1 Introducción

En este documento se resume el proceso seguido para extender con nuevas instrucciones en forma de intrinsiscs LLVM y Clang, diviendo el proceso en lo hecho para el frontend (Clang) y lo hecho para el backend (LLVM). Con el objetivo de facilitar la revisión de la documentación en un futuro si se desean agregar nuevas instrucciones que se vayan a implementar para el ISA del acelerador. Se agregan ademas algunas recomendaciones que puedan llegar a ayudar en el desarrollo futuro del proyecto.

2 FrontEnd

El frontend está definido totalmente en Clang, en esta etapa las instrucciones que se agregaron se representan primero como un *Builtin*, un tipo de función que el compilador conoce de forma intrínseca, es decir que no es parte de una biblioteca externa, y que el compilador sabe manejar directamente.

Esta es una forma muy común de proveer al programador en un lenguaje de alto nivel una forma más directa de interactuar con el hardware ya que usualmente estos *Builtins* van a ser representados luego por una o unas pocas instrucciones en el código ensamblador. Para este proyecto se van a relacionar uno a uno los *Builtins* y las instrucciones en ensamblador ya que se busca que estos representan una operación especifica que pueda hacer el acelerador.

A la hora de definir los *Builtins* de Clang estos pueden ser dependientes o independientes de la arquitectura objetivo, esto es importante para seleccionar la clase del *Builtin* y los tipos de datos ya que distintas arquitecturas populares ya tienen clases y tipos de datos definidos, además, tienen también métodos diferentes de disminuir el *Builtin* y generar código LLVM IR.

Por esto para agregar el nuevo *Builtin* se añadió una clase en el archivo riscv_vector_common.td, que es donde se definen las clases existentes para RVV (RiscV Vector extension), la clase definida se puede observar en la Figura 1, esta clase extiende RVVOutOp1Op2Builtin con el fin de aprovechar la generación de código automática que tienen ya las clases definidas para RVV y aprovechar también los tipos de datos vectoriales ya definidos. Luego los *Builtins* se definen en riscv_vector.td extendiendo la clase creada [1].

```
class AccOutOp0Op1Builtin<string intrinsic_name, string suffix, string prototype, string type_range>
    : RVVOutOp0Op1Builtin<suffix, prototype, type_range> {
  let HasMasked = false;
  let HasMaskedOffOperand = false;
  let HasVL = false;
  let MaskedPolicyScheme = NonePolicy;
  let HasTailPolicy = false;
  let HasMaskPolicy = false;
  let Log2LMUL = [0];
  let IRName = intrinsic_name;
}
```

Figure 1: Definición de la clase para Builtins del acelerador

[1] Ambos archivos se encuentran en llvm-project/clang/include/clang/Basic

En la Tabla 1 se explican los parámetros de entrada que se utilizan a la hora de definir un nuevo *Builtin* de la clase añadida, con el fin de poder explicar y ejemplificar bien su uso.

Parámetro	Descripción	Ejemplo y explicación
Intrinsic name	Este es un parámetro de tipo *string* que define el nombre del *Intrinsic* de LLVM IR al que se debe transformar este *Builtin*, esto se hace para que el generador de código de Clang pueda hacer esta conexión de forma automática.	`"mul_vec"`: Con esta definición Clang va a buscar un *Intrinsic* que se haya definido con el nombre `"mul_vec"` además de los prefijos que se utilizan al definir *Intrinsics*
Type range	Es un *string* conformado por varias letras que definen los tipos de datos para los que se va a generar un *Builtin*, es decir si se define más de un tipo de dato se van a generar varios *Builtins* en una misma definición, para cada tipo de dato	"cif": Aqui la letra "c" indica el tipo de dato entero representado con 8bits (int8_t), la "i" el tipo de dato entero de 32bits (int32_t) y la "f" flotante de 32bits (float32_t).
Prototype	Es un *string* con diferentes transformadores de tipos, se utiliza junto al *Type range* para definir los parámetros del *Builtin* en C, siendo primero el tipo de retorno y luego el tipo de las entradas	"vvUe": Considerando un *type range* "i" este ejemplo representa que la salida (v) debe ser un vector de enteros de 32bits, el primer parámetro de entrada (v) debe ser otro vector de enteros de 32bits y el segundo parámetro (e) debe ser un entero de 32bits sin signo (U).
Sufix	Es un *string* similar al de *prototype*, pero puede ser opcional, este utiliza igualmente modificadores de tipo para definirse y se utiliza para acompañar el nombre del *Builtin* con el tipo de datos, luego de un guion bajo.	"vv": Con esta definición y un *type range* "i", el nombre del *Builtin* va a estar acompañado por un `_i32m1_i32m1`.

