LLVM-Based C Compiler for the PicoBlaze Processor

Technical Report

Jaroslav Sýkora

Institute of Information Theory and Automation of the ASCR
Pod Vodarenskou vezi 4, CZ-182 08, Prague 8

Abstract. The implementation of the optimizing C compiler for the Xilinx PicoBlaze architecture is presented. The compiler is implemented in the open-source LLVM 2.9 framework. The description focuses on the issues specific to the PicoBlaze code generation process.
The compiler will be made available free of charge for evaluation and non-commercial use at UTIA's web pages:
http://sp.utia.cz/index.php?ids=pblaze-cc.

1 Introduction

One may argue that the efficiency and speed of a compiler for a processor architecture is as critical to the success of the product as the processor microarchitecture and its implementation itself. For from the point of view of a naïve hardware designer it is not that difficult to design a high-performance architecture, yet it may be impossible to compile programs for it. For example the FPGA[1] chip can be viewed as a massively parallel processor, yet we have not seen an efficient compiler of the common C language for it. On the other hand, when software engineers design a processor architecture and a compiler for it, they may simplify the architecture too much for the sake of an efficient compilation process, and unknowingly sacrifice the performance of a future hardware implementation. This is one of the reasons why Google re-defined the Java Virtual Machine, originally a stack-based interpreted processor architecture, to use not-so-compiler-friendly register architecture to increase speed and lower memory footprint [1].

1.1 PicoBlaze Processor Architecture

PicoBlaze [6] is a simple 8 bit processor that can be instantiated in Xilinx FPGA chips. Its intended role is to perform simple control functions in places where a hard-coded state automaton would be too complicated and inflexible, yet a full-fledged RISC processor (such the 32 bit MicroBlaze) too costly.

The architecture diagram is in Figure 1. The PicoBlaze ISA is orthogonal, having 16 general-purpose registers (GPR), each 8 bits wide. The registers are

[1] FPGA = Field-Programmable Gate Array

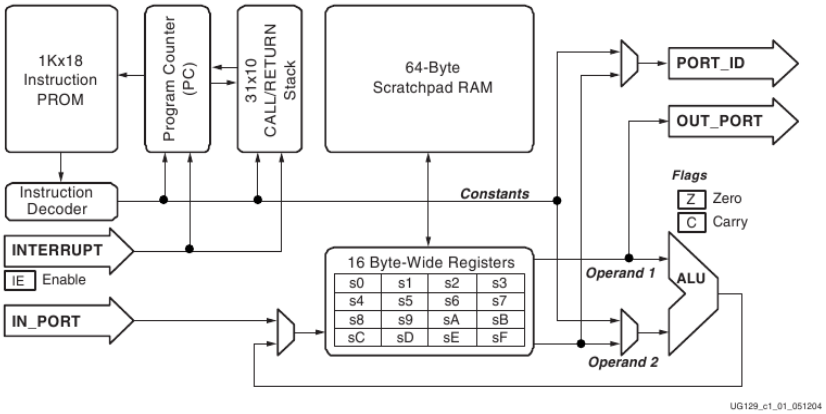


Fig. 1: The architecture of the PicoBlaze CPU. Picture is from [6].

named `%s0` – `%s9`, and `%sA` – `%sF`. All the registers are completely symmetric, and none has any special meaning in the hardware. The processor ALU has a 2 bit flag register with the `CARRY` and `ZERO` flags.

All instructions execute in two clock cycles, without any overlap (there is no pipelining). The instructions are two-address, meaning that in binary operations one of the input register is also the output register. Program address space and data address space are separated (Harvard architecture). The ISA is Load/Store because ALU operations operate exclusively on the GP registers.

1.2 Previous PicoBlaze Compiler Attempts

For the purpose of this report two specific previous C compilers for the PicoBlaze architecture will be discussed as they are in use in the author's research department. The first one is the famous 'pccomp' compiler from Francesco Poderico [4]. The main 'disadvantage' of the compiler is that it was found practically unusable due to a high number of bugs. Even simple test programs are sometimes miscompiled [7].

The second work is the bachelor thesis by B. Nováček [7]. He implemented a stack-based syntax-driven C compiler from scratch. Although there were some bugs, the compiler is quite operational. The disadvantage of the compiler is its relatively high instruction overhead due to the primitive stack-based code generator.

2 The LLVM Compiler Framework

The LLVM[2] [2] compiler framework is an open source collection of libraries and tools written in C++ for building and experimenting with compilers. Other sim-

[2] LLVM = Low-Level Virtual Machine, though the acronym no longer holds any proper meaning.

ilar frameworks include for example the Stanford SUIF system, the commercial CoSy compiler technology, and – of course – the GNU GCC toolchain.

Figure 2 shows the flow of data in the compiler system. The framework is built around a well-defined intermediate representation (LLVM IR [3]). The frontend tool ('Clang' in the case of C/C++ languages) parses the source code language (C/C++) into an internal AST (Abstract Syntax Tree) representation, which may be language-specific, and then lowers it into the language- and target-independent LLVM IR. There are three possible physical formats of the IR:

1. an in-memory representation using C++ objects;
2. a human-readable text file (with the .ll file extension);
3. and a binary 'bitcode' file (.bc) that should load faster than the textual .ll file.

The three formats are logically equivalent, and they can be transformed from one to another without information loss. Target-independent optimizations are performed in the LLVM IR, often by invoking the OPT tool (though usually the optimizations and code generation are performed within one program to speed up the compilation process).

Target code generation is performed in the LLC tool. The target ISA[3] description comprises custom C++ code and 'tablegen' declarations (described below). Some target description is also required in the CLANG frontend; however, this is quite insignificant (on the order of 50 source code lines in C++), and generally the frontend only needs to know the bit sizes of various data types of the target architecture (i.e. how the various **int** types are wide in the ISA).

3 The Target-Independent LLVM IR

The LLVM IR [3] is target- and language-independent representation because its formal structure and the range of opcodes (nodes) is pre-defined in the framework. However, a C/C++ program compiled from its source code into the IR is *not* portable to a different architecture, because the CLANG language frontend has already made some decisions based on the intended target architecture (for example the layout of data in memory, the bit widths of data).

The textual format of the IR looks like an assembler for a virtual RISC-like processor. The elementary instructions of the virtual processor are three-address (two input registers, one output register, but there are some exceptions). They operate on an infinite set of typed virtual registers; the registers are created as needed, they can be given a name, and each register is assigned exactly once so that the IR is implicitly in the SSA form.

