Das HLS-Framework LegUp – Aufbau, Funktionsweise und Anwendung

Florian Mayer

8. Juni 2016

Inhaltsverzeichnis

Zusammenfassung

In dieser Arbeit soll mithilfe des HLS-Frameworks LegUp C-Programmcode zu Hardware synthetisiert werden. Um die Funktionsweise von LegUp besser demonstrieren zu können, wird zunächst LegUp selbst detailliert vorgestellt und der Begriff HLS geklärt. Das letzte Kapitel beschreibt sodann die Lösung eines praktischen Problems mithilfe von LegUp-C-Synthese.

LegUp unterstützt mehrere Toolchain-Arbeitsmodi, auf die im zweiten Kapitel näher eingegangen wird.

Besonderes Ziel dieser Arbeit ist es, die Transformationsschritte zu dokumentieren, die auf ein C-Programm angewandt werden müssen, damit es zu einem korrekten RTL-Modell wird. Hierbei sind allerdings nur die Verarbeitungsstufen relevant, die unmittelbar zu LegUp gehören. Insbesondere irrelevant sind alle Abstraktionsstufen unterhalb oder gleich der RTL-Ebene.

Die wichtigesten beteiligten HLS-Konzepte werden im Legup-Kapitel ebenfalls angesprochen, doch nicht genauer erläutert. Diese Arbeit möchte als Folge dessen eine gute Referenz und Litaratursammlung sein, die für weitere Recherchen im Kontext HLS herangezogen werden kann.

1 Einführung in die HLS

High Level Synthese bezeichnet die Transformation von Code in einer imperativen, sequenziellen Sprache zu Hardware. LegUp ist ein Tool, welches HLS von ANSI-C zu Verilog ermöglicht. Um den Begriff HLS genauer erörtern zu können, fehlen zunächst die Voraussetzungen, nämlich genaueres Verständnis der Abstraktionshierarchie, die sich im Laufe der Entwicklung von EDA-Werkzeugen in der Elektro- und Informationstechnik etabliert hat.

1.1 In welche Abstraktionshierarchie gliedert sich HLS?

Clifford Wolf, der Autor des Synthesis-Toolkits *Yosys* schlägt in seinem Manual (vgl. [1]) zur Software (informell) die Beispielhierarchie in Abbildung 1 vor. Die jeweiligen Pfeile zwischen den Schichten stellen Transformationen dar. Im Kontext dieser Arbeit (und der EDA) nennt man diese Transformationen Synthese, bzw. Synthetisierung. In den folgenden Unterkapiteln werden einige Schichten und Transformationen genauer beleuchtet.

1.1.1 Systemlevel

Die Systemebene stellt derzeit die höchste Abstraktionsstufe dar. Hier geht es darum, Syteme von sehr großer Komplexität (z.B. das Kommunikationsnetz und die Beschaffenheit der verbundenen Subsysteme in Autos) überhaupt einmal entwerfen zu können. In [5] wird diese Stufe wie folgt definiert: “abstractions in order to increase comprehension about a system, and to enhance the probability of a successful implementation of functionality in a cost-effective manner”. Beim Systemlevel-Design ist daher die Grundidee zunächst das System grobgranular zu beschreiben. Dieses Ziel wird entweder mit graphischen Modellierungsmitteln oder mithilfe von formalen Sprachen erreicht. Beispiele für Technologien aus dieser Schicht sind:

- *SystemC*
- *Matlab*, z.B. mit dem Toolkit Simulink

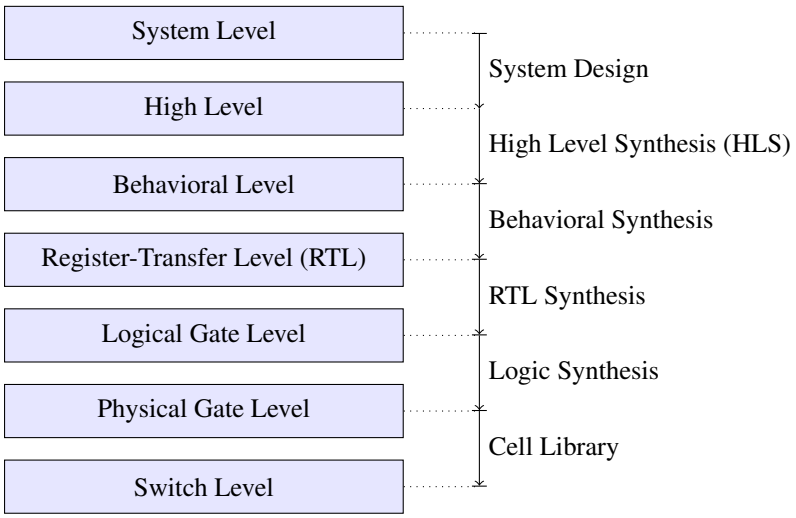

Abbildung 1: Abstraktionshierarchie

- *SysML*

SystemC ist kein reines Systemlevel Designwerkzeug, da es auch zur Systemsynthese und Simulation verwendet werden kann (vgl. [6]). Matlab ist eine Allzweckprogrammiersprache, die jedoch mithilfe der Simulink Programmpakete in ein Simulations- und Systemlevel-Desingwerkzeug verwandelt werden kann.

Informell: Auf dem Systemlevel modelliert man die Interaktion von Komponenten, die für sich selbst jeweils unabhängige digitale/analoge Systeme darstellen, deren Komplexität mit der Komponentenkomplexität auf Motherboards oder Microkontrollerschaltungen zu vergleichen ist.

1.1.2 Behavioural Level und High Level

Die enschlägige Literatur bezeichnet HLS häufig auch als *algorithmic level*. Auf dieser Ebene wird nämlich mithilfe von herkömmlichen imperativen, sequenziellen Programmiersprachen modelliert. Sequenzielle Algorithmen lassen sich aber auch auf Behavioral level (BL) beschreiben, wozu also HLS?

In traditionellen BL-Sprachen wie z.B. Verilog oder VHDL, müssen Designer nach wie vor alle Feinheiten des RTL- Designs kennen und verstehen. Algorithmische Modellierungen sind in VHDL ausschließlich innerhalb eines speziellen "concurrent-statements" möglich: Dem "process-statement".

Das folgende Listing zeigt einen n-Bit Counter:

Listing 1: N-Bit Counter

```
library ieee;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_1164.all;

entity counter is
   generic(n : natural := 4);
   port(clk    : in  std_logic;
        cnt    : in  std_logic;
        clear  : in  std_logic;
        outCnt : out std_logic_vector(n-1 downto 0));
end counter;

architecture behaviour of counter is
   signal temp : std_logic_vector(n-1 downto 0);
begin
   outCnt <= temp;

   process(clk, count, clear)
   begin
      if clear = '1' then
         temp <= '0'
      elsif (clk = '1' and clk'event) then
         if cnt = '1' then
            temp <= temp + 1;
         end if;
      end if;
```

```
    end process;
end behaviour;
```

Der Code innerhalb des `process`-Statements wird je nach Beschaffenheit der *sensitivity list* entweder zu einem Schaltwerk oder einem Schaltnetz synthetisiert. Ein Schaltwerk besitzt einen internen Status (also einen Speicher, der mithilfe von Registern realisiert wird), wohingegen das Schaltnetz keinen internen Speicher hat.

In VHDL müssen die (möglicherweise) algorithmisch/behavioural beschriebenen Komponenten immer noch manuell miteinander verbunden werden. Dabei stellt, neben der korrekten Verdrahtung der Komponenten, die korrekte Taktung ein ebenso großes Problem dar. Die sog. statische Timing-Analyse liefert Auskunft über die maximale Taktrate eines digitalen Systems, muss zudem manuell erstellt und ausgewertet werden.

Das folgende VHDL Codebeispiel beschreibt eine weitere Feinheit des BL-Designs: Nutzung von Block-RAM.

Listing 2: RAM-Block in VHDL

```
architecture Behavioral of seven_segment is
    type segment_data_t is array(3 downto 0) of
       std_logic_vector(7 downto 0);
    signal segment_data: segment_data_t;
    signal counter: unsigned(1 downto 0);
begin
    -- [...] previous code
    picoblaze_interface: process(clock, reset)
    begin
        if reset = '1' then
            segment_data <= (others => (others => '0'));
        elsif rising_edge(clock) then
            if write_enable = '1' then
                segment_data(conv_integer(
                    unsigned(address))) <= bit_pattern;
            end if;
        end if;
    end process picoblaze_interface;
end Behavioural;
```

Manche FPGA-Architekturen bieten sog. RAM-Blöcke an. Der obige VHDL-Code modelliert genau so einen RAM-Block inclusive Adress-en- bzw. Decoder. Für das ungeschulte Auge ist auf den ersten Blick nicht ersichtlich, dass hier Block-RAM synthetisiert wird. Auch war die Semantik dieses Codestücks nicht immer standardisiert (vgl. Kapitel 6.5 in [4]). Aktuelle Synthesetools synthetisieren natürlich keinen RAM-Zugriff, wenn kein RAM auf dem Ziel-FPGA vorhanden ist. In diesem Fall werden herkömmliche Register zusammengeschaltet. Die Feinheit besteht nun darin, dass das Timing von Register- und Block-RAM-Zugriffen höchst ungerschiedlich ist.