Table 1: Parámetros de la clase para Builtins del acelerador

Además, en la Tabla 2 se explican los atributos definidos para la clase, en general estos parámetros representan información que utilizan las instrucciones de RVV por la forma en que se define el ISA, pero ya que las instrucciones añadidas van a basarse en otro ISA se define una forma más simplificada para la clase de *Builtin* añadida.

Además de definir las instrucciones como un *Builtin*, se definieron también

Atributo	Descripción
HasMasked	Indica si existe una forma con mascara para el *Builtin*. La máscara es una propiedad de RVV que ayuda a operar solo ciertos valores de los vectores.
HasMasked OffOperand	Indica si el *Builtin* espera un operador especial relacionado a las operaciones con mascara de RVV.
HasVL	Indica si el *Builtin* recibe una definición de VL, variable de RVV que indica el tamaño de los vectores, en este caso se indica que no lo toma en cuenta, pero puede ser una buena opción a considerar para manejar el tamaño de vectores variables en el ISA del acelerador.
. MaskedPolicyScheme, HasTailPolicy y HasMaskPolicy	Estos atributos indican las formas en que se van a manejar algunas operaciones que utilizan características de RVV, que no son importantes para estos *Builtins*.
Log2LMUL	Este parámetro indica el valor de LMUL, este indica la multiplicidad para las operaciones una característica de RVV que permite operar sobre varios vectores a la vez, se define como 0 que representa una multiplicidad estándar de 1.

Table 2: Especificación de los atributos de la clase para el Builtin

en forma de *Intrinsics*, estas son instrucciones que se van a utilizar durante la generación de código para representar en el código LLVM IR los *Builtins* definidos. Estas le indican al compilador que estas instrucciones no forman parte del estándar de C y con sus atributos le indican la forma en que se deben manejar para optimización y que luego en el *backend* se va a definir la forma en que van a ser reducidas estas *Intrinsics*.

Para definir los nuevos *Intrinsics* se modificó el archivo IntrinsicsRISCV.td[2], se definieron algunas clases para las instrucciones que compartían tipos de datos tanto de las entradas como de la salida y para las instrucciones que no tenían tipos de datos iguales a otras se tuvo que hacer una definición de una en una. En la Figura 2 se muestra una de las clases definidas para uno de los tipos de *Intrinsics*.

```
class AccVecVecVecIntrinsics : DefaultAttrsIntrinsic<[llvm_anyvector_ty]
                                                     [LLVMMatchType<0>,
                                                      LLVMMatchType<0>],
                                                     [IntrNoMem,
                                                      IntrWillReturn,
                                                      IntrSpeculatable]>,
                               RISCVVIntrinsic;
```

Figure 2: Clase definida para un tipo de Intrinsic

En este caso la clase se llama AccVecVecVecIntrinsics, para un tipo de *In-*

[2]El archivo se encuentra en llvm-project/llvm/include/llvm/IR

trinsic que recibe dos vectores y retorna un vector. El primer atributo es una lista con los tipos de datos de las salidas, en este caso es solo un valor, llvm_anyvector_ty, que significa que la salida será un vector de cualquier tipo, el segundo parámetro es otra lista con los tipos de las entradas, en este caso ambas se definen con $LLVMMatchType < 0 >$, que significan que ambas deben tener el mismo tipo que la salida. El ultimo parámetro es otra lista donde se definen algunos atributos para del *Intrinsic*, en este caso se indica que la instrucción no lee ni escribe en memoria, que va a retornar algo y que el optimizador puede especular con ella respectivamente.