Examples: Multiply a 32 b value (IR type 'i32') in the virtual register %X by 8, the result goes into a new virtual register which is named '%result':

```
%result = mul i32 %X, 8
```

[3] ISA = Instruction Set Architecture

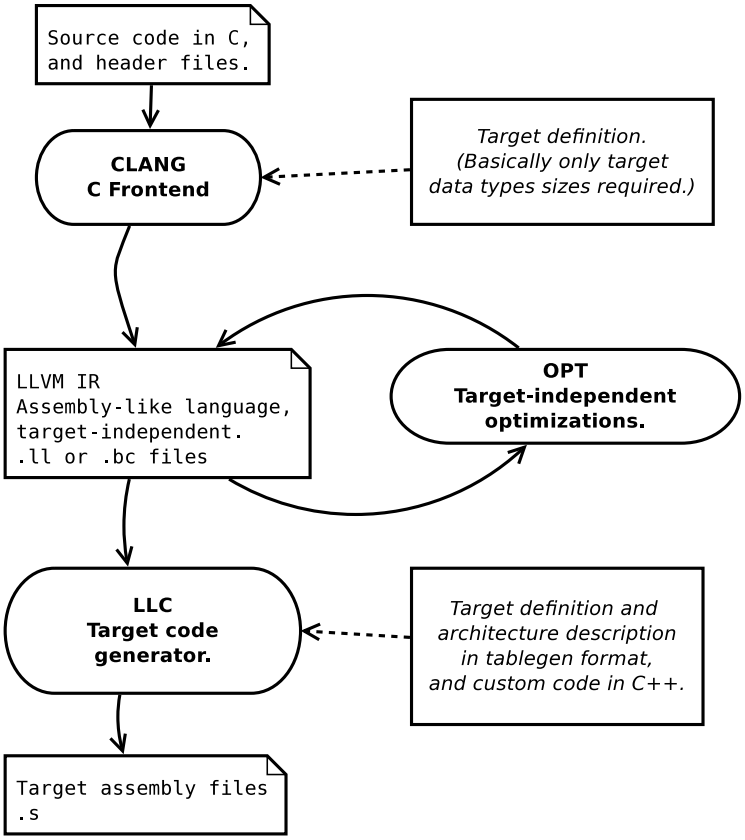


Fig. 2: General compiling flow in the LLVM framework.

After strength reduction (multiplication replaced by left shift by 3; note the 'i8' (integer 8 bit) type of the constant):

```
%result = shl i32 %X, i8 3
```

And the hard way:

```
%0 = add i32 %X, %X           ; yields {i32}:%0
%1 = add i32 %0, %0           ; yields {i32}:%1
%result = add i32 %1, %1
```

The last example illustrates the use of comments in the code–they start with the ';' character and run till the end of the line. When a new temporary virtual register is needed, it is simply named by assigning a number in sequence (%0, %1, ...). Virtual register names begin with the % character. Global symbol names (functions, global variables) begin with the @ character, e.g. '@foo'.

The IR typing system is simple, yet effective. Elementary integer types of any bit widths can be expressed using the 'iN' syntax, for example i1, i8, i32, i80, and so on. (The language frontend lowers the platform-independent C type **int** into the target-dependent i32 (most 32 bit processors), or i16 type (e.g. the PicoBlaze ISA).) Aggregate types are *arrays* (e.g. '[40 x i32]' = array of 40 32-bit integer values), *vectors* (e.g. '<4 x i32>' = vector of 4 32-bit integer values), and *structures* (e.g. '{ i32, i32, i32 }' = a triple of three i32 values). There are also *function* types, *pointers*, and *opaque* (unresolved) types.

Two aggregate types are deemed identical when their structure is the same. The names of such identical types are just aliases to the same underlying type. The implementation ensures that internally there is exactly one unique object representing the type. In practice this means that checking for type equivalence in the compiler is as cheap as a pointer comparison.

4 The Target-Independent Optimizations

Figure 3 shows a simple source program in C. After partial compilation using CLANG the LLVM IR in the human-readable format (.ll file) can be obtained (Figure 4). In the later figure the compilation was intentionally performed without any target-independent optimizations (i.e. at level -O0). The structure of the *for* loop can be clearly seen in the unoptimized code. Also note the explicit use of stack for the local variable 'i'.

The optimization passes can be performed using the OPT program. The result of level 3 target-independent optimizations (-O3) is in Figure 5. The number of basic blocks was reduced, and the local variable 'i' was moved from stack to virtual registers. As the LLVM IR is kept in the *static single assignment* form (SSA), the Φ node at line 10 is used to assign the current value of the induction variable 'i' to the virtual register %0 based on the incoming control flow (zero if coming from 'bb.nph', or %inc if coming from the 'for.body' basic block).

Fig. 3: A simple loop in C.

```
1  extern void print(unsigned char i);
2
3  int testmain()
4  {
5      unsigned char i = 0;
6      for (; i < 250; ++i) {
7          print(i);
8      }
9      return 42;
10 }
```

Fig. 4: A simple loop from Figure 3 after compilation to the LLVM IR, without any optimizations (-O0).

```
; ModuleID = 'e001.c'
target datalayout = "e-p:8:8:8-i8:8:8-i16:8:8-i32:8:8-f32:32:32-n8"
target triple = "pblaze---"

define i16 @testmain() nounwind {
entry:
  %i = alloca i8, align 1
  store i8 0, i8* %i, align 1
  br label %for.cond

for.cond:
; preds = %for.inc, %entry
  %tmp = load i8* %i, align 1
  %conv = zext i8 %tmp to i16
  %cmp = icmp slt i16 %conv, 250
  br i1 %cmp, label %for.body, label %for.end

for.body:
; preds = %for.cond
  %tmp2 = load i8* %i, align 1
  call void @print(i8 zeroext %tmp2)
  br label %for.inc

for.inc:
; preds = %for.body
  %tmp3 = load i8* %i, align 1
  %inc = add i8 %tmp3, 1
  store i8 %inc, i8* %i, align 1
  br label %for.cond

for.end:
; preds = %for.cond
  ret i16 42
}

declare void @print(i8 zeroext)
```

Fig. 5: A simple loop from Figure 3 after compilation to the LLVM IR, with target-independent optimizations performed (-O3).

```
; ModuleID = 'e001.O0.ll'
target datalayout = "e-p:8:8:8-i8:8:8-i16:8:8-i32:8:8-f32:32:32-n8"
target triple = "pblaze---"

define i16 @testmain() nounwind {
bb.nph:
  br label %for.body

for.body:
; preds = %for.body, %bb.nph
  %0 = phi i8 [ 0, %bb.nph ], [ %inc, %for.body ]
  tail call void @print(i8 zeroext %0) nounwind
  %inc = add i8 %0, 1
  %exitcond = icmp eq i8 %inc, -6
  br i1 %exitcond, label %for.end, label %for.body

for.end:
; preds = %for.body
  ret i16 42
}

declare void @print(i8 zeroext)
```

5 The Code Generation Process

Figure 6 shows the organization of the code generation process. The process begins with the target-independent LLVM IR that is transformed in several steps into the final assembly source code in the target ISA.

