Frontend

Leon Mrosewski, Jan Van Wickern, Darius Happe

Einleitung

Die Hauptaufgabe des Frontends bestand daraus einen gegebenen MSP430 Assembler Code einzulesen und diesen anschließend auf die LLVM-Zwischensprache LLVM IR (LLVM Intermediate Representation) zu “liften“.
Im Allgemeinen heißt das, dass wir jede Instruktion gegen ihren Pendant in der LLVM IR Sprache austauschen.
Allerdings gibt es nicht für jede Instruktion für den MSP430 eine passende Gegeninstruktion in der LLVM IR. Somit müssen manche Instruktionen mit mehreren LLVM IR Instruktionen dargestellt werden.
Dies ist vor allem bei den verschiedenen Adressing Modes der Fall.

Im folgenden wird der allgemeine Programmverlauf kurz Skizziert. Im Verlauf dieser Dokumentation wird noch genauer auf die einzelnen Funktionen eingegangen.

Zunächst wird, nachdem das Inputfile geöffnet wurde, nach bestimmten Header Anweisungen gesucht. Diese sind zwar nicht unbedingt wichtig für llvm aber da wir später wieder zur Ausgangsarchitektur zurück kehren wollen (MSP430) werden diese benötigt.

Im zweiten Schritt wird nach Labeln gesucht. Dafür wird einmal das gesamte File durchlaufen. Dies ist wichtig damit später ordentliche Verkettungen (wie Jumps) erstellt werden können.

Daran anschließend werden nun die Instruktionen ausgelesen. Dabei wird der Vorgang unterteilt in das Auslesen des Opcodes (also der Anweisung) und das Auslesen der betroffenen Register und deren Adressing Modes.

Hierbei ist zu beachten, dass wir nur eine Instruktion auf einmal einlesen und liften. Ist der lift abgeschlossen wird zur nächsten Instruktion gesprungen.

Wenn eine Instruktion geliftet werden soll werden zunächst, abhängig vom Adressing Mode, eventuell erforderliche zusätzliche Instruktionen erstellt.

Im Anschluss wird die vorbereitete Instruktion durch eine andere Funktion letztendlich geliftet. Das heißt das die passende llvm Instruktion erstellt wird und in den aktuellen Basicblock eingetragen wird.

Anschließend werden noch die Metadaten angehängt, die von der Transformation und dem Backend benötigt werden.

Damit ist der lift einer Instruktion abgeschlossen und das Programm springt zur nächsten.

Blick in die Funktionen: parse

Verantwortliche: Leon Mrosewski, Jan Van Wickern
Optimierungen: Darius Happe

Die Hauptaufgabe der „*parse(..)*“ Funktion ist das einlesen der Instruktionen und das weitergeben der relevanten Informationen an die Funktionen die das liften umsetzen.

Zunächst müssen die Header des gegebenen Assembler Files eingelesen werden.
Dies geschieht über die „*getHeadInformations(...)*“ Funktion. Derzeit werden nur „*.global*“, welches den Namen des Assembler Files definiert und „*.text*“, welches den Start des Programms angibt unterstützt.
Sobald das erste Label eingelesen wird wird die Schleife beendet, da nun die Instruktionen folgen und keine weiteren Header Informationen folgen können.

Im nächsten Schritt müssen zunächst alle Label die in dem Input File vorhanden sind eingelesen werden. Anschließend wird für jedes Label ein Basicblock erstellt und in einer Map hinterlegt.
Dies ist wichtig damit später, wenn zum Beispiel Jump Instruktionen umgesetzt werden müssen, der folge Basicblock direkt eingesetzt werden kann.
Dabei wird das Komplette Input File einmal komplett durchlaufen. Ist dies geschehen und alle vorhanden Labels wurden eingelesen wird der Inputstream, über welchen die Datei eingelesen wurde, wieder zurück gesetzt.