1.1.3 Register Transfer Level

Auf dem RTL ist das Design durch kombinatorische Datenpfade und Register beschrieben. Die Abbildung 2 zeigt ein extrem einfaches RTL-Design.

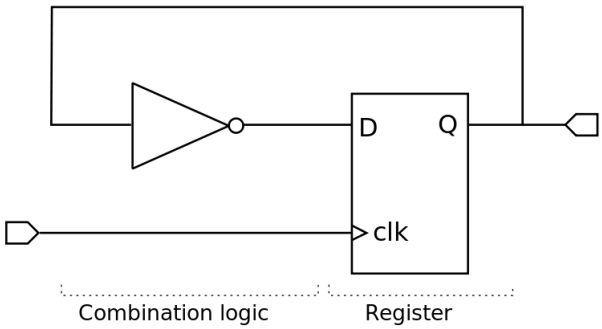

Abbildung 2: RTL-Design

Nimmt man als kombinatorischen Pfad z.B. die Addition zweier Bits (A und B), lässt sich in VHDL folgendes Beispiel geben:

Listing 3: Reines RTL-Design in VHDL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

--
-- Hier Entity-Deklaration mit ports
-- fuer A, B, entity_result  und clock
--

architecture behav_add of adder is
    signal result: std_ulogic;
begin
    -- Die overload Resolution waehlt die
    -- fuer std_logic ueberladene
    -- Version der Funktion aus der
    -- library std_logic_arith aus
    result <= A + B; -- rein kombinatorisch

    register: process(clock) is
    begin
        entity_result <= result;
    end register;
end behav_add;
```

HDL-Sprachen wie VHDL oder Verilog, stellen Designs nicht nur auf Verhaltensebene, sondern auch dem RTL dar. Es ist dem Designer überlassen für die verschiedenen Komponenten die jeweilige Abstraktionsstufe zu wählen. In praktischen Designs werden aber oft beide Schichten gemischt verwendet. Außerdem gilt: Ein synchroner Zähler lässt sich z.B. ohne das `process`-Statement in VHDL nicht implementieren, wohingegen dies für Addierer, Subtrahierer, etc. sehr wohl möglich ist.

Damit ein VHDL-Design als reines RTL-Modell gelten kann, muss auf `process`-Statements so weit wie möglich verzichtet werden. Konkreter: `process`-Statements dürften nur zur Erzeugung einzelner Register, bzw. Registervektoren verwendet werden.

Diese Fallunterscheidung wirkt pedantisch und provoziert die Frage nach dem Sinn. Manche Optimierungen können nur auf RTL-Ebene effektiv durchgeführt werden. Die sog. "FSM-Detection" (vgl. Seite 14 [1]) sowie Optimierungsalgorithmen die Speicherblöcke oder größere wiederverwendbare Ressourcen erkennen, sind Beispiele (z.B. Multipliziererblöcke, Addierer, ...).

RTL-Designs werden mithilfe sog. Netzlisten repräsentiert. Die obige Abbildung 2 zeigt ein sehr einfaches Exemplar dieser Darstellungsform. Netzlisten können entweder grob- oder feinkörnig sein. Listen auf RTL-Ebene können Funktionseinheiten wie Addierer, Multiplizierer und Multiplexer enthalten, also insbesondere Komponenten mit mehr als einem Eingang. Netzlisten sind im mathematischen Sinn zyklische Graphen. Zyklen kommen z.B. an jedem direkt rückgekoppelten Schaltnetz/Schaltwerk vor. Funktionsblöcke in Netzlisten nennt man auch Zellen.

RTL-Netzlisten können durch einfache Ersetzungsmechanismen in feinkörnigere Netzlisten synthetisiert werden. Netzlisten dieser Art befinden sich auf dem Logical Gate Level und enthalten nur noch die primitivsten Zellentypen (AND, OR, NOT, NAND, XOR, 1-Bit D-Type Flip-flops, ...). Diese Komponenten zeichnen sich unter anderem dadurch aus, dass deren Eingänge jeweils nur ein Bit breit sind (vgl. Seite 15 [1]).

1.1.4 Weitere Ebenen

Die restlichen und detailreichsten Ebenen sind:

- *Logical Gate Level*
- *Physical Gate Level*
- *Switch Level*

Sie sind für weitere Erörterungen nicht mehr relevant und werden daher nicht weiter beschrieben.

1.2 Warum wurde als Quellsprache ANSI-C gewählt?

C hat als HLS-Quellsprache gegenüber modernen objektorientierten oder funktionalen Programmiersprachen zwei wesentliche Vorteile. Erstens "kann die Welt C", zweitens ist die Sprache äußerst minimal gehalten und dennoch algorithmisch sehr prägnant und hardwarenah.

Legup unterstütz nicht alle Sprachfunktionen von ANSI-C. Insbesondere sind in Legup-C Rekursionen, Funktionspointer, Aufrufe zu `malloc` und ähnlichen Systemroutinen illegal. Funktionen, Arrays, Structs, globale/lokale Variablen, ausgewählte Pthread-Routinen, Gleitpunktoperationen und Zeiger bzw. Zeigerarithmetik sind aber valide Bestandteile der Untermenge (vgl. FAQ in [2]).

1.2.1 Die Welt spricht C

Fast keine andere Programmiersprache hat einen derart großen Bekanntheitsgrad wie C. Diesen Umstand hat C unter anderem seiner langen Existenz zu verdanken; Kernighan und Ritchie spezifizierten in ihrem Buch "The C Programming Language" bereits 1978 den ersten C-Dialekt (K&R1). Auch die Tatsache, dass Microcontrollerhersteller für jede neue Version ihrer Controller eine Variante des GCC herausbringen müssen um am Markt bestehen zu können, trägt dazu bei. Ebenso spielen Betriebssysteme eine signifikante Rolle, da zumindest die drei meistbenutzten Betriebssysteme Linux (BSD, ...), Mac OS X und Windows in C geschrieben sind, bzw. deren Systeminterfaces mithilfe von puren C-API's exportiert werden. Das zwingt natürlich Universitäten mit einer Informatik- oder bspw. Elektrotechnikfakultät dazu, diverse C-Kurse anzubieten, insbesondere dann, wenn Fächer mit hohem Praxisbezug angeboten werden.

Im Jahr 1989 verabschiedete das ANSI den ersten Offiziellen C-Standard, der unter anderem die synonymen Begriffe ANSI-C, C89 und ISO C90 trägt. Die zweite Auflage des Buchs "The C Programming Language" reflektiert die Änderungen dieses Standards und ist in dieser Version heute immer noch im Handel erhältlich. Aufgrund des schieren Alters dieses Standards, gibt es heute keinen Praxisrelevanten C-Compiler mehr, der nicht mindestens ANSI-C problemlos übersetzen kann — Probleme gibt es in der Praxis natürlich trotzdem, vor allem beim Übersetzten von systemabhängigem Code.

1.2.2 Minimalismus

Dieses Attribut lässt sich leider nur unscharf definieren, da C zum Zeitpunkt der Erscheinung sicherlich nicht als so schlank galt, wie heute im Vergleich zu Sprachen wie C++, Java oder gar Haskell, deren Featureset beachtlich angewachsen ist. *Minimalismus* bedeutet in diesem Kontext:

- Spärliche Abstraktionsmechanismen (Keine vernünftigen ADT's möglich, kein Modulsystem)
- Keine lokalen Funktionen (geschweige denn Closures)
- "easygoing" Typsystem; weakly typed language (keine Array-Bounds-Checks, keine Laufzeit-Checks, implizite Typecoercions, Void-Pointer, ...)
- Keine Continuations, keine Exceptionmechanismen
- Funktionen geben lediglich einen Wert eines Typs zurück
- Lediglich 32 Schlüsselwörter (in ANSI-C). Zum Vergleich: C++ hat 90 (vgl. [7])
- Keine Operator- sowie Funktionsüberladung (vgl. [8])
- Keine Vererbung und ebenso keine Datentyppolymorphie (vgl. [8])

All dies führt dazu, dass die Semantik der Sprache äußerst klar und einfach zu lernen ist. Ganz offensichtlich ist jedoch auch, dass semantische Einfachheit keinen guten programmierten Code impliziert. Dennoch lässt sich daraus sehr wohl folgern, dass C-AST's wesentlich einfacher verarbeitbar sind, als z.B. abstrakte Syntaxbäume der Sprachen Java oder C++.

Warum ist Minimalismus wichtig? Imperative, logische, funktionale oder objektorientierte Programmiersprachen haben eins gemeinsam: Ihre Zielsprachen sind ausschließlich

Assemblerdialekte verschiedener Register- oder Stackmaschinen mit sequenzieller Ausführung, Arbeitsspeicher und optionaler

Bei der HLS gilt dies nicht. Die Hauptzielsprachen, VHDL oder Verilog, werden nich Register- oder Stackbasiert weiterverarbeitet. VHDL und Verilog synthetisieren nämlich, wie in der Einführung erwähnt, zu Graphenstrukturen (Netzlisten), anstatt zu linearem Code wie Assembler. Weiterhin sind fast alle Hardware-Designs freistehend und kommen nicht in den Genuss vorgefertigter IO-Hilfsroutinen.

Die Algorithmen für die Synthese von C zu VHDL/Verilog (oder auch direkt in Netzlisten) sind hochkomplex, weslhalb man die Implementierung nicht unnötig durch die Interpretation manchmal ohnehin ungewünschter Features objektorientierter Sprachen verkomplizieren will. Das Ziel ist den AST der jeweiligen Quellsprache so einfach wie möglich zu halten.

2 Legup

2.1 Allgemeines

Legup ist ein HLS-Forschungsprojekt der Universität Toronto, welches unter einer sehr restriktiven Open-Source-Lizenz erhältlich ist. Legup darf der Lizenz zufolge nicht kommerziell benutzt oder weiterentwickelt werden.

Da es sich um ein Forschungsprojekt handelt, gibt es zum Zeitpunkt dieser Arbeit, noch keinen eigentständigen Installer. Die auf der Webseite empfohlene Art der Benutzung erfolgt durch eine virtuelle Appliance (eine vorgefertige Virtualbox-VM).

2.2 Eingabesprache

Legup akzeptiert ANSI-C als Eingabesprache. Im Gegensatz zu z.B. SystemC, werden keine speziellen Annotationen, Pragmas oder Schlüsselwörter für die Annotation diverser Kontrollstrukturen gebraucht; simples ANSI-C genügt (vgl. FAQ in [2]).

- Funktionspointer,
- dynamisches Speichermanagement (`malloc`, `calloc`, ...),
- Rekursionen
- und die meisten Systembibliotheken

sind im Quelltext nicht erlaubt!

Ein Beispiel für validen LegUp Input ist dieses in der LegUp-VM vorhandene Beispiel:

Listing 4: Synthetisierbare Arrayverarbeitung