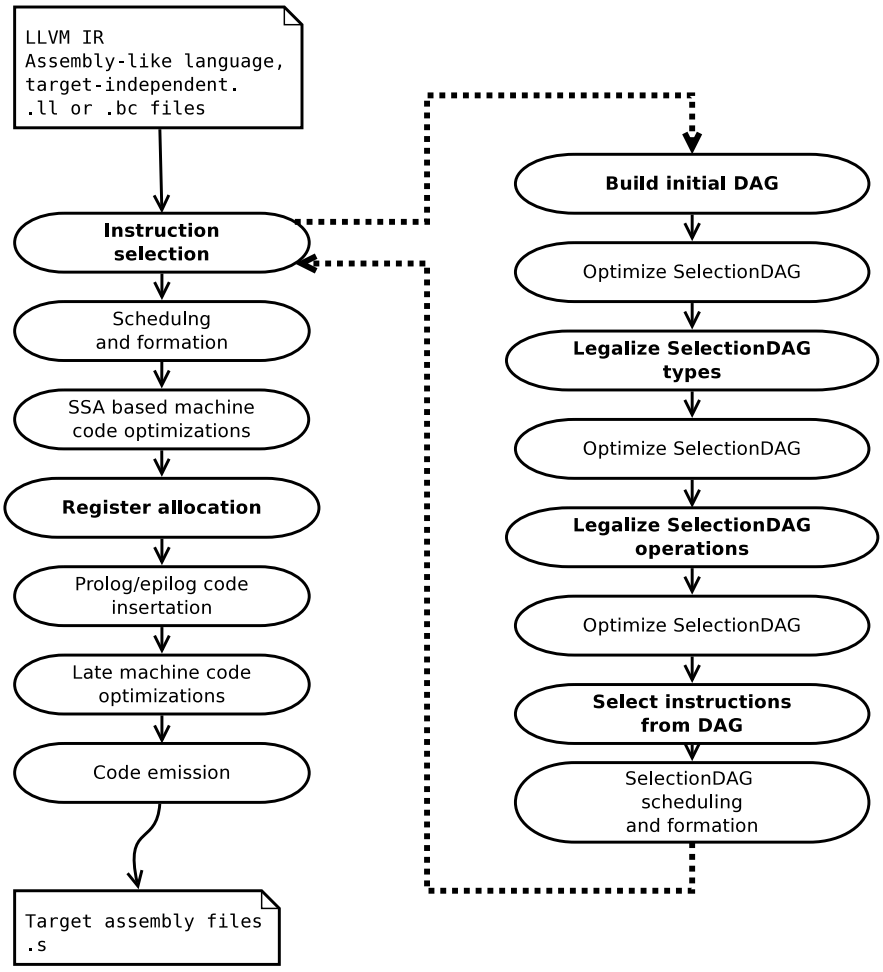


Fig. 6: The LLC code generator.

The first phase of the process – the *instruction selection* phase – is the most involved in regards to the target-dependent information and algorithms that must be supplied to the compiler by a developer; thus this report will focus on the phase. The other phases are much more target-independent, or they require only small portions of target information (e.g. the register allocation phase).

5.1 Building the Initial SelectionDAG

The instruction selection phase of the code generation process is carried out using a so-called *SelectionDAG* representation of the code. SelectionDAG is a control-data-flow directed acyclic graph of operation nodes. The input LLVM IR is converted into the representation at the beginning. For each basic block of the LLVM IR a stand-alone SelectionDAG is created. The nodes of the graph (operations) initially correspond to the instructions in the input LLVM IR, though the mapping is not 1:1. During the course of the code generation process the abstract target-independent operations in the graph are gradually morphed into target-dependent CPU instructions. Ideally, the instruction selection process would be a simple pattern matching task, however, this is regrettably not always the case.

Figure 7 shows the initial SelectionDAG of the *for.body* basic block of the optimized LLVM IR from Figure 5. The SelectionDAG pictures can be automatically generated by the LLC tool as a debugging aid for compiler developers. The arrows in the graph represent data and control dependencies, thus they go against the normal flow of data and control. Pure data dependencies are shown using full black lines. As some operations can have side effects that must be taken into the account during scheduling (function calls, memory loads and stores), the particular operation nodes must be connected by dependency *chains* to explicitly state their relative ordering. A special *TokenFactor* node can be used to split a chain into parallel branches. The chains are shown using blue dashed lines. Some instructions can also have implicit dependencies through the machine flags (e.g. compare+branch). These are modeled using the special *flag* dependency (more appropriately called *glue* in the newer version of the LLVM framework), which is shown in full red line in the picture.

SelectionDAG operation nodes can have multiple outputs (e.g. an ADD instruction may define a register and a flag). This is in contrast to the LLVM IR in which each instruction has at most one result. The initial SelectionDAG in Figure 7 uses mostly target-independent operations (`add`, `xor`, `brcond`, `setcc`), although some nodes are already from the target opcode space (PBISD::CALL, but this is *not* a target instruction yet!). We will use the notation where target-independent codes are written in lower-case letters (e.g. `add`), while target-specific codes and CPU instruction will be written using (mostly) the upper-case characters (e.g. `ADDNCkk`, `COMPAREkk`, `JUMP_cond`).

The example graph in Figure 7 can be logically separated into three parts that loosely correspond to the lines 11, 12, and 13–14 in Figure 5. The subroutine call at line 11 is represented by `callseq_start`, `PBISD::CALL`, and `callseq_end` nodes. (Also note the `CopyToReg` node marked 0x2f6c820 which puts the variable 'i' into the *physical* register `%s0` as per the function calling convention.) The integer addition at line 12 is represented by the `add` node marked 0x2f6cd20. Finally, the conditional branch at line 13–14 is represented by the `setcc`, `seteq`, `xor`, and `brcond` nodes. The `brcond` node is a generic branch conditioned on a boolean value (type 'i1'). This boolean value is generated by the `setcc` node which compares the result of the addition (see operand 0 of the `setcc` node) with the constant value -6 (operand 1), using the equal-to condition (`seteq`,