La clase además, se define para extender primero a DefaultAttrsIntrinsic, una clase que se utiliza para especificar ciertos valores estandar para los atributos de los *Intrinsics* y que a su vez extiende la clase base de *Intrinsics*. Luego se define para que extienda a RISCVVIntrinsic con el fin que Clang pueda hacer de forma automática la vinculación con el *Builtin* y se aproveche la generación de código definida para RVV.

3 *Backend*

El *backend* de LLVM es la etapa donde se transforma la LLVM IR a código de ensamblador de la arquitectura objetivo. Para eso se debe definir la forma en que el *backend* va a vincular los *Intrinsics* con instrucciones reales.

Para efectos de este proyecto la definición se hizo uno a uno en todo el proceso, es decir un *Builtin* se transforma como un único *Intrinsic* para LLVM IR que a su vez se transforma en una única instrucción en ensamblador. Al igual que para el frontend se aprovechan algunas definiciones de RVV para hacer la reducción y definición de las instrucciones.

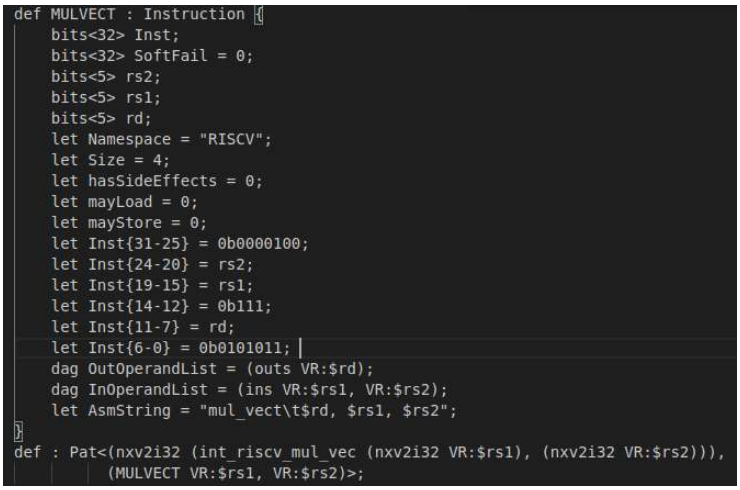
Debido a que la relación es directa y que ninguna de las instrucciones a agregar interactúa con memoria o tiene efectos secundarios se puede hacer una reducción estándar, es decir no se tuvieron que agregar casos específicos para las instrucciones. Sin embargo, para posibles expansiones futuras del ISA se van a explicar más adelante algunas consideraciones para instrucciones que si requieran un proceso personalizado.

Para poder transformar el *Intrinsic* lo primero que se debe hacer es definir la forma de la instrucción a la que se va a reducir, para esto se creó el archivo RISCVInstrInfoAcc.td y se incluyó en RISCVInstrInfoV.td, que es donde se definen las instrucciones para RVV. La definición de una de las instrucciones se puede ver en la Figura 3. En este caso se utiliza la clase más básica *Instruction* ya que el formato para el ISA del acelerador aún no se ha establecido, para el futuro del proyecto se podría definir una clase para el ISA que extienda Instruction o alguna otra clase en el archivo RISCVInstFormats.td [3].

Para explicar mejor la Figura 3 se van a definir sus atributos.