```
#include <stdio.h>
int array[2][2][3] = {
  {
    {1, 2, 3},
    {4, 5, 6}
  } , {
    {7, 8, 9},
    {10, 11, 12}}};
```

```
int fct(int *array, int size) {
    int result = 0;
    int i;
    for (i = 0; i < size; i++) {
        result += array[i];
    }
    return result;
}

int main() {
    int result = 0;
    int a, b, c;
    for (a = 0; a < 2; a++) {
        for (b = 0; b < 2; b++) {
            for (c = 0; c < 3; c++) {
                result += array[a][b][c];
            }
        }
    }

    result += fct((int *)array, 12);

    printf("Result: %d\n", result);
    if (result == 156) {
        printf("RESULT: PASS\n");
    } else {
        printf("RESULT: FAIL\n");
    }
    return result;
}
```

Aufrufe zu `printf()` sind explizit erlaubt und werden in der resultierenden Verilogdatei zu `display`-Statements konvertiert. Verilogsynthesetools ignorieren diese Anweisungen, wohingegen sie von Simulatoren ausgeführt werden. Somit können Algorithmen zunächst mithilfe von `printf`-Aufrufen debugged und dann in einem Verilogsimulator getestet werden (vgl. Verilogcode in 3.4.2).

Ein Beispiel für die in SystemC notwenigen Annotationen liefert das folgenden Listing.

Listing 5: SystemC Beispiel

```
#include "systemc.h"

SC_MODULE(adder)            // module (class) declaration
{
    sc_in<int> a, b;        // ports
    sc_out<int> sum;

    void do_add()           // process
    {
                            //or just sum = a + b
        sum.write(a.read() + b.read());
```

```
    }

    SC_CTOR(adder)          // constructor
    {
        SC_METHOD(do_add);  // register do_add to kernel
        sensitive << a << b;// sensitivity list
    }
};
```

SC_MODULE, SC_METHOD und SC_CTOR sind z.B. Makros aus der systemc.h Headerdatei (vgl. [9]).

2.3 Syntheseabläufe

LegUp unterstützt hauptsächlich zwei Syntheseflows: *Pure hardware* und *Hybrid*. In beiden Fällen werden die benötigten Werkzeuge mittels TCL-Skripten und Makefiles orchestriert.

2.3.1 Der Hardware Flow

Hierbei werden die in C verfassten Algorithmen und Designbeschreibungen in ein semantisch äquivalentes Verilog-Designfile überführt. Der *pure hardware flow* kann mithilfe eines TCL-Skripts ausführlich gesteuert werden (vgl. Kapitel 10 [2]). Z.b. können in den Configurationsskripten experimentelle Synthesefeatures an- und ausgeschalten oder Scheduling-, Allokations-, und Bindingparameter festgelegt werden.

2.3.2 Der Hardware Flow Im Detail

Um den Hardware-Designflow besser zu beschreiben, beziehen sich alle weiteren Ausführungen in diesem Abschnitt auf das C-Listing in 3.2. Wichtig: Die hier beschriebene Kommandoabfolge ist der Synthesestandard von LegUp; der gesamte Toolchainflow kann vollständig angepasst werden. Solange die grobe Reihenfolge

ANSI-C $\rightarrow$ LLVM IR $\rightarrow$ Optimierter IR $\rightarrow$ Verilog

erhalten bleibt.

Legup unterstütz sog. Loop-Pipelinging. Damit die Pipeliningalgorithmen greifen können, müssen Schleifen mit einem Label versehen werden.

In Schritt 1 des Hardwareflows erzeugt das Perl-Skript *mark_labels.pl* aus dem Code

```
loop: for (i = 0; i < foo; i++){
  // weitere Statements [...]
}
```

den folgenden Ergebniscode:

```
loop: for (i = 0; i < foo; i++){
  __legup_label("loop");
  // weitere statements [...]
}
```

Der `__legup_label()`-Aufruf muss sich immer unmittelbar als erste Anweisung nach dem Beginn des Statementsblock der jeweiligen Schleife im Ergebniscode befinden. Da der C-Code von einem modifizierten C-Compiler in Zwischencode (LLVM-IR) übertragen wird, welcher leider die C-Labels nicht mehr beinhaltet, ist ein derartiger Hack notwendig.

Der zweite Schritt überführt die soeben annotierte Datei in LLVM-Zwischencode. Dies geschieht durch folgendes Kommando:

```
clang-3.5 array_labeled.c \
    -emit-llvm -c -fno-builtin -I ../lib/include/ \
    -m32 -I /usr/include/i386-linux-gnu -O0 -mllvm \
    -inline-threshold=-100 -fno-inline -fno-vectorize \
    -fno-slp-vectorize -o array.prelto.1.bc
```

Die Parameter weisen Clang dazu an (`-O0`) im niedrigesten Optimierungslevel 32 Bit (`-m32`) LLVM-IR-Code (`-mllvm`, `-emit-llvm`) zu generieren ohne dabei (`-fno-vectorize`, `-fno-slp-vectorize`) Vektorisierungen, oder (`-fno -inline`) Inlining vorzunehmen. Der Bytecode steht danach in der Datei `array.prelto.1.bc`

Im dritten Schritt wird der eben produzierte Bytecode mittels

```
../../llvm/Release+Asserts/bin/llvm-dis array.prelto.1.bc
```

in lesbaren ASCII llvm code umgesetzt. Das Format ist nach wie vor LLVM-IR.

In Aktion Nr. 4 tritt zum ersten Mal der IR-Code Optimizer `opt` in Erscheinung. `opt` ist ein Tool aus der LLVM Compilertoolchain mit dessen Hilfe diverse low-level Optimierungen durchgeführt werden können. Die Entwickler von LegUp haben diesem Optimierer weitere Optimierungsläufe hinzugefügt, die mit eigenen Compilerflags an- oder ausgeschaltet werden.

Der Aufruf

```
../../llvm/Release+Asserts/bin/opt -mem2reg \
    -loops -loop-simplify \
    < array.prelto.cv.bc > array.prelto.2.bc
```

produziert einen Zwischencode, bei dem alle RAM-Referenzen in Registerreferenzen (`-mem2reg`) umgewandelt wurden. LegUp entscheidet selbst darüber, welche Speicherreferenzen als Block-RAM und welche als Register realisiert werden. Weiterhin bedeuten `-loops` und `-loop-simplify` generische Schleifenoptimierungen.

In Schritt 5 werden zunächst standard link-time Optimierungen durchgeführt

```
../../llvm/Release+Asserts/bin/opt \
    -load=../../llvm/Release+Asserts/lib/LLVMLegUp.so \
    -legup-config=../legup.tcl  \
    -disable-inlining -disable-opt \
    -std-link-opts < array.prelto.linked.bc -o
        array.prelto.linked.1.bc
```

alle anderen Optimierungen sind dabei ausgeschalten (vgl. `-disable-inlining`, -disable-opt). Dieser Schritt unterscheidet sich nicht von den link-time Optimierungsläufen normaler Compilerläufen

Danach führt der Optimizer außschließlich pre-link-time Optimierungen durch.