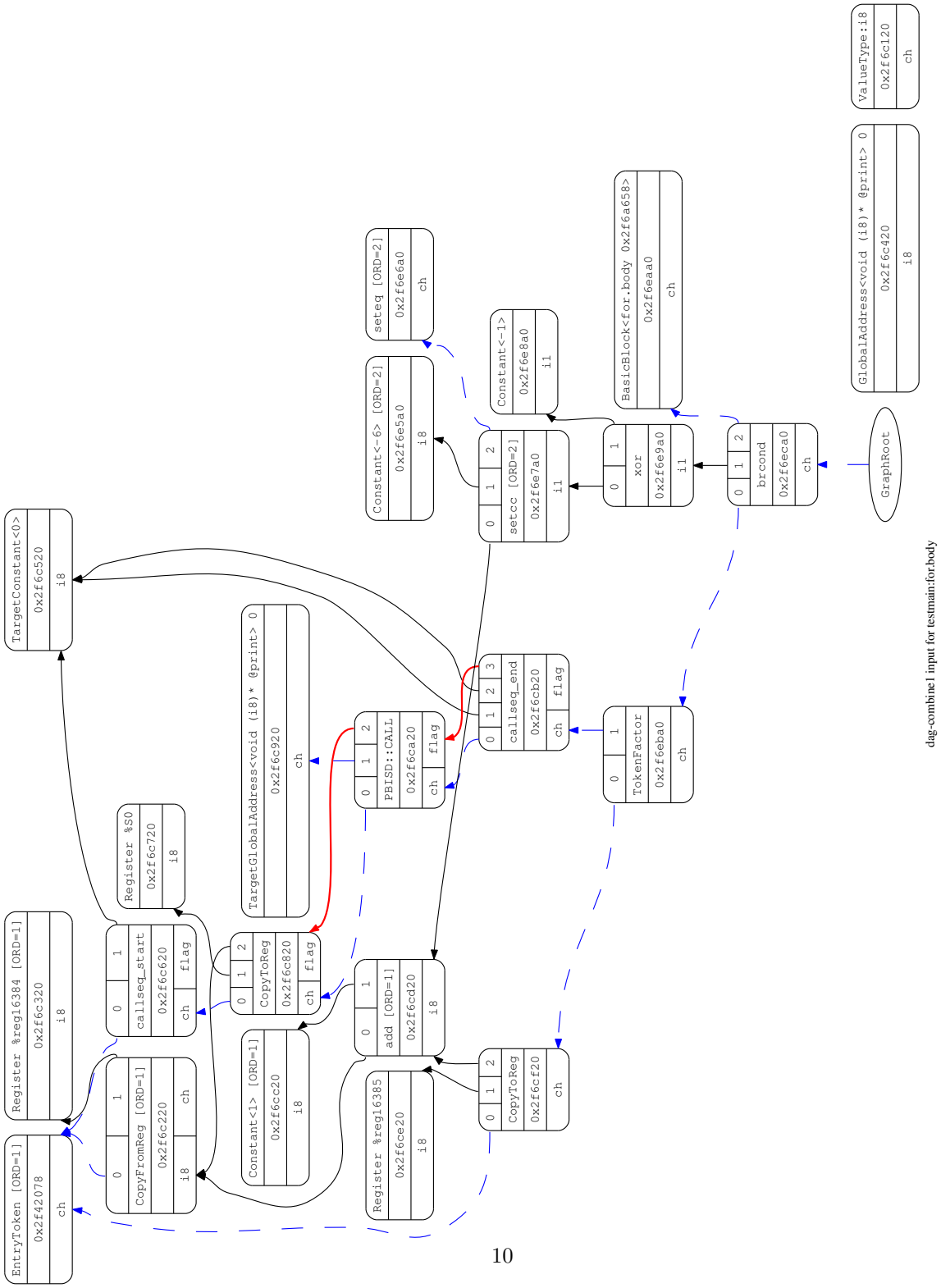


Fig. 7: Initial SelectionDAG of the 'for.body' basic block from Figure 5 (optimized version of the example code).

operand 2). The boolean result is negated using the `xor` node, and then fed into the `brcond` node. An optimization pass will collapse the `xor` node by inverting the condition test in the `setcc` node.

5.2 SelectionDAG Legalization Passes

The initial SelectionDAG can potentially use data types and operations that are not directly available in the target machine architecture. A typical example is 16 bit integer type on an 8 bit ISA architecture. In this case the compiler has to *expand* the 16 bit operation into a sequence of 8 bit ones. For example, a 16 bit addition using the `add` node will be expanded into a subgraph of `addc` and `adde` nodes with a dependency on the *flags*. This compiler pass is called *legalization*, or *lowering*.

Similarly, the compiler can automatically expand some operations that are not supported in the target. Examples include division, shifts, and some kinds of branches. For each combination of the target-independent SelectionDAG operation and data type, it is possible to specify an action that the legalizer should take. The actions are:

- *Legal* – The target natively supports this operation.
- *Promote* – This operation should be executed in a larger type.
- *Expand* – Try to expand this to other operations, otherwise use a libcall.
- *Custom* – Use the LowerOperation() hook to implement custom lowering.

The *custom lowering* action usually replaces the problematic operation node (and possibly some nodes around it) with a target-specific custom node. (Example transformations implemented for the PicoBlaze target will be given later.)

5.3 Instruction Selection Using TableGen

Once a SelectionDAG, which represents one basic block, has been legalized by expanding/promoting illegal data types and operations, possibly by introducing custom target-specific operation nodes, the *instruction selection* phase can take place. Theoretically, the instruction selector matches a DAG pattern representing a CPU instruction to a subgraph in the SelectionDAG. In LLVM the primary way to specify instruction patterns for matching is by *TableGen* [5] tool. TableGen allows a compiler developer to describe the target CPU architecture using a declarative syntax that is easier to maintain than a custom C++ code. Ideally, the whole target architecture would be described in TableGen, requiring no custom C++ code to be written, but the LLVM framework is not there yet.

The target CPU instruction patterns are described in '.td' files. During compiler build the TableGen tool translates the .td files into C++ code which is then included in the build process.

Figure 8 shows a simplified example of a snippet of the '.td' file from the PicoBlaze code generator. In the example two instructions are defined: `ADDkk` and `ADDsy`, representing two formats of the PicoBlaze ADD instruction. The `ADDkk`

variant of the addition has an immediate 8 bit value as the second operand, while the `ADDsy` instruction takes the second operand from a register. The instructions are defined as instances of *InstPB* class. The parameters for the class instantiation are given inside the arrow brackets. The other way of defining parameter values is by the **let** command that allows to override parameters in multiple instances at once. In the example it is used in the *Defs* and *Constraints* parameters at line 1.

Fig. 8: TableGen definitions of two related PicoBlaze instructions ADDkk and ADDsy.

```
1  let Defs = [FLAGS], Constraints = "$src=$dst"
2  in {
3      def ADDkk : InstPB<
4              (outs IntRegs:$dst), (ins IntRegs:$src, i8imm:$k),
5              "ADD  $dst, $k",
6              [(set IntRegs:$dst, (addc IntRegs:$src, simm8:$k))]
7      >;
8
9      def ADDsy : InstPB<
10             (outs IntRegs:$dst), (ins IntRegs:$src, IntRegs:$sy),
11             "ADD  $dst, $sy",
12             [(set IntRegs:$dst, (addc IntRegs:$src, IntRegs:$sy))]
13     >;
14 }
```

The *InstPB* class defines a new CPU instruction. It has four parameters:

- List of outputs: *(outs IntRegs:$dst)* – the output is a single register from the integer registers class *IntRegs*, named $dst here for the substitution purposes.
- List of inputs: *(ins IntRegs:$src, i8imm:$k)* – the first input is an integer register $src, the second input is an 8 bit immediate constant $k.
- Assembly output string: *"ADD $dst, $k"* – in the code emission phase the parameters are substituted and the string is printed into the '.s' assembler file.
- DAG pattern for matching: *[(set IntRegs:$dst, (addc IntRegs:$src, simm8:$k))]* – the pattern is visualized in Figure 9. The pattern matches the target-independent `addc` node ('addition with carry output, no carry in') that has an integer register and a signed 8 bit constant as its arguments.

The *Defs=[FLAGS]* parameter, specified by the **let** command, indicates that the instruction sets ('defines') the `FLAGS` register as its side-effect. The *Constraints* parameter can be used to place additional restrictions on the instruction. In this case it forces the $src and $dst registers to be physically the same because the instruction is two-address. As the LLVM target-independent instructions are three-address, this is an important point. During the instruction selection phase

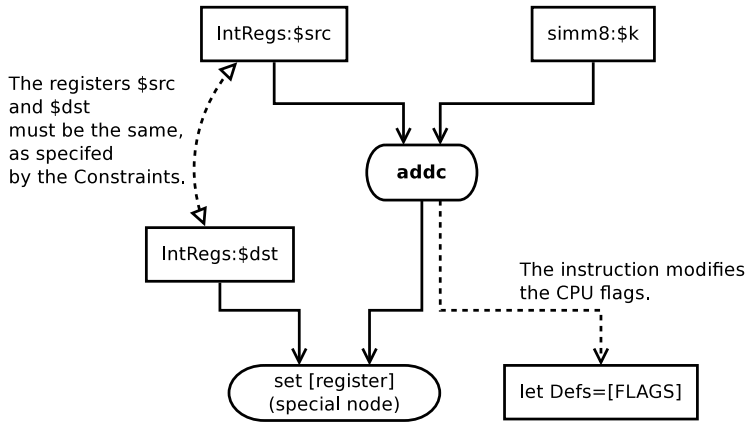


Fig. 9: The DAG pattern of the ADDkk PicoBlaze instruction as modelled in the .td file in Figure 8. (In this picture the arrows represent the actual flow of data.)

the SelectionDAG is in the SSA form and thus the constraint cannot hold yet (each virtual register can be defined only once). The constraint '$src=$dst' is resolved during the *register allocation* phase by allocating the same *physical* register for the two distinct *virtual* registers $src and $dst. However, this maneuver destroys the original content of the $src register. Thus if the original value (in $src) is required in some later computation the allocator has to insert an additional 'move' instruction to copy the value around. (In PicoBlaze ISA the move is the `LOADsy` instruction.)