1. Primero se definen características de la instrucción, en este caso bits¡32¿ Inst define el tamaño de la instrucción, SoftFail = 0 es un atributo para

[3]Estos archivos se encuentran en llvm-project/llvm/lib/Target/RISCV



```
def MULVECT : Instruction {
    bits<32> Inst;
    bits<32> SoftFail = 0;
    bits<5> rs2;
    bits<5> rs1;
    bits<5> rd;
    let Namespace = "RISCV";
    let Size = 4;
    let hasSideEffects = 0;
    let mayLoad = 0;
    let mayStore = 0;
    let Inst{31-25} = 0b0000100;
    let Inst{24-20} = rs2;
    let Inst{19-15} = rs1;
    let Inst{14-12} = 0b111;
    let Inst{11-7} = rd;
    let Inst{6-0} = 0b0101011;
    dag OutOperandList = (outs VR:$rd);
    dag InOperandList = (ins VR:$rs1, VR:$rs2);
    let AsmString = "mul_vect\t$rd, $rs1, $rs2";
}
def : Pat<(nxv2i32 (int_riscv_mul_vec (nxv2i32 VR:$rs1), (nxv2i32 VR:$rs2))),
          (MULVECT VR:$rs1, VR:$rs2)>;
```

Figure 3: Definición de la instrucción vec mul para el backend

el desensamblador que no es relevante para todas las arquitecturas pero que se debe definir para que funcione correctamente la instrucción, luego se define el tamaño en bits de los registros destino y fuente, por ultimo Size define el tamaño en bytes de la instrucción y Namespace vincula la instrucción con una arquitectura, en este caso Riscv5.

2. HasSideEffects indica si la instrucción va a tener efectos secundarios que no van a ser capturados por operandos de la instrucción o banderas. En este caso 0 indica que no tiene efectos secundarios.

3. mayLoad/mayStore se utilizan para instrucciones que vayan a leer o escribir en memoria respectivamente, un 0 significa que la instrucción no interactúa con la memoria.

4. InstA-B es una forma para definir el valor de los bits de la instrucción desde el bit A hasta el bit B. De esta forma se deben definir los bits que corresponden a los registros de destino y fuente definidos al inicio de la instrucción y que la definición sea coherente con el tamaño indicado al inicio.

5. OutOperandList/InOperandList estos son de tipo DAG y se usan para indicar las entradas y salidas de la instrucción referenciando los registros fuente y registros de salida. En este caso outs VR:$rd indica que hay un único registro destino y que este es de la clase VR (Vector Register), en el caso de las entradas se define que hay dos registros de donde viene la información y ambos son de tipo VR.

6. AsmString define la forma en que se va a escribir la instrucción en los archivos de código ensamblador (.s). en este caso primero se indica el

nombre a mostrar para la instrucción y luego los registros que participan (en el archivo .s no se muestra como $rd pero como el registro físico de la arquitectura que se utiliza, por ejemplo, v6).

Además de registros de tipo VR, el *backend* de RISCV tiene otros tipos como GPR (Registros de propósito general), FPR (Registros para punto flotante), ademas se puede agregar una clase nueva de registros de ser necesario, definiendola en el archivo RISCVRegisterInfo.td que extienda de alguna clase de registro RiscV o de la clase Register base. Ademas, se deben definir los registros existentes para esta nueva clase.

Luego de haber definido la instrucción se debe hacer la conexión entre el *Intrinsic* y la instrucción, para eso se definió un patrón (Pattern) junto a la definición de la instrucción, estos patrones se utilizan durante la selección de instrucciones en la etapa de generación de código para poder vincular un nodo del DAG generado a partir del LLVM IR con un instrucción máquina de la arquitectura objetivo.

En la Figura 3 se puede observar al final, como se definió el patrón para la instrucción vec mul, este recibe dos parámetros primero el *Intrinsic* y luego la instrucción. En ambos casos se definen las entradas en forma de dag con el tipo de registros. En el caso del *Intrinsic* también se debió definir los tipos de datos tanto del resultado de la operación como de las entradas, en este caso los tres son del tipo nxv2i32, que es uno de los tipos de datos vectoriales definidos para RVV que representa vectores escalables de enteros de 32bits para una multiplicidad de 1.

Para las instrucciones implementadas no era necesario definir una forma de reducirlas personalizada ya que todas eran operaciones aritméticas relacionadas uno a uno entre C y RISCV. Pero ahora se van a especificar algunas consideraciones para instrucciones que no cumplan estos requisitos.