```
../../llvm/Release+Asserts/bin/opt \
    -load=../../llvm/Release+Asserts/lib/LLVMLegUp.so \
    -legup-config=../legup.tcl  -disable-inlining \
    -disable-opt -legup-prelto \
    < array.prelto.linked.1.bc > array.prelto.6.bc
```

Beendet wird Schritt 5, indem noch einmal die standard link-time Optimierungen durchgeführt werden.

Schritt 6: Hier werden zwei Assembly-Bibliotheken hinzugelinkt. In liblegup sind Ersatzimplementierungen der Funktionen `memmove`, `memset` und `memcpy` enthalten und libm.bc stellt eine vollständige (von math.h unabhängige) Mathebibliothek zur Vefügung.

```
../../llvm/Release+Asserts/bin/llvm-link array.prelto.bc \
    ../lib/llvm/liblegup.bc \
    ../lib/llvm/libm.bc -o array.postlto.6.bc
```

Im darauf folgenden 7. Kommando werden mithilfe des Optimizers alle (aus libm.bc oder liblegup.bc) ungenutzten Funktionen entfernt. `-globaldce` entspricht dabei globaler *dead code elemination*.

```
../../llvm/Release+Asserts/bin/opt \
    -internalize-public-api-list=main -internalize \
    -globaldce array.postlto.6.bc -o array.postlto.8.bc
```

Schritt 8. Aufgrund der vorherigen Veränderungen, werden wieder link-time Optimierungen durchgeführt.

```
../../llvm/Release+Asserts/bin/opt \
    -load=../../llvm/Release+Asserts/lib/LLVMLegUp.so \
    -legup-config=../legup.tcl  -disable-inlining \
    -disable-opt -instcombine \
    -std-link-opts < array.postlto.8.bc -o
       array.postlto.bc
```

Die vorletzten zwei Aktionen führen sog. iteratives-modulo Scheduling aus (die Begriffe werden nicht geklärt)

```
../../llvm/Release+Asserts/bin/opt \
    -load=../../llvm/Release+Asserts/lib/LLVMLegUp.so \
    -legup-config=../legup.tcl  -disable-inlining \
    -disable-opt -basicaa -loop-simplify -indvars2 \
    -loop-pipeline array.postlto.bc -o array.1.bc
```

```
../../llvm/Release+Asserts/bin/opt \
    -load=../../llvm/Release+Asserts/lib/LLVMLegUp.so \
    -legup-config=../legup.tcl  -disable-inlining \
    -disable-opt -instcombine array.1.bc -o array.bc
```

Im letzten Schritt wird sodann der voll optimierte LLVM-Bytecode mithilfe von

```
../../llvm/Release+Asserts/bin/llc \
    -legup-config=../legup.tcl  \
    -march=v array.bc -o array.v
```

in ein synthetisierbares Verilog Design übersetzt. `llc` ist der Bytecodecompiler des LLVM-Projekts. Die LegUp Entwickler haben dort ein zusätzliches Backend eingefügt, sodass anstatt von Assembly, Verilogcode erzeugt werden kann. Die Option hierfür ist `-march=v`.

2.3.3 Custom Verilog Flow

Mit LegUp lassen sich Hardware-only Syntheseabläufe auch mit Handgeschriebenen Verilogmodulen erweitern. Beispielsweise könnte eine Funktion die derzeit noch zu kompliziert für Legup ist, manuell in Verilog implementiert werden. Der Programmierer muss sich dabei natürlich an gewisse Konventionen halten (vgl. 3.4.1).

Im C-Programm muss der Programmierer zudem die Funktion annotieren, die er selbst nach Verilog transkribieren möchte. Im C-Code selbst sähe dies so aus:

```
void __attribute__((noinline)) __attribute__((used))
    foobar(int baz, ...) {...}
```

2.3.4 Der Hybrid Flow

Es können auch nur Teile von C-Programmen für die Synthese ausgewählt werden, während die restlichen Funktionen auf einer herkömmlichen CPU, oder einem Softcore ausgeführt werden. Ein Programm das z.B. aus einer Funktion für die Berechnung der Mandebrot-Menge — ein höchst parallelisierbares Problem — und einer Funktion zur Kontrolle dieser Berechnung besteht, kann mithilfe des Hybridflows folgendermaßen auf einem FPGA ausgeführt werden:

Zunächst erfolgt die Auswahl welche Funktion direkt in eine Hardwareschaltung und welche zu Binärcode transformiert werden soll. Nun synthetisiert Legup die ausgewählte Funktion und den Tiger MIPS Prozessor (ein MIPS-Softcore der Universität von Cambridge). Weiterhin compiliert LegUp den restlichen C-Code zu einem MIPS-Binärprogramm. Bei diesem Übersetzungsvorgang ersetzt LegUp noch die Funktionsaufrufe durch Wrappercalls, die die Daten an das synthetisierte Hardwaremodul weitergeben (vgl. Kapitel 2.4 in [2]).

Wie dem LegUp-Whitepaper (vgl. Figure 1 in [11]) zu entnehmen ist, planen die Entwickler die Auswahl der rechenintensiven Funktionen zu automatisieren, sodass der gesamte Hybridflow ohne manuelle Auswahl der zu synthetisierenden Funktionen ablaufen kann. Hierbei würde ein modifizierter Tiger MIPS Softcoreprozessor auf dem Rechner des Entwicklers laufen und zunächst eine bestimmte Zeit lang Profilingdaten sammeln.

2.4 Was passiert mit den C-Primitiven?

In diesem Abschnitt wird exemplarisch dargestellt wie LegUp die diversen C-Primitiven behandelt und in Verilog-Code umsetzt.

2.4.1 Funktionen

Die Funktion

```
void dispatch_instruction(uint16_t instruction);
```

wird von Legup in folgende Verilog Moduldeklaration umgewandelt

Listing 6: Syntheseergebnis für Funktion dispatch_instruction

```
module dispatch_instruction (
    clk,
    clk2x,
    clk1x_follower,
    reset,
    start,
    finish,
    memory_controller_waitrequest,
    memory_controller_enable_a,
    memory_controller_address_a,
    memory_controller_write_enable_a,
    memory_controller_in_a,
    memory_controller_size_a,
    memory_controller_out_a,
    memory_controller_enable_b,
    memory_controller_address_b,
    memory_controller_write_enable_b,
    memory_controller_in_b,
    memory_controller_size_b,
    memory_controller_out_b,
    arg_instruction
);

parameter  LEGUP_0 = 1'd0;
parameter  LEGUP_F_dispatch_instruction_BB__0_1 = 1'd1;
parameter [8:0] tag_offset = 9'd0;
parameter ['MEMORY_CONTROLLER_ADDR_SIZE-1:0]
    tag_addr_offset = {tag_offset, 23'd0};

input  clk;
input  clk2x;
input  clk1x_follower;
input  reset;
input  start;
output reg  finish;

input  memory_controller_waitrequest;
```

```
output reg  memory_controller_enable_a;
output reg  memory_controller_write_enable_a;
output reg [`MEMORY_CONTROLLER_ADDR_SIZE-1:0]
    memory_controller_address_a;
output reg [`MEMORY_CONTROLLER_DATA_SIZE-1:0]
    memory_controller_in_a;

output reg [1:0] memory_controller_size_a;
input [`MEMORY_CONTROLLER_DATA_SIZE-1:0]
    memory_controller_out_a;

output reg  memory_controller_enable_b;
output reg  memory_controller_write_enable_b;
output reg [`MEMORY_CONTROLLER_ADDR_SIZE-1:0]
    memory_controller_address_b;
output reg [`MEMORY_CONTROLLER_DATA_SIZE-1:0]
    memory_controller_in_b;

output reg [1:0] memory_controller_size_b;
input [`MEMORY_CONTROLLER_DATA_SIZE-1:0]
    memory_controller_out_b;

input [15:0] arg_instruction;
reg  cur_state;
reg  next_state;
```

Im Folgenden werden einige dieser Deklarationen näher beleuchtet:

- `clk`, `clk2x` sind der Systemtakt und der doppelte Systemtakt
- `clk1x_follower` ist ein um 180° phasenverschobener Systemtakt
- `start` ist das Signal der Legup-Runtime-FSM. Wenn dieses Signal am Modul positiv anliegt, muss es mit der Ausführung beginnen. Hiermit wird die Funktionsaufrufsemantik nachgebaut. Analog verhält es sich mit `finish`. `finish` baut die Funktionsrückkehr nach.
- `memory_controller_enable_a` kündigt Lesen oder Schreiben im aktuellen Zyklus an
- `memory_controller_write_enable` Wenn 1 → schreibender Zugriff, ansonsten lesender.
- `arg_instruction` Der Parameter der entsprechenden C-Funktion. Dargestellt als 16-Bit Array (wegen `uint16_t` in der C-Deklaration)
- `memory_controller_address_a` ist die Adresse der Speicherstelle
- `reg cur_state` und `reg next_state` sind interne register

2.4.2 If-Else Statements

Gegeben sei das triviale C-Snippet:

```c
#include <stdint.h>
#include <stdio.h>

int main(int argc){
    if (argc == 3){
        printf("foo\n");
    } else {
        printf("bar\n");
    }

    return 0;
}
```

Clang übersetzt dieses Codestück in folgende LLVM-IR:

```
@.str = private unnamed_addr constant [5 x i8]
   c"foo\0A\00", align 1
@.str1 = private unnamed_addr constant [5 x i8]
   c"bar\0A\00", align 1

; Function Attrs: noinline nounwind
define i32 @main(i32 %argc) #0 {
  %1 = icmp eq i32 %argc, 3
  br i1 %1, label %2, label %4

; <label>:2
  %3 = call i32 (i8*, ...)* @printf(i8* getelementptr
     inbounds ([5 x i8]* @.str, i32 0, i32 0)) #2
  br label %6

; <label>:4
  %5 = call i32 (i8*, ...)* @printf(i8* getelementptr
     inbounds ([5 x i8]* @.str1, i32 0, i32 0)) #2
  br label %6

; <label>:6
  ret i32 0
}
```

Dieser Assemblycode ist sehr einfach zu verstehen. Für den generierten Verilog trifft das allerdings nicht mehr zu. Die produzierte Moduldefinition wäre zu groß um sie hier abzudrucken. Die Moduldeklaration ist analog zum Beispiel im Abschnitt Funktionen und muss daher auch nicht erläutert werden.

LegUp übersetzt das `if-else`-Statement in eine FSM mit folgenden (relevanten) Zuständen:

- `LEGUP_F_main_BB__2_2`
- `LEGUP_F_main_BB__4_3`
- `LEGUP_0`

```
/* Platzhalter fuer die Moduldeklaration mit Signal- und
   Registerspezifikationen */
```

```
/* Unsynthesizable Statements */
always @(posedge clk)
    if (!memory_controller_waitrequest) begin
    /* main: %2*/
    /*  %3 = call i32 (i8*, ...)*
        @printf(i8* getelementptr inbounds
        ([5 x i8]* @.str, i32 0, i32 0)) #2 */
    if ((cur_state == LEGUP_F_main_BB__2_2)) begin
        $write("foo\n");
    end
    /* main: %4*/
    /*  %5 = call i32 (i8*, ...)*
        @printf(i8* getelementptr inbounds
        ([5 x i8]* @.str1, i32 0, i32 0)) #2 */
    if ((cur_state == LEGUP_F_main_BB__4_3)) begin
        $write("bar\n");
    end
end
```

Das Schlüsselwort `always` leitet in Verilog Sensitivitätslisten und BL-Code (Behaviour Level) ein. Es ist analog zu dem VHDL-Schlüsselwort `process`. In diesem Fall wird eine Taktflankengesteuerte synchrone Schaltung synthetisiert. Das `write` Statement selbst ist freilich nicht synthetisierbar. Der obige Prozess überprüft also jeden Takt ob der momentanen Zustand `cur_state` einen für die Ausgabe von `foo` oder für `bar` notwenigen Wert angenommen hat und tätigt die Ausgabe sofern dies zutrifft.

Der folgende Codeausschnitt ist für den korrekten Folgezustand der Modul-FSM zuständig:

```
always @(posedge clk) begin
    if (reset == 1'b1)
        cur_state <= LEGUP_0;
    else if (memory_controller_waitrequest == 1'd1)
        cur_state <= cur_state;
    else
        cur_state <= next_state;
end
```

Dieses Listing mach auch klar wozu das Signal `..._waitrequest` dient: Ist es gesetzt, so hält die FSM ihren aktuellen Zustand.

Der nächste Prozess ist nicht sequenziell, sondern repräsentiert lediglich die Verdrahtung eines 32-Bit-Komparators (`argc == 3`) mit dessen Operanden. Diese sind hier zu sehen:

```
/* koennte auch always @(arg_argc) sein */
/* @(*) nimmt alle gelesenen Signale in die
   sensitivity list auf! */
always @(*) begin
    /* main: %0*/
    /*   %1 = icmp eq i32 %argc, 3*/
    main_0_1 = (arg_argc == 32'd3);
end
```

main_0_1 ist ein Modulregister, das somit sofort beschrieben wird, wenn arg_argc am Moduleingang anliegt!

An dieser Stelle fehlen nun noch einige Details, die hier jedoch aus Mangel von Relevanz vernachlässigt werden. Die hier beschriebenen Ergebnisse können natürlich sehr einfach in der Legup-VM nachvollzogen werden.

2.4.3 Schleifen

Zu unterscheiden sind hier zwei Typen von Schleifen: For- und While.

Forschleifen: Im Rahmen dieser Arbeit wird folgende vereinfachende Syntax für Forschleifen angenommen.

```
for (var = number; var < number; increment-expression)
    { ... }
```

Gegeben sei folgendes C-Snippet:

```
#include <stdio.h>
int array[32] = { 0 };

int main(){

    for (i = 0; i < 32; i++){
        array[i] = i;
    }

    return 0;
}
```

Der LLVM-IR-Code repräsentiert das obige Programm wie folgt:

```
@array = internal global [32 x i32] zeroinitializer,
   align 4

; Function Attrs: noinline nounwind
define i32 @main() #0 {
  br label %1

; <label>:1
  %2 = phi i32 [ 0, %0 ], [ %5, %4 ]
  %exitcond = icmp eq i32 %2, 32
  br i1 %exitcond, label %6, label %3

; <label>:3
  %scevgep = getelementptr [32 x i32]*
             @array, i32 0, i32 %2
  store i32 %2, i32* %scevgep, align 4
  br label %4

; <label>:4
```

```
  %5 = add nsw i32 %2, 1
  br label %1

; <label>:6
  ret i32 0
}
```

Die Schleifenlogik wird in LLVM einzig durch die Instruktion `phi` abstrahiert. Diese hat folgende Syntax:

```
%indvar = phi i32 [ 0, %LoopHeader ],
                  [ %nextindvar, %Loop ]
```

Im obigen Beispiel gibt es keinen `LoopHeader`, daher ist der dortige Wert auch `%0`. `%indvar` entspricht im obigen Beispiel. `%2`.

Die Semantik dieses Codes lässt sich grob so zusammenfassen:

- Rufe Instruktion `phi` auf mit den Parametern:
  - Schleifenbeginn: 0
  - Schleifenheader: n.a.
  - Folgeindex: Variable 5
  - Scheifensprunglabel: 1
- Überprüfe ob die Schleifenbedingung `i < 32` noch wahr ist
  - Falls ja:
    - Springe zu Label 3
    - Speichere den entsprechenden Schleifenindexwert in @array
    - Springe zu Label 4
    - Berechne den neuen indexwert
    - Springe zu Label 1
  - Falls nein:
    - Springe zu Label 6
    - Gebe den Wert 0 als Rückgabewert für @main

Den “Falls ja” bzw. “Falls nein” Teil der oben gezeigten Semantik, lässt sich abermals als FSM darstellen. LegUp tut zunächst genau das. Die von LegUp generierte State-Machine sieht wie folgt aus:

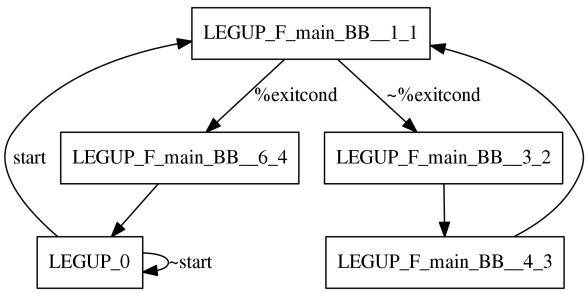

Abbildung 3: FSM für Forschleife

Solange `%exitcond` nicht wahr ist, befindet sich die Statemachine in einer Schleife mit der Zustandsfolge `1_1` → `3_2` → `4_3` → `1_1`. Diese Zustände werden dann von parallel arbeitenden `always`-Blöcken erkann.

Bspw. kümmert sich der Block

```
always @(*) begin
    /* main: %1*/
    /*   %2 = phi i32 [ 0, %0 ], [ %5, %4 ]*/
    if ((((cur_state == LEGUP_0)
          & (memory_controller_waitrequest == 1'd0))
          & (start == 1'd1))) begin
        main_1_2 = 32'd0;
    end
    /* main: %1*/
    /*   %2 = phi i32 [ 0, %0 ], [ %5, %4 ]*/
    else begin
        main_1_2 = main_4_5;
    end
end
```

um die korrekte Initialisierung der Schleifenvariable die im generierten Code den Namen `main_1_2` trägt.

Der nächst Block

```
always @(*) begin
    /* main: %1*/
    /*   %exitcond = icmp eq i32 %2, 32*/
    main_1_exitcond = (main_1_2_reg == 32'd32);
end
```

überprüft jeden Takt, ob die Schleifenbedingung bereits erfüllt ist und setzt ein internes Register, das vom FSM-`always`-Block zur Folgezustandsbestimmung verwendet wird.

Außerdem berechnet der Prozess

```
always @(*) begin
    /* main: %4*/
    /*   %5 = add nsw i32 %2, 1*/
    main_4_5 = (main_1_2_reg + 32'd1);
end
```

den Schleifenindex der nächsten Iteration. LegUp synthetisiert natürlich ebenfalls `always`-Blöcke für die Speicherzugriffe. Darauf wird aber nicht weiter eingegangen.

Whileschleifen: While Schleifen unterscheiden sich konzeptionell nicht von der Synthese von For-Schleifen, da im LLVM-IR die Unterschiede der beiden Formen bereits semantisch aufgelöst wurden.

2.4.4 Switch-Case Klauseln

Switch-Case Klauseln dürfen in C lediglich zur Compilezeit berechenbare Ausdrücke an den Case-Klauseln tragen. Der Ausdruck innerhalb des Switch-Körpers unterliegt natürlich nicht dieser Einschränkung.