In Figure 8 two variants of the ADD instruction are defined. The first one is more specific and matches an addition with a constant value. The second one is general as it matches an addition of two registers. If the definition of first variant is removed, the compiler will always use the reg+reg form by first loading the immediate constant value into a temporary register, then performing the addition in registers.

During compiler build the DAG instruction matching patterns are translated by the TableGen tool into a stack FSM automaton. Figure 10 shows the part of the automaton transitions that were generated to recognize the `ADDkk` and `ADDsy` instructions. The FSM transition table is encoded in the C array called *MatcherTable*, with some embedded comments.

The automaton traverses SelectionDAG to match instruction. It has the following internal state (the list is not complete):

- Current node in the SelectionDAG.
- Stack of nodes recorded in the SelectionDAG.
- Current position in the matcher table.
- Stack of previous states of the automaton.

In the beginning the current node pointer is at an unselected target-independent instruction, such as `addc`. Both stacks are empty, and the automaton starts at

Fig. 10: Part of the matcher table, which is automatically generated by TableGen from the instruction definitions in Figure 8.

```
1   static const unsigned char MatcherTable[] = {
2 /*0*/       OPC_SwitchOpcode /*25 cases */, 47|128,1/*175*/,
  TARGET_OPCODE(ISD::LOAD),// ->180
3 ... [many lines removed]
4             /*SwitchOpcode*/ 34,  TARGET_OPCODE(ISD::ADDC),// ->557
5 /*523*/       OPC_RecordChild0, // #0 = $src
6 /*524*/       OPC_RecordChild1, // #1 = $k
7 /*525*/       OPC_Scope, 19, /*->546*/ // 2 children in Scope
8 /*527*/         OPC_MoveChild, 1,
9 /*529*/         OPC_CheckOpcode, TARGET_OPCODE(ISD::Constant),
10/*532*/         OPC_CheckPredicate, 0, // Predicate_simm8
11/*534*/         OPC_MoveParent,
12/*535*/         OPC_EmitConvertToTarget, 1,
13/*537*/         OPC_MorphNodeTo, TARGET_OPCODE(PB::ADDkk), 0|OPFL_FlagOutput,
14                      1/*#VTs*/, MVT::i8, 2/*#Ops*/, 0, 2,
15                  // Src: (addc:i8 IntRegs:i8:$src, (imm:i8)<<P:Predicate_simm8>>:$k)
16                  // Dst: (ADDkk:i8 IntRegs:i8:$src, (imm:i8):$k)
17/*546*/       /*Scope*/ 9, /*->556*/
18/*547*/         OPC_MorphNodeTo, TARGET_OPCODE(PB::ADDsy), 0|OPFL_FlagOutput,
19                      1/*#VTs*/, MVT::i8, 2/*#Ops*/, 0, 1,
20                  // Src: (addc:i8 IntRegs:i8:$src, IntRegs:i8:$sy) - Complexity = 3
21                  // Dst: (ADDsy:i8 IntRegs:i8:$src, IntRegs:i8:$sy)
22/*556*/       0, /*End of Scope*/
23 ...
24            0, // EndSwitchOpcode
25    0
26  };
```

the beginning of the *MatcherTable*. In each step a command is read from the *MatcherTable* at the current position, and executed. If the current node is the `addc` opcode, the following steps will be executed:

1. (line 2 in Figure 10) The very first command `OPC_SwitchOpcode` is a giant case switch conditioned by the SelectionDAG node type (`TARGET_OPCODE`), the first one being `ISD::LOAD`. Eventually the switch will reach `ISD::ADDC` at line 4, and the case is taken.
2. (lines 5, 6) The first two commands `OPC_RecordChild`*n* extract the current node's input operands 0 and 1 (whatever they are), and place them on the node stack.
3. (line 7) Then, a new *scope* is opened, saving the current state of the automaton on a stack for a possible future backtracking. This particular scope contains two blocks (the second block starts at line 17). If any of the following checks fail, the automaton state is restored to the last scope, and different path (block) is taken.

4. (line 8) The current node pointer is moved to the child number 1 – that is the second operand of the `addc` node.
5. (line 9) The opcode of the operand is checked to be an immediate constant (`ISD::Constant`).
6. (line 10) The constant node is checked to be a signed 8 bit immediate value.
7. (line 11) The current node pointer is moved back to the parent node (the `addc` node).
8. (lines 12, 13) The node is converted into the target instruction `PB::ADDkk`.
9. (line 17) If any of the checks above have failed, the automaton backtracks to the last scope, and continues with the next untried block in the scope. In this case the second block starts at line 17.
10. (line 18) The node is converted into the `PB::ADDsy` target instruction.

5.4 Register Allocation and Calling Conventions

Compared to the instruction selection, the *register allocation* phase of the code generation process is relatively simple.

In TableGen definitions the 16 GP registers of the CPU form the *IntRegs* register class. The allocation order is: S4, S5, S6, S7, SC, S0, S1, S2, S3, S8, S9, SA, SB. The registers SD, SE, SF have special meaning in the ABI[4] convention, and thus they are not visible to the allocator. The register convention is summarized in Table 1.

Table 1: PicoBlaze register usage convention (ABI).

Register	**Order**	**Saves**	**Usage**
s0	5	parent	inputs, outputs, locals
s1	6	parent	inputs, outputs, locals
s2	7	parent	inputs, locals
s3	8	parent	inputs, locals
s4	1	parent	locals
s5	2	parent	locals
s6	3	parent	locals
s7	4	parent	locals
s8	9	child	locals
s9	10	child	locals
sA	11	child	locals
sB	12	child	locals
sC	13	child	locals
sD	-	-	reserved for assembly
sE	-	-	frame pointer
sF	-	-	stack pointer

[4] ABI = Application Binary Interface

6 Cross-Cutting Issue: Conditional Branches in the PicoBlaze ISA

The handling of conditional branches in the code generator is relatively complicated. The reason is that processor architectures implement conditions in a wildly different manner. There are several basic ways to implement conditions in an ISA:

- Comparison instruction tests a single given condition and sets a boolean value in a GP register (zero/non-zero). Branch sees if the register is zero. Example: MIPS.
- Comparison instruction tests all possible conditions at once and sets a host of flags (e.g. Zero, Carry, Negative, Overflow). The actual condition is encoded in the branch instruction, which tests for specific combinations of flags. Example: SPARC, x86.
- Comparison instruction tests some conditions (e.g. Zero, Carry). Branch instruction can evaluate only a subset of flag combinations. Example: PicoBlaze.
- Some comparison tests and branches can be fused in one instruction.
- As an alternative, the flow control can be implemented using *predicated execution* of instructions.

6.1 Branches in PicoBlaze ISA

The PicoBlaze ISA has a '`COMPARE A, B`' instruction which subtracts two GP registers (or a register and a constant) and sets the `CARRY` and `ZERO` flags accordingly (no GP register is modified by the instruction). The subtraction is always *unsigned*. The `ZERO` flag is set when `A - B` is zero, otherwise it is cleared. The `CARRY` flag is set when `(unsigned)A < B`, otherwise it is cleared.

The conditional jump instructions can test whether the given flag is set or cleared. Thus there are four types of branches: JUMP Z, JUMP NZ, JUMP C, JUMP NC. The instructions jump when the Zero flag is set, or cleared, or when the Carry flag is set, or cleared, respectively. It is not possible to test a combination of flags in one branch instruction.