Nun folgt das einlesen der Instruktionen. Dafür wird zunächst über „*gettok(...)*“ zunächst der Opcode der Instruktion einlesen. Opcodes können zum Beispiel „*add*“, „*mov*“, „*sub*“ etc. sein.
Es können über diese Funktion allerdings auch Labels eingelesen werden. Wenn dies der Fall ist wird aus der zuvor erstellten Basicblock Map der passende Basicblock geladen und in die llvm Mainfunktion eingefügt. Somit wird jede weitere Instruktion die wir erstellen in diesen Basicblock eingefügt.

Wenn jetzt allerdings ein anderer Opcode eingelesen wird müssen, abhängig vom Opcode, die weiteren zugehörigen Variabeln (Register, Immediates,…) eingelesen werden.
Dies wird über die Funktion „*gettok2op(...)*“ realisiert.
Hierbei wird in der *parse* Funktion zwischen verschiedenen Fällen unterschieden.

Wenn es sich um eine Instruktion mit zwei Parametern handelt (also mit einem Source und einem Destination Register) werden diese hintereinander eingelesen und in Variablen gespeichert.
Dabei wird auch der entsprechende Adressing Mode mit ausgelesen.
Ist dies ohne Fehler geschehen werden all diese ermittelten Daten an die „*prepareForLift(...)*“ Funktion übergeben.

Wenn es sich bei dem Opcode um einen der emulierten Befehle handelt muss vorher noch überprüft werden ob der emulierte Befehl einen oder zwei Variablen nutzt und dementsprechend auslesen und die *prepareForLift(…)* Funktion aufrufen.

Wurden nun alle Instruktionen eingelesen wird der Inputstream geschlossen.
Anschließend wird über eine llvm Funktion der erstellte LLVM IR Code auf Korrektheit überprüft.
Falls dieser in seiner Form nicht lauffähig ist wird der Nutzer darauf hingewiesen das der Code „*malformed*“ ist.

Falls er jedoch alle Tests besteht wird er in die Ausgabedatei, die der Nutzer bei Programmstart festgelegt hat geschrieben. Das Ausgabeformat ist dabei Bitcode.

Blick in die Funktionen: getHeadInformations

Verantwortliche: Leon Mrosewski, Darius Happe

Die „*getHeadInformations(...)*“ Funktion liest im allgemeinen nur die Header einer MSP430 Assembler File ein. Mögliche Header sind in dem Fall „*.global*“ oder „*.text*“.
Die Funktion gibt, falls sie ein passenden bzw. existierenden Header gefunden hat, ein entsprechendes Token für diesen Header zurück. Dieses Token wird dann in der *parse(…)* Funktion verarbeitet. Zusätzlich werden, wenn dem Header noch weitere Informationen folgen (wie im Fall von *.global*), diese in einem übergeben String Pointer gespeichert, so dass *parse* darauf zugriff hat.

Desweiteren werden die zuvor ermittelten Header noch als Metadaten angehängt, damit diese im Backend wieder ausgelesen werden können und in die Ausgabe MSP430 Assembler Datei eingefügt werden können. Dies wird über die *appendGlobalMD(…)* Funktion realisiert.

Blick in die Funktionen: gettok

Verantwortliche: Leon Mrosewski

Die *gettok(…)* Funktion liest den Opcode der aktuellen Instruktion ein. Dabei gibt es zu beginn beim einlesen keine Restriktionen oder Verschachtelungen. Es wird erst mal nur eingelesen. Wichtig wird es sobald der Opcode an sich abgeschlossen ist. Nun wird nämlich festgelegt ob es sich um einen Word (16 bit) oder einen Byte (8 bit) Befehl handelt.
Ein Word Befehl liegt vor wenn entweder “*.w*“ direkt an den Opcode angehängt ist oder ob einfach gar nichts angehängt wurde.

Beispiel Word Befehl:

1. add.w r5, r6
2. add r5, r6

Um einen Byte Befehl zu erhalten gibt es nur die Möglichkeit ein “*.b*“ an den Opcode anzuhängen.