```
#include <stdio.h>

int main () {
    volatile int x = 2;

    switch(x) {
    case 0:
        printf("case␣0\␣n␣" ) ;
        return 1;
        break ;
    case 1:
        printf ("case␣1\␣n␣" ) ;
        return 1;
        break ;
    default :
        return 2;
    }
}
```

Im obigen Beispiel ist die volatile Deklaration von x notwendig, da der Compiler ansonsten — je nach Optimierungsein- stellung — den gesamten Switch-Case-Block “wegoptimieren” würde.

Clang generiert für obiges Snippet folgenden LLVM-Zwischencode:

```
@.str = private unnamed_addr constant [8 x i8] c"case
    0\0A\00", align 1
@.str1 = private unnamed_addr constant [8 x i8] c"case
    1\0A\00", align 1

; Function Attrs: noinline nounwind
define i32 @main() #0 {
  %x = alloca i32, align 4
  store volatile i32 2, i32* %x, align 4
  %1 = load volatile i32* %x, align 4
  switch i32 %1, label %6 [
    i32 0, label %2
    i32 1, label %4
```

```
  ]

  %3 = call i32 (i8*, ...)*
       @printf(i8* getelementptr inbounds
       ([8 x i8]* @.str, i32 0, i32 0)) #2
  br label %7

; <label>:4
  %5 = call i32 (i8*, ...)*
       @printf(i8* getelementptr inbounds
       ([8 x i8]* @.str1, i32 0, i32 0)) #2
  br label %7

; <label>:6
  br label %7

; <label>:7
  %.0 = phi i32 [ 2, %6 ], [ 1, %4 ], [ 1, %2 ]
  ret i32 %.0
}
```

LegUp übersetzt nun diesen Assemblercode analog zum If-Else-Block in eine FSM. Ein Graph stellt die generierte FSM viel übersichtlicher dar als der korrespondierende Verilog Code:

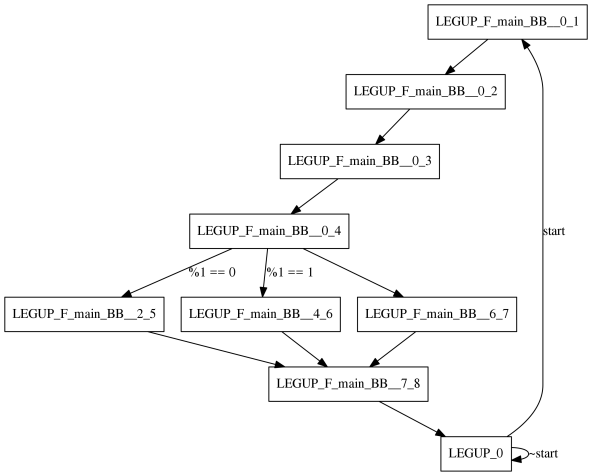

Abbildung 4: FSM für Switch-Case

Das Modulsignal `start` triggert dabei das Funktionsmodul und deren Ausführung. Dadurch wird ebenfalls die FSM in den ersten Inputstatus versetzt. Die Erläuterung aller Zustände dieses Automaten ist außerhalb des Rahmens dieser Arbeit. Wesentlich für das grobe Verständnis sind jedoch die Zustände `6_7`, `4_6` und `2_5`. Der erste wird angenommen falls `x==0`, der zweite falls `x==1` und der dritte im `default` Fall.

Die tatsächlichen Anweisungen der Caseklauseln werden mit einem parallel laufenden Schaltwerk getriggert.

Der `always`-Block

```
always @(posedge clk)
    if (!memory_controller_waitrequest) begin
    /* main: %2*/
    /*  %3 = call i32 (i8*, ...)*
        @printf(i8* getelementptr inbounds
        ([8 x i8]* @.str, i32 0, i32 0)) #2*/
    if ((cur_state == LEGUP_F_main_BB__2_5)) begin
        $write("case␣0\n");
    end
    /* main: %4*/
    /*  %5 = call i32 (i8*, ...)*
        @printf(i8* getelementptr inbounds
        ([8 x i8]* @.str1, i32 0, i32 0)) #2*/
    if ((cur_state == LEGUP_F_main_BB__4_6)) begin
        $write("case␣1\n");
    end
end
```

kontrolliert die Abarbeitung von nicht-synthetisierbaren Statements und der zweite Block

```
always @(*) begin
    /* main: %7*/
    /*   %.0 = phi i32 [ 2, %6 ], [ 1, %4 ], [ 1, %2 ]*/
    if (((cur_state == LEGUP_F_main_BB__2_5)
          & (memory_controller_waitrequest == 1'd0)))
            begin
        main_7_0 = 32'd1;
    end
    /* main: %7*/
    /*   %.0 = phi i32 [ 2, %6 ], [ 1, %4 ], [ 1, %2 ]*/
    else if (((cur_state == LEGUP_F_main_BB__4_6)
              & (memory_controller_waitrequest ==
                 1'd0))) begin
        main_7_0 = 32'd1;
    end
    /* main: %7*/
    /*   %.0 = phi i32 [ 2, %6 ], [ 1, %4 ], [ 1, %2 ]*/
    else begin
        main_7_0 = 32'd2;
    end
end
```

steuert die Zuweisung des richtigen Returnwerts. In letzterem Codeblock ist `main_7_0` ein privates Modulregister, dessen Wert im Status `7_8` dem output-Register `return_val` zugewisen wird.

2.4.5 Primitive Datentypen

Primitive Datentypen wie `int`, `char` oder `double` werden von Legup in typenlose Bitvektoren mit entsprechender breite übersetzt. Der in 3.4.1 angeführte Verilog-Code bestätigt dies,

da die dort enthaltene `input`-Deklaration

```
input [15:0] arg_instruction;
```

ganz klar den ersten Parameter der vorher deklarierten Funktion

```
void dispatch_instruction(uint16_t instruction);
```

darstellt. Da diese Funktion keinen Rückgabeparameter besitzt, gibt es keine korrespondierende `output`-Deklaration im Verilogdesign.

Allgemein gilt für die Bitbreiten der synthetisierten Datentypen: Es werden die im LLVM-Zwischencode verzeichneten Längenangaben genutzt. Das Kommando soll hier noch einmal aufgeführt werden:

```
clang-3.5 array_labeled.c \
    -emit-llvm -c -fno-builtin -I ../lib/include/ \
    -m32 -I /usr/include/i386-linux-gnu -O0 -mllvm \
    -inline-threshold=-100 -fno-inline -fno-vectorize \
    -fno-slp-vectorize -o array.prelto.1.bc
```

Das Flag `-m32` ist hier maßgeblich für die Bitbreiten der Primitiven verantwortlich. So bekommt z.B. der Typ `int` in LLVM-IR 32 Bit spendiert. Weiterhin lassen sich Typen wie `uint16_t` oder `int8_t` aus `stdint.h` zum exakten Steuern der Signedness und Bitbreite nutzen.

2.4.6 Speicherzugriffe und Speichermodell

Variablen können in C entweder global oder lokal im Quelltext angegeben werden. Lokale Variablen werden dabei immer auf den Stack-Frame der aufgerufenen Funktion gelegt und deren Inhalt geht nach Rückkehr dieser Funktion wieder verloren, bzw. ist undefiniert. Weiterhin gibt es dynamsich allozierten Speicher. Dynamische Speicheranforderungen sind zwar im Prinzip auch in Hardware umsetzbar, aber nicht ohne erheblichen zusätzlichen Platzbedarf. Man könnte pragmatischerweise einen in ANSI-C geschriebenen Memory-Manager für eingebettete Systeme zweckentfremden und alle `mallo`-Aufrufe (`stdlib.h`) durch Aufrufe an die Ersatzfunktion ändern. Diese Bibliothek muss allerdings ohne Funktionspointer auskommen, da solche noch nicht von Legup unterstützt werden.

Legup benutzt intern die sog *Points-To*-Analyse, um herauszufinden welcher Speicherbereich von welchen Funktionen verwendet werden. Sofern diese Untersuchung zeigt, dass das entsprechende Array lediglich von *einer* Funktion genutzt wird, so wird der Speicher dafür, ausschließlich in dem Verilogmodul der betreffenden Funktion instanziiert. Falls die Analyse fehlschlägt oder zeigt, dass der Datenspeicher von mehr als einer Funktion genutzt wird, legt Legup ihn global innerhalb eines eigenen Memory-Controllers an (vgl. Kapitel 3.2, 3.3 und 3.4 [2]).

2.4.7 Statisch initialisierte Arrays

Der Inhalt von statisch initialisierten Arrays wird dem Verilog Synthesetool mit *.MIF*-Dateien bereitgestellt.

Bei der HLS-Synthese des Array-Beispiels von 3.2, wird ein globales 3D-Array definiert:

```
int array[2][2][3] =
{
  {
    {1, 2, 3},
    {4, 5, 6}
  } , {
    {7, 8, 9},
    {10, 11, 12}
  }
};
```

Die dazugehörige *.MIF*-Datei sieht wie folgt aus:

```
Depth = 12;
Width = 32;
Address_radix = dec;
Data_radix = hex;
Content
Begin
0: 00000001;    -- array[0] = 1
1: 00000002;    -- array[1] = 2
2: 00000003;    -- array[2] = 3
3: 00000004;    -- array[3] = 4
4: 00000005;    -- array[4] = 5
5: 00000006;    -- array[5] = 6
6: 00000007;    -- array[6] = 7
7: 00000008;    -- array[7] = 8
8: 00000009;    -- array[8] = 9
9: 0000000A;    -- array[9] = 10
10: 0000000B;   -- array[10] = 11
11: 0000000C;   -- array[11] = 12
End;
```

Ihr Dateiname entspricht dem Bezeichner der für das Array verwendet würde, in diesem Fall also *array.mif*.

2.5 Beteiligte Algorithmen und Konzepte

Der gesamte HLS-Transformationsprozess lässt sich grob in drei Teile gliedern: Aufbauen eines Datenflussgraphen, Ressourcenallokation und Scheduling (vgl. Kapitel 4.2 in [10]). Diese Begriffe werden in den folgenden Abschnitten kurz geklärt.

2.5.1 Bau von Datenflussgraphen

Ein Datenflussgraph gibt die Abhängigkeit der jeweiligen Berechnungen untereinander innerhalb eines sequenziellen Programms wieder (vgl. Kapitel 4.2.1 in [10]). Als Beispiel soll der folgende Code dienen:

```
int summe(int a, int b, int c, int d){
    int temp1, temp2;
    temp1 = a + b;
```

```
    temp2 = temp1 + c;
    return temp2 + d;
}
```

Ein möglicher Datenflussgraph wäre z.B.:

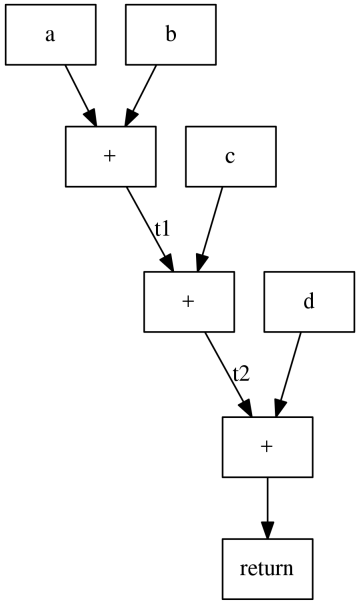

Abbildung 5: Unoptimierter Datenflussgraph

Ein großer Teil der HLS-Forschung bezieht sich auf die Transformation bzw. Vereinfachung von Datenflußgraphen. Ein wichtiger Algorithmus hier: Tree Height Reduction. Dieses Verfahren wird eingesetzt, um die Anzahl der parallel ausgeführten Operationen zu maximieren und somit die Höhe des Baumes zu reduzieren.

Beim oben dargestellten DFG könnte eine Höhenreduktion bspw. folgendermaßen aussehen:

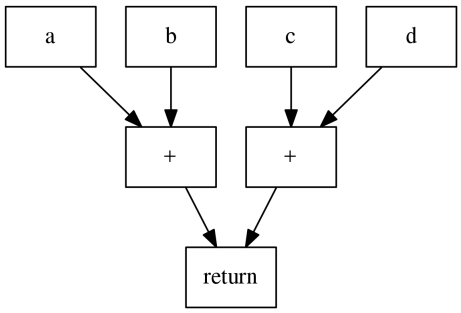

Abbildung 6: Optimierter Datenflussgraph

2.5.2 Allokation

Der Begriff Allokation erweckt für den ersten Moment den Anschein als habe er etwas mit Speichermanagement zu tun. Dies trifft im Kontext der HLS nicht zu. Allokation beschreibt vielmehr die Zuordnung von Operationen (aus der Quellsprache) zu Hardwarefunktionseinheiten (Zielsprache). Hardwarefunktionseinheiten sind z.B. Addierer, Multiplizierer, Dividierer, Multiplexer oder auch FSM's (vgl. Kapitel 4.2.2 in [10]).

Operationen der Quellsprache können, neben einfachen Arithmetischen Ausdrücken, auch Statements (oder allgemeiner: Kontrollstrukturen) sein.

Während des Allokationsprozesses werden sowohl Timing- als auch Platzinformationen[1] verwertet.

Allokationsalgorithmen müssen beim oben angeführten Akkumulatorbeispiel bspw. folgende Fragen beantworten: Soll nur ein Addierer synthetisiert und die Zugriffe darauf gemultiplext oder soll für jede Additionsoperation ein dedizierter Hardwareaddierer spendiert werden? Besitzt die Zielplattform (Ziel-FPGA) einen Pool an arithmetischen Operationseinheiten als ASIC-Block? Wenn ja, welche Timingcharakteristika besitzen diese?

Gute Verfahren müssen sich selbstverständlich ebenfalls mithilfe diverser Parameter vom Nutzer anpassen lassen.

2.5.3 Scheduling

Schedulingalgorithmen weisen den Operationen im DFG Taktzyklen zu. Hier wird zeitlich exakt bestimmt wann welche Aktion (taktsynchon) zu erfolgen hat. Ein wesentilicher Teil der Arbeit besteht auch darin, Register für Zwischenergebnisse an die richtien Stellen in der Pipeline einzubauen. Die meisten Algorithmen befinden sich somit auf der RTL-Ebene (vgl. Kapitel 4.2.3 [10]).

3 Anwendungsbeispiel: Pocoblaze

In der Vorlesung *Rechnerarchitektur 2* stellte Prof. Tempelmeier den Xilinx Softcore *Picoblaze* vor. Dieser Prozessor besitzt eine konstante Instruktionslänge von 16 Bit und eine Datenbreite von 8 Bit. Der Funktionsumfang ist äußerst begrenzt, so gibt es nur drei Addressierungsarten — Immediate, Register und Registerindirekt — und lediglich 18 Instruktionen (vgl. Appendix C [3]):

- `ADD`, außerdem ADDCY, die zusätzlich das Carry zuaddiert
- `AND`
- `CALL`, außerdem CALL [C | NC | Z | NZ][2]
- `COMPARE`
- `DINT`, Disable Interrupt
- `EINT`, Enable Interrupt
- `FETCH`

[1] Bezieht sich auf Hardwaremetriken wie z.B. LUT's, verfügbare Register, ...
[2] C $\mapsto$ if Carry = 1, NC $\mapsto$if Carry = 0, Z $\mapsto$ if Zero = 1, NZ $\mapsto$ if Zero = 0

- `STORE`, Fetch und Store werden für das RAM-Scratchpad verwendet
- `INPUT`
- `OUTPUT`, Für I/O auf 8 externe pins
- `JUMP`, außerdem JUMB [C | NC | Z | NZ]
- `LOAD`
- `OR`
- `RETURN`, außerdem RETURN [C | NC | Z | NZ]
- `SHIFT & ROTATE`, RL, RR, SL [0|1|A|X], SR [0|1|A|X]
- `SUB`
- `TEST`, Funktion definiert durch: $Zero \leftarrow \{sX \wedge sY\}, Carry \leftarrow odd\ parity\ of\ \{sX \wedge sY\}$
- `XOR`

Operationen wie `ADD` oder `TEST` können in zwei Varianten verwendet werden:

```
ADD sX, ff
```

und

```
ADD sX, sY
```

Weiterhin verfügt der Picoblaze über einen Instruktionsspeicher von maximal 256 Befehlen, einer max. Stackgröße von 8 und 64 Speicherplätze im RAM. Trotz dieser Limitierungen ist er ein vollständiger Prozessorkern, der insbesondere in eingebetteten Systemen seine Stärke zeigt. Auf einem modernen Spartan-6 FPGA von Xilinx nimmt der Softcore lediglich 3–5% der verschiedenen Ressroucen (LUT's, Register, Block-RAM, ...) ein. In der Praxis wird er hauptsächlich bei Kontrollanwendungen benötigt, bei deren Entwicklung häufig das Verhalten der im Kontroller steckenden State-Machine flexibel angepasst werden muss. Eine in Assembler geschriebene FSM lässt sich wesentlich einfacher ändern oder warten als eine äquivalente RTL-Repräsentation des Automaten. Außerdem lässt sich die Logik, die das Verhalten der FSM regelt, völlig getrennt vom Rest der Schaltung entwickeln. Dadurch muss das System auch nicht mehr komplett synthetisiert werden, falls nur die Statemachine-Logik verändert wurde.

Im Rahmen dieser Arbeit wurde ein in C geschriebener (binärkompatibler) Emulator entwickelt, der sich automatisch, mithilfe von LegUp, in ein enstprechendes Hardware-Desingfile (Verilog) übertragen lässt. Diese Reinraumimplementierung heißt *Pocoblaze*.

3.1 Implementationsdetails

Die Implementierung des Emulators *Pocoblaze* befindet sich in den Dateien pocoblaze.c sowie pocoblaze.h, wobei sich in der Headerdatei lediglich Funktionsprototypen befinden.

3.1.1 Repräsentation von Instruktionsrom, Registersatz, Ramfile und Stack

Das globale `uint16_t`-Array `instruction_rom` beinhaltet die auszuführenden Befehle. Das selbe Verfahren wird ebenfalls für den Registersatz, den RAM und den Stack verwendet. Wichtig ist hierbei, dass die Arrays dafür den Datentypen `uint8_t` tragen.