Table 2 shows all the general integer condition codes in the compiler (EQ, NE, LT, GT, LE, GE, ULT, ULE, UGT, UGE) and their mapping to sequences of instructions in the PicoBlaze ISA. The unsigned condition tests are quite easy to compile as they can be directly mapped to a combination of the COMPARE+JUMP instructions. However, the signed conditions are tricky because the ISA does not support them directly. But it turns out that all that is needed is an inversion of the most significant bits of both the operands L and R. This is achieved by the `XOR 0x80` instructions.

6.2 Branches in SelectionDAG

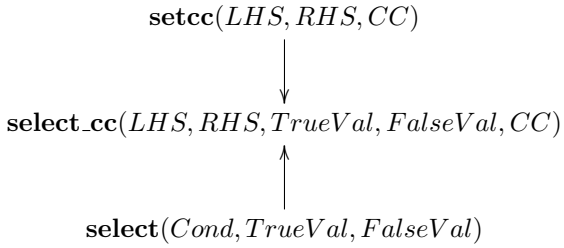
In SelectionDAG target-independent representation there are five types of nodes to support branching: `select`, `select_cc`, `brcond`, `br_cc`, ans `setcc`.

Table 2: The LLVM general condition codes and the corresponding evaluation sequence in PicoBlaze ISA.

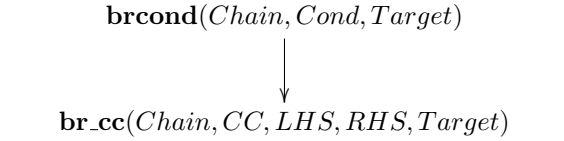
Signess	**Cond.Code**	**Expression**	**Instruction Sequence**
	SETEQ	$L = R$	`COMPARE L, R; JUMP Z, Dest`
	SETNE	$L \neq R$	`COMPARE L, R; JUMP NZ, Dest`
Signed	SETLT	$L < R$	`XOR L, 0x80; XOR R, 0x80; COMPARE L, R; JUMP C, Dest`
	SETGT	$L > R$	`XOR L, 0x80; XOR R, 0x80; COMPARE R, L; JUMP C, Dest`
	SETLE	$L \leq R$	`XOR L, 0x80; XOR R, 0x80; COMPARE R, L; JUMP NC, Dest`
	SETGE	$L \geq R$	`XOR L, 0x80; XOR R, 0x80; COMPARE L, R; JUMP NC, Dest`
Unsigned	SETULT	$L \prec R$	`COMPARE L, R; JUMP C, Dest`
	SETUGT	$L \succ R$	`COMPARE R, L; JUMP C, Dest`
	SETULE	$L \preceq R$	`COMPARE R, L; JUMP NC, Dest`
	SETUGE	$L \succeq R$	`COMPARE L, R; JUMP NC, Dest`

The `setcc` node compares two values (LHS and RHS) according to a given condition code (CC) and outputs a boolean value (i.e. 1 bit integer of type 'i1'). The allowed condition codes are given in Table 2 in the 'Cond.Code' column.

The `select` and `select_cc` nodes are used to implement a C ternary operator – their result is a selection from two input values (TRUEVAL and FALSEVAL) according to a condition (COND). They can be helpful when compiling for an ISA with the predicated execution. The simpler `select` node has as an input a boolean condition which directly drives the selection. Typically the condition value is the output of the `setcc` node. The complex `select_cc` node is a fused `setcc`+`select`, i.e. it has an embedded test of LHS and RHS to a CC.



The `brcond` and `br_cc` nodes are conditional branches. The simpler `brcond` is a conditional branch on a given boolean value, which is typically an output of the `setcc` node. The complex `br_cc` node is fused `setcc`+`brcond`, i.e. it has an embedded test of LHS and RHS to a CC. As the branching operations have side-effects, they have to be linked in the control-flow dependency *chains*.



6.3 Branches Compilation in the PicoBlaze Backend

A compiler backend in LLVM does not have to support all five types of the branching SelectionDAG nodes. The compiler can automatically transform some of the node types into the more powerful ones during the legalization pass. The `setcc` and `select` nodes can be automatically expanded into `select_cc`, and the `brcond` node can be expanded into `br_cc`. The PicoBlaze code generator takes advantage of this offer; thus it has to care of only the `select_cc` and `br_cc` nodes. The handling of the various branching nodes is summarized in Table 3.

Table 3: Handling of the branching SelectionDAG nodes in various compiling phases, as implemented in the PicoBlaze backend.

Original ISD	**Legalization Pass**	**Instruction Selection**	**Instr. Emission**
`setcc`	expand into `select_cc`	(impossible)	(impossible)
`select`	expand into `select_cc`	(impossible)	(impossible)
`select_cc`	custom-expand into `COMPARE` and `SELECT_ICC`	`COMPAREkk`/`COMPAREsy`, `SELECT_ICC_PSEUDO`	Replace the pseudo-op by a diamond CF.
`brcond`	expand into `br_cc`	(impossible)	(impossible)
`br_cc`	custom-expand into `COMPARE` and `JUMPCC`	`COMPAREkk` or `COMPAREsy`, `JUMP_cond`	(no change)

Compilation of the br_cc nodes: The compilation process of the `br_cc` node is simpler, so it will be described first. In the *legalization phase* of the code-generation process, `br_cc` nodes are expanded by custom C++ code to combinations of `COMPARE`, `xor`, and `JUMPCC` nodes using the receipt in Table 2. The `COMPARE` and `JUMPCC` nodes are special SelectionDAG operations (not CPU instructions!) that are specific to the PicoBlaze target. For example, the `COMPARE` operation in the SelectionDAG is more general that the `COMPAREkk` and `COMPAREsy` target instructions, because it does not care if its operands are in registers or immediate constant fields. Similarly, in the `br_cc` legalization of the signed comparison tests the *target-independent* `xor` operation is used. This allows for an optimization pass to perform more constant folding.

Why are the combinations of `setcc`+`brcond` nodes lowered into the fused `br_cc` node, only to be immediately custom-expanded into the `COMPARE`+`JUMPCC` nodes? In the abstract target-independent `setcc`+`brcond` organization the value comparison and condition evaluation is strictly localized in the `setcc` node, while the `brcond` node is only concerned with branching based on the boolean result of the condition evaluation. However, in the PicoBlaze ISA the condition evaluation is distributed between the `COMPARE` and `JUMPCC` operations. The `JUMPCC` operation needs to know which CPU flag it has to test (Z, NZ, C, NC), and this in turn requires the knowledge of the original condition code.