Das heißt also wenn ein Punkt direkt nach dem Opcode folgt muss entweder ein *w* oder ein *b* folgen. Falls jedoch nur ein Leerzeichen folgt wird die Instruktionsgröße direkt auf Word gesetzt.

Danach wird geprüft zu welcher Instruktionskategorie der Opcode gehört. Der ermittelte Opcode wird nun mithilfe des übergebenen Pointers auf einen String gespeichert und es wird ein passendes Token zurückgegeben, sodass die *parse* Funktion damit weiter verfahren kann.

Desweiteren können auch Label in der *gettok* Funktion erkannt werden. Diese sind erkennbar daran das direkt nach dem ermittelten Opcode (welcher in diesem Fall kein Opcode ist) ein “*:*“ folgt.

Blick in die Funktionen: gettok2op

Verantwortliche: Leon Mrosewski, Jan Van Wickern

die *gettok2op(…)* Funktion ermittelt die dem Opcode nachfolgenden Variablen die für für die Instruktion genutzt werden.
Dabei wird zwischen 7 Verschiedenen Adressingmodes unterschieden:

Adressing Mode	**Syntax**	**Beispiel**
Register Mode	Rn	add r5, r6
Indexed Mode	X(Rn)	add 5(r5), 6(r6)
Symbolic Mode	ADDR	add EDE, TONI
Absolute Mode	&ADDR	add &EDE, &TONI
Indirect Register Mode	@Rn	add @r5, @r6
Indirect Autoincrement Register Mode	@Rn+	add @r5+, @r6+
Immediate Mode	#N	add #5, r5

Dabei kann jeder Adressing Mode mit jedem anderem Adressing Mode kombiniert werden. Eine Ausnahme bildet da allerdings der Immediate Mode, da dieser nicht auf der Destination Seite stehen kann. Dies hat den Grund das man kein Ergebnis in eine Zahl schreiben kann.

Jetzt liest die *gettok2op* eben die Register mit ihrem Adressing Mode ein. Dabei liest sie jedoch bei jedem Aufruf der Funktion nur ein Register ein. Das heißt um zwei Register für einen Opcode einzulesen muss diese zweimal aufgerufen werden.
In ihrem Verlauf unterscheidet die Funktion zwischen den verschiedenen Darstellungsarten für die verschiedenen Adressing Modes und kann so genau bestimmen um welchen es sich handelt. Dementsprechend wird wieder ein Token zurückgegeben, sodass die *parse* Funktion dies entsprechend verarbeiten/ weitergeben kann.

Blick in die Funktionen: prepareForLift

Verantwortliche: Leon Mrosewski
Optimierungen: Darius Happe

Die Aufgabe der *prepareForLift(…)* Funktion ist, falls die aktuelle Instruktion nur durch mehrere LLVM IR Instruktionen dargestellt bzw. emuliert werden kann, diese Instruktionen zu erstellen. Dabei muss zwischen den verschiedenen Adressing Modes unterschieden werden.

Je nach Adressing Mode müssen eventuelle zusätzliche Instruktionen hinzugefügt werden, damit der MSP430 Befehl korrekt emuliert bzw. in LLVM IR übersetzt werden kann.

Bei dem Indexed Mode zum Beispiel muss zunächst der Inhalt des betroffenen Registers ausgelesen werden. Anschließend muss der Offset, welcher mitgegeben wurde, zu dem Wert in dem Register addiert werden. Das Ergebnis aus dieser Addition ergibt die Speicheradresse von der aus geladen bzw. in die geschrieben werden soll. Der Inhalt der Register wird dabei nicht verändert.