```
// Setup of the processors register comonents
uint16_t instruction_rom[INS_ROM] = {
    0xD005, 0x00BE, 0x01EF, 0x0202, 0x9000,
    0xD801, 0xF800, 0xF901, 0xE040, 0x2201,
    // noch mehr Instruktionen [...]
}
```

3.1.2 Umsetzung der Instruktionen

Jede Instruktion besitzt im Pocoblaze ihre eigene Funktion. Dies ist eine Designentscheidung, da jede Funktion zu einer Verilog-Moduldefinition synthetisiert. Genau das ist erwünscht!

Die Umsetzung des `store sx, sy` sieht bspw. so aus:

```
void store_sX_at_sY(){
#ifdef DEBUG
    printf("store\n");
#endif

    ram_file[register_file[y_register_pointer]]
        = register_file[x_register_pointer];
}
```

3.1.3 Fetch, Decode und Execute

In der main-Funktion ist die Logik für's Befehlssfetching untergebracht. Die Implementation ist kaum einfacher möglich:

```
while (program_counter < INS_ROM &&
       instruction_rom[program_counter] != 0x0000) {
    instruction = instruction_rom[program_counter];

    dispatch_instruction(instruction);
}
```

Die Funktion `dispatch_instruction()` setzt sowohl die Dekodierung als auch die Ausführung um. Legup unterstützt die Synthese von Funktionspointern leider nicht, weshalb die Funktion Befehle mithilfe einer großen Switch-Case-Klausel dekodiert. Konkret sieht dies wie folgt aus:

Listing 7: Instruction decoder

```
void dispatch_instruction(const uint16_t instruction) {
    // Dekodierung der Registeradressen und der
    // unmittelbaren Konstante
    x_register_pointer = (instruction & SX_MASK) >> 8;
    y_register_pointer = (instruction & SY_MASK) >> 5;
    constant_argument = instruction & 0x00FF;

    // Hier Definition mehrerer nur lokal
    // verwendeter Bitmasken ausgelassen
```

```
    switch (instruction & INS_MASK) {
        // [ ... ] weitere Fallunterscheidungen
        case 0x0000: load_k_to_x(); break;
        case 0x4000: load_y_to_x(); break;
        case 0x8000: and_k_to_x(); break;
        // [ ... ] weitere Fallunterscheidungen
    }
```

3.2 Known Limitations

Pocoblaze unterstützt zum jetzigen Zeitpunkt noch kein Interrupthandling. Ebenso werden die Befehle `input` sowie `output` ignoriert.

3.3 Fähigkeiten von Pocoblaze

Schon jetzt können komplexere Assemblerprogramme ausgeführt werden. Im Picoblaze Manual (vgl. Seite 31 [3]) ist ein Algorithmus für die korrekte Multiplikation von zwei 8-Bit Registern aufgeführt. Dieser ist jedoch Fehlerhaft! Betrachtet man nämlich die beiden Prozeduren:

```
mult_loop:
        TEST multiplier, bit_mask ; check if bit is set
        JUMP Z, no_add ; if bit is not set, skip addition
        ADD result_msb, multiplicand ; addition only
            occurs in MSB

no_add: SRA result_msb ; shift MSB right, CARRY into bit
    7,
```

so fällt auf, dass im Falle eines Jumps nach `no_add`, das von `TEST` (vgl. Seite 116 in [3]) gesetzte Carryflag nicht gelöscht wird. Da dies recht häufig vorkommt und `SRA` (vgl. Seite 111 in [3]) das Carrybit unmittelbar nach der Shiftopreation an die msb Position schreibt, produziert der Algorithmus Unfug. Abhilfe schafft folgende Modifikation:

```
no_add: AND s0, s0              ; clear CARRY
        SRA result_msb          ; shift MSB right, CARRY
            into bit 7,
```

Mit der oben genannten Änderung multipliziert *Pocoblaze* korrekt! Ebenso implementiert er die verschiedenen `FETCH` und `STORE` Befehle korrekt. Das Programm

Listing 8: Store und Fetch Test

```
        ;; test for fetch and store
        ;;,,,,,,,,,,,,,,,,,,,,,,,,
NAMEREG s2, pointer

        jump main

preload:
        load s0, be
```

```
        load s1, ef
        load pointer, 02
        return

main:   call preload
        store s0, 00
        store s1, 01
        store s0, pointer
        add pointer, 01
        store s0, pointer
        add pointer, 01
        store s0, pointer

        fetch s7, pointer
        sub pointer, 3
        fetch s6, pointer
        fetch s5, 00
```

wird z.B. einwandfrei ausgeführt.

Es muss angemerkt werden, dass sowohl die Emulation, als auch die Synthese und darauf folgende Simulation von Pocoblaze (incl. der jeweiligen Programme im Instructionsspeicher) bisher korrekte Ergebnisse geliefert hat!

3.4 Assembler

Der Assembler stammt aus der Vorlesung *Rechnerarchitektur 2*. Hinzugefügt wurden 5 neue Instruktionen, die der Assembler vorher noch nicht beherrscht hat. Außerdem wurde das Ausgabeverhalten modifiziert. Der Assembler schrieb in der früheren Version immer 4 Dateien in das gegenwärtige Arbeitsverzeichnis: Jeweils eine Hex-, VHDL-, Log- und Bindatei.

Die neue Version erwartet hingegen einen zweiten Parameter auf der Kommandozeile, der die Erzeugung verschiedener Formate steuert. Des Weiteren schreibt der Assemblierer nicht mehr Standardmäßig in von ihm angelegte Dateien, sondern ausschließlich auf die Standardausgabe. Mittels Redirektion (auf der Shell), kann die Ausgabe trivial in eine Datei geleitet werden.

Hier die Parameter:

- `c` produziert den Initialisierungscode für das `uint16_t`-Instruktionsarray
- `b` erzeugt eine Liste von 256, 16-Bit breiten, Instruktionen
- `h` ist analog zu `b` mit dem Unterschied, dass der Assembler die Instruktionen zur Basis 16 ausgibt

4 Glossar

- *DFG*: Datenflussgraph
- *HSL*: High Level Synthesis
- *RTL*: Register-Transfer-Ebene
- *EDA*: Electronic Design Automation
- *BL*: Behavioral Level
- *HDL*: Hardware Design Language
- *Zelle*: Funktionseinheit an einem Knoten in der Netzliste
- *Netzliste*: Zyklischer Graph, der die Verschaltung und Struktur einer Digitalen Schaltung wiedergibt
- *ADT*: Abstrakter Datentyp, die Definition einer Datenstruktur mit entsprechend darauf definierten Operationen
- *AST*: Abstrakter Syntaxbaum
- *Softcore*: Ein in einer HDL geschriebener Prozessor
- *IP*: Intellectual Property Core, ein lizensiertes Hardware-Designfile
- *LLVM IR*: LLVM Intermediate Representation. Eine (bit) typisierte Zwischensprache, die sich auf etwa der selben Abstraktionsebene befindet wie z.B. x86 Assembly.

Abbildungsverzeichnis

Listings

Literatur

[1] Clifford Wold, *Yosys Manual*, 2015

[2] *LegUp Documentation*, University of Toronto, Release 4.0, October 17, 2015

[3] Xilinx, *PicoBlaze 8-bit Embedded Microcontroller User Guide*, Xilinx, v1.1, June 10, 2004

[4] IEEE, *IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis*, IEEE Std 1076.6-2004, October 11, 2004

[5] Brian Bailey, Grant Martin and Andrew Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology*

[6] SystemC, *http://www.systemc.org/community/systemc/about-systemc*

[7] C++-Keywords, *http://en.cppreference.com/w/cpp/keyword*

[8] Brian W. Kernighan und Dennis M. Ritchie, *The C Programming Language*, Prentice Hall, 2nd edition, 1988.

[9] SystemC, *https://en.wikipedia.org/wiki/SystemC*

[10] *High-Level Synthesis – Blue Book*, Mike Fingeroff, Januar 2010 Mentor Graphics Corporation,

[11] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason Anderson, Stephen Brown und and Tomasz Czajkowski, *LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems*, University of Toronto