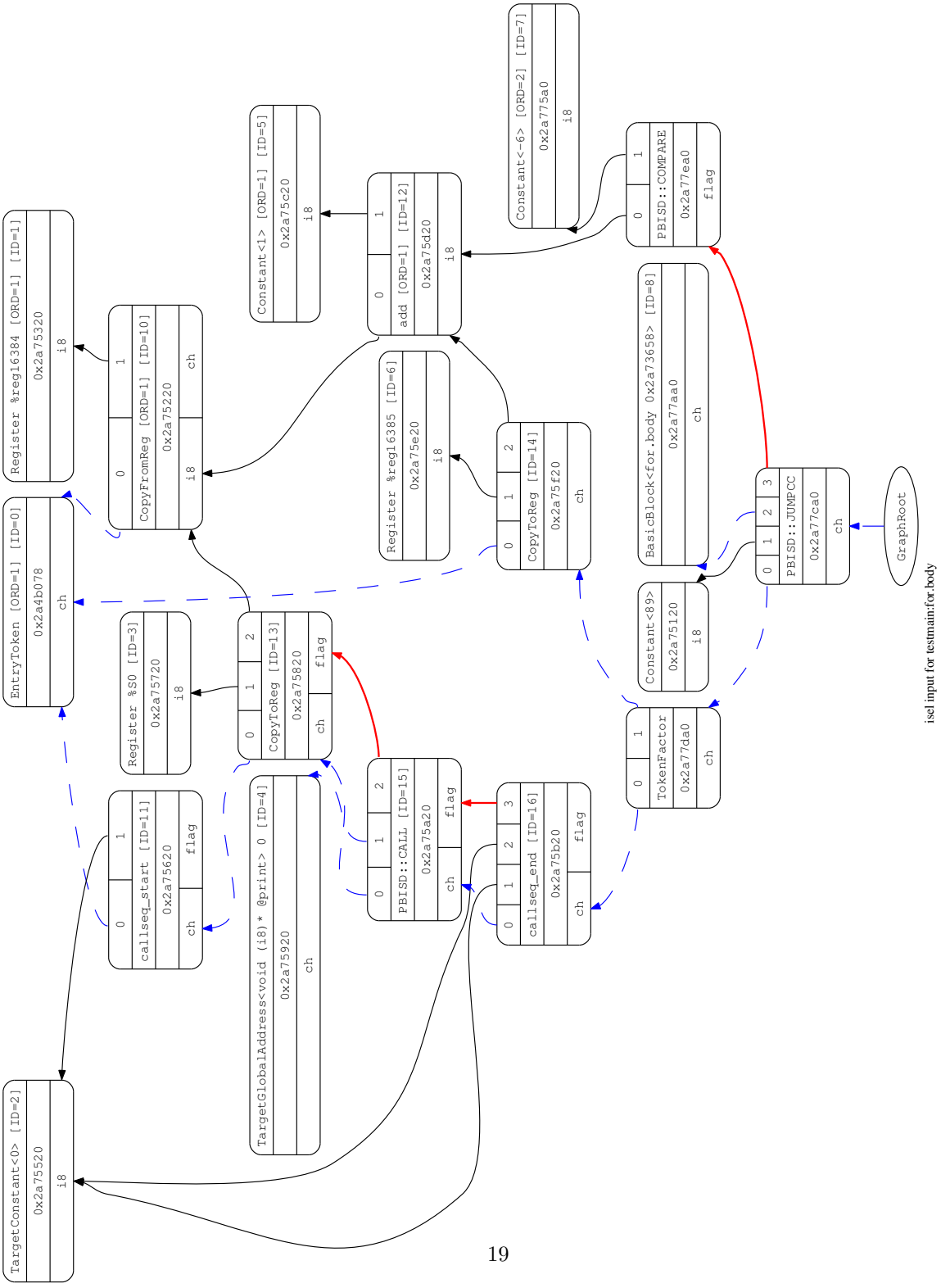
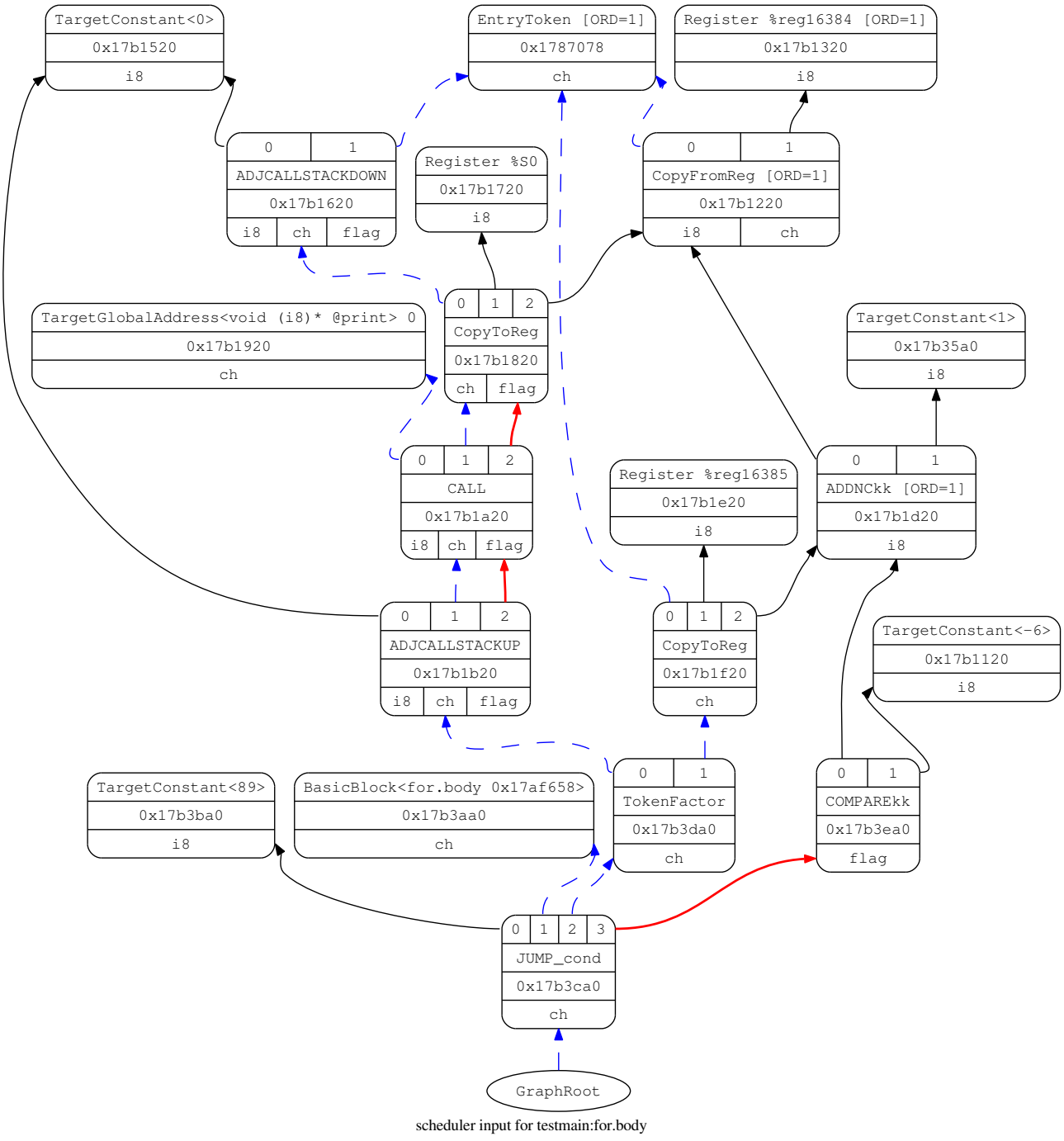


Fig. 11: SelectionDAG after legalization pass. Note the use of PBISD::COMPARE and PBISD::JUMPCC nodes to express the branch.



scheduler input for testmain:for.body

Fig. 12: SelectionDAG after instruction selection. The branch operation nodes were morphed into the COMPAREkk and JUMP_cond target instruction.