Zum Beispiel wird aus der MSP430 Instruktion

ADD 2(R5), 4(R6)

folgende LLVM IR Instruktionsfolge

%r5_TMP_1 = add i16 %r5, 2	→ addieren des Offsets auf Registerinhalt
%r5_TMP_2 = load i16, i16 %r5_TMP_1	→ Laden von der zuvor berechneten Adresse
%r6_TMP_1 = add i16 %r6, 4	→ addieren des Offsets auf Registerinhalt
%r6_TMP_2 = load i16, i16 %r6_TMP_1	→ Laden von der zuvor berechneten Adresse
%TMP_1 = add i16 %r5_TMP_2, %r6_TMP_2	→ Addieren der geladenen Adressinhalte
store i16 %TMP_1, i16 %r6_TMP_2	→ speichern des Ergebnisses

Also wird für jedes benutze Register zunächst die passende Funktion zum Vorbereiten der Instruktion aufgerufen.
Dabei muss nur für den Register Mode und den Immediate Mode keine extra Funktion durchlaufen werden. Alle anderen Adressing Modes benötigen zusätzliche Instruktionen um diese in LLVM IR darstellen zu können.
Ist dies geschehen wird anschließend die finale Funktion zum liften des eigentlichen Befehls aufgerufen.
Handelt es sich um das normale Instructionset wird *liftBinRegister(…)* aufgerufen. Falls es sich um einen emulierten Befehl handelt wird *liftEmulOp(…)* aufgerufen.

Zusätzlich wird in der *prepareForLift* Funktion noch ein String erstellt der die alte MPS430 Instruktion enthält. Dieser wird letztendlich für die Metadaten benötigt.

Blick in die Funktionen: liftBinRegister

Verantwortliche: Leon Mrosewski
Optimierungen: Darius Happe, Jan Van Wickern

In dieser Funktion wird letztendlich die LLVM IR Instruktion erstellt die die MSP430 Instruktion darstellt.
Allerdings nur für die Befehle die direkt vorhanden sind. Für die Emulierten Befehle gibt es eine weitere Funktion.

Zunächst wird anhand des übergeben Opcodes eine LLVM IR Instruktion mittels *BinaryOps* erstellt.
Dies hat die Form

Instruction::BinaryOps instruction = Instruction::Add

Erstellen einer Add Instruktion

Anschließend werden die benötigen Register (welche in LLVM IR schon existieren und in der *prepareForLift(…)* erstellt wurden) aus der Register Map geladen.

Nun wird, abhängig von Adressing Mode, die eigentliche Instruktion erstellt. Zu unterscheiden ist dabei zwischen dem Speichern des Ergebnisses in einem Register oder an einer berechneten Speicherstelle. Im ersten Fall muss einfach nur die Instruktion erstellt werden und der Vorgang ist abgeschlossen.
Falls jedoch an eine Speicheradresse geschrieben werden muss, wird erst die eigentliche Instruktion erstellt und im Anschluss wird das Ergebnis an die berechnete Speicheradresse mittels einer *Store* Instruktion geschrieben. In diesem Schritt wird die erstelle Instruktion auch letztendlich an den aktuellen Basicblock angehängt.

Nun wird noch die *appendMetadata(…)* Funktion aufgerufen um die benötigten Metadaten anzuhängen.

Ein Sonderfall stellt jedoch die MSP430 *Mov* Instruktion dar.
Es gibt keine LLVM IR Instruktion die diese Funktion genau nachbildet. Daher wird hier die Add Instruktion genutzt. Es wird also der Wert der in das Zielregister geschrieben werden soll mit 0 Addiert und dann im LLVM IR Register gespeichert.

Aus zum Beispiel

MOV #5, R5

wird also in LLVM IR

%r5 = add i16 0, 9

Blick in die Funktionen: liftEmulOp

Verantwortliche: Leon Mrosewski

Diese Funktion hat eigentlich die gleiche Funktionalität wie *liftBinRegister* behandelt jedoch nur emulierte Befehle.
Emulierte Befehle des MSP430 können durch eine Folge von “normalen“ Befehlen aus dem MSP430 Instructionsset dargestellt werden.

Im allgemeinen ist der Ablauf genau der selbe wie in *liftBinRegister*.

Es wird der Opcode überprüft und die passende Instruktion erstellt.
Anschließend wird passend zum Adressing Mode eine LLVM IR Instruktion erstellt die dann im Basicblock angehängt wird.