LLVM permite el uso de Pseudoinstrucciones, estas son similares a las instrucciones definidas en el sentido de que reciben un dag para especificar las salidas y para especificar las entradas, además de otra información, pero estas no se definen en forma de bitcode ya que lo que distingue a este tipo de instrucciones es que no representan una instrucción de ensamblador, se utilizan para etapas intermedias y LLVM entiende que luego se va a especificar de forma explícita lo que debe hacerse con estas instrucciones, por ejemplo según algún parámetro convertirse a cierta instrucción específica para cierto tipo de dato, o bien ser reducida a varias instrucciones de ensamblador que se deban ejecutar en secuencia.

Estas son de gran utilidad para poder definir *Builtins* más complejos que no representan directamente una instrucción permitida en la arquitectura pero que luego LLVM va a saber cómo representarlos utilizando instrucciones de máquina, con esto se puede facilitar el trabajo a la hora de desarrollar en C, por ejemplo, al hacer análisis del código si se encuentran patrones de instrucciones que se repitan se pude generar un *Builtin* que agrupe estas instrucciones y facilite la programación.

En el caso de ensamblador otra forma en que se pueden optimizar instruc-

ciones más complejas es definir un patrón compuesto, en el que el intrínseco se relacione con varias instrucciones maquina anidadas para que se pueda generar código más eficiente por ejemplo casos donde el resultado de una operación va a ser la entrada de otra.

Si se quiere hacer una reducción personalizada se debe modificar el archivo RISCVISelLowering.cpp [4], en este primero debemos definir la operación mediante la función setOperationAction(), este recibe primero el tipo de nodo, luego el tipo de dato (*Machine value type*) y luego la acción (*Custom* para una reducción personalizada), esta función le indica al compilador que si encuentra un nodo del tipo indicado con los tipos de datos indicados debe buscar la definición para una reducción personalizada.

Luego en este mismo archivo se debe buscar la función RISCVTargetLowering::LowerOperation, esta es la encargada de legalizar las operaciones, es decir tomar operaciones que son válidas para LLVM IR y convertirlas en operaciones que sean válidas para la arquitectura objetivo. Para esto se debe conocer el identificador de la operación y generar un caso dentro de la función para este identificador. En el caso se debe entonces definir la forma en que el compilador debe construir el nuevo nodo del DAG para que sea válido.

Los pasos anteriores definen una forma personalizada para la etapa de legalización de operaciones de la reducción, pero también puede ser necesario personalizar la forma en que se hace la reducción en la etapa de selección de instrucciones. Para esto se debe modificar la función RISCVDAGToDAGISel::Select en el archivo RISCVDAGToDAGISel.cpp [5], igualmente se debe definir un caso para el tipo de nodo de la función que queremos reducir y ahí definir la forma en que queremos que se remplace el nodo genérico del DAG por un nodo maquina relacionado con la arquitectura objetivo.

4 Recomendaciones

A la hora de definir los *Builtins* que se van a agregar a C se recomienda tener en consideración como van a representarse estos luego mediante instrucciones reales del ISA, ya que estos pueden querer abarcar demasiada complejidad que en C puede ser posible pero luego en ensamblador no poder ser representados por una instrucción.

Se recomienda considerar añadir *intrinsics* que sirvan para configurar características del *hardware* o para preparar el programa para instrucciones que puedan necesitarlo, ya que esta alternativa puede ser más sencilla y controlada que generar relaciones de un *Builtin* a varias instrucciones de ensamblador.

Ya que el ISA que se propone para el acelerador va a tener ciertas características similares a la extensión vectorial de RISCV se recomienda analizar la posibilidad de aprovechar ciertas características que ya están implementadas en el compilador para RVV, para el manejo de tamaños de vectores variables, el manejo de que elementos de los vectores se quieren procesar, entre otras, ya que

[4] Este se encuentra en llvm-project/llvm/lib/Target/RISCV
[5] Este se encuentra en llvm-project/llvm/lib/Target/RISCV

esto disminuye el trabajo al extender el compilador. O bien utilizar estas como guía o base a la hora de implementarlas de forma separada para el acelerador.