentation on TableGen Backend - <https://llvm.org/docs/TableGen/BackEnds.html>
- llvm-tblgen options - <https://llvm.org/docs/CommandGuide/tblgen.html>
- How to write an LLVM Backend - <https://llvm.org/docs/WritingAnLLVMBackend.html>
- AArch64 backend code, TableGen Backend code for examples, finer details

Acknowledgments

Thanks to my colleagues at NVIDIA for their valuable feedback on the presentation

Shekhar Divekar, Jason Eckhardt, Madhur Amilkanthwar,
Subhranil Mukherjee, Dhruv Chawla, Soumya AR