Figures 11 and 12 show example SelectionDAGs after the legalization phase, and after the instruction selection phase (respectively). Contrast the branching subgraph near the lower-right corners of the pictures with the initial SelectionDAG in Figure 7. In Figure 7 (initial SelectionDAG) the branching is realized using the `setcc`+`brcond` combination. In Figure 11 (after legalization) the branching subgraph is morphed into the `COMPARE`+`JUMPCC` target-dependent SelectionDAG. Note the operand 1 of the `JUMPCC` node ('Constant 89') which encodes the condition code and the CPU flags that the conditional jump shall test. Finally, after instruction selection in Figure 12, the concrete CPU instructions `JUMP_cond` and `COMPAREkk` were selected.

Compilation of the select_cc nodes: The compilation process of the `select_cc` nodes is more intricate. During the legalization phase, `select_cc` nodes are lowered into combinations of `COMPARE` and `SELECT_ICC` custom nodes using a similar process as previously. The function of the `SELECT_ICC` node is to choose between two input values based on the result of the comparison (like the ternary operator in C). As the node requires knowledge of the original condition code, the same trick as previous is employed.

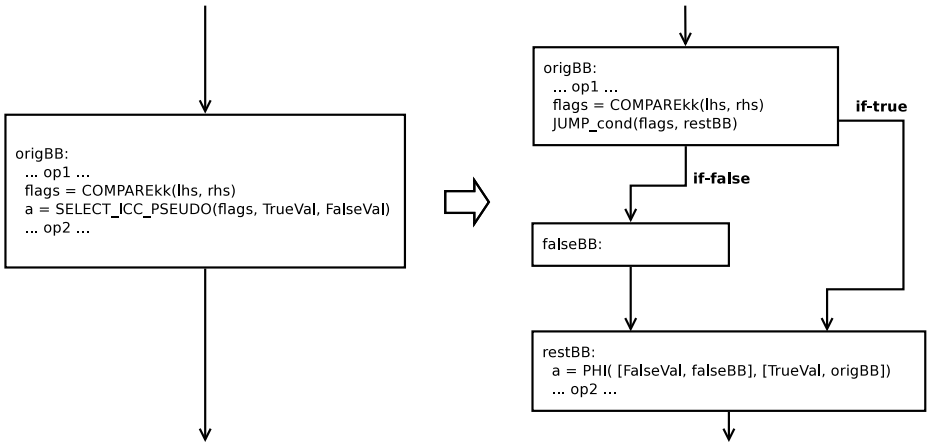


Fig. 13: The transformation of the `SELECT_ICC_PSEUDO` instruction into the diamond control flow pattern.

In PicoBlaze ISA there is no instruction corresponding to the `SELECT_ICC` operation. Instead, the compiler has to emit a diamond control-flow pattern with conditional jumps. To accomplish that, a pseudo CPU instruction called `SELECT_ICC_PSEUDO` is defined in TableGen, and it is mapped in the instruction selection phase directly to the `SELECT_ICC` operation. Then during an instruction emission pass (after instruction selection, but before register allocation) a custom C++ code is run that transforms the pseudo-instruction into a corresponding

diamond-shaped control-flow pattern, as shown in Figure 13. Two new basic blocks are created ('falseBB' and 'restBB'), and the `JUMP_cond` instruction is inserted in the place of the original `SELECT_ICC_PSEUDO` instruction. The original basic block is split at the conditional jump, and all the instruction after it are moved into the 'restBB' basic block. Then a Φ node is inserted at the beginning of the 'restBB' basic block to assign the output virtual register 'a' with a value according to the direction from which the control flow has come:

$$a = \Phi([FalseVal, falseBB], [TrueVal, origBB])$$

(If the control flow has arrived at 'restBB' from the 'falseBB' basic block, the value of 'a' should be 'FalseVal'; if the control flow has arrived from the 'origBB' basic block, the value of 'a' should be 'TrueVal'.) During the register allocation phase the Φ nodes direct the compiler to insert the load instructions at appropriate places to satisfy the constraints.

6.4 Analysis of the Compiled Example Program

Figure 14 shows the final assembly code produced by the compiler from the example code in Figure 3. The comments starting ;> were added manually for the presentation purpose. Note that in PicoBlaze ISA the `LOAD` instruction loads an immediate value into the register, or moves data between registers. To read/write memory the `FETCH/STORE` instructions can be used.

The compiler-generated code is quite efficient. It uses the register %s8 to hold the loop counter 'i', because this register is guaranteed not to be destroyed by a subordinate function. However, the original register content has to be saved at the beginning of the function (line 10–13), and later restored at the end (lines 26–28). Prior to calling the subordinate function 'print' at line 19, the register %s0 has to be loaded with the function's input value–the loop counter (lines 14, 21).

The function prologue and epilogue codes (lines 7–9, 29–31) can be optimized away. Functions with a statically known number of stack-allocated bytes do not need to create their own stack frames. The addressing of the local variables can be done by references through the stack pointer register %sF.

7 Conclusion

This report has described the key steps encountered by the author when implementing a new CPU target in the LLVM framework. We have focused only on the most interesting parts of the code-generation process, entirely skipping the early target-independent phases of the compiler.

Much of the target-specific ISA information can be encoded in the TableGen format, greatly simplifying the development and maintenance of the compiler's source code. Yet many transformations still have to be done 'by hand' in a C++ code. It is striking that such a basic thing as a conditional jump was

relatively complex to implement. On the other hand, register allocation and stack handling (spilling) code worked automatically 'out of the box' without much target-specific support code.

Acknowledgement

This work has been supported from project SMECY, project number Artemis JU 100230 and MSMT 7H10001.

References

1. Dalvik (software). http://en.wikipedia.org/wiki/Dalvik (software)
2. The LLVM Compiler Infrastructure. http://www.llvm.org/
3. LLVM Language Reference Manual. http://www.llvm.org/docs/LangRef.html
4. Picoblaze C Compiler by Francesco Poderico. http://www.asm.ro/fpga/, The web page is non-functional now, use web.archive.org.
5. TableGen Fundamentals. http://llvm.org/docs/TableGenFundamentals.html
6. UG129: PicoBlaze 8-bit Embedded Microcontroller User Guide. Tech. rep., Xilinx
7. Nováček, B.: Překladač jazyka C pro mikrokontroléry PicoBlaze (2008), bechelor thesis, ČVUT FEL, Prague

Fig. 14: Compiled example program. The comments starting ;> were added manually for the presentation purpose.

```
1           .file   "e001.O3.ll"
2           .text
3           .globl  testmain
4           .type   testmain,@function
5   testmain:                                   ; @testmain
6   ; BB#0:                                     ; %bb.nph
7           SUB     %sF, 1              ;> prolog: alloc byte
8           STORE   %sE, (%sF)          ;> prolog: store old frame ptr.
9           LOAD    %sE, %sF            ;> prolog: set new frame ptr.
10          SUB     %sF, 1              ;> Alloc 1B on stack,
11          LOAD    %sD, %sE            ;> set %sD to
12          ADD     %sD, -1             ;> point to it,
13          STORE   %s8, (%sD)          ;> and save old %s8.
14          LOAD    %s0, 0              ;> Initialize counter.
15  .LBB0_1:                                    ; %for.body
16                                              ; =>This Inner Loop Header: Depth=1
17          LOAD    %s8, %s0            ;> %s8 is preserved across calls.
18          ADD     %s8, 1              ;> Increment the counter.
19          CALL    print               ;> (Input is in %s0)
20          COMPARE %s8, -6             ;> Test counter.
21          LOAD    %s0, %s8
22          JUMP    NZ, .LBB0_1
23  ; BB#2:                                     ; %for.end
24          LOAD    %s0, 42             ;> Func. result is in %s0,%s1.
25          LOAD    %s1, 0
26          LOAD    %sD, %sE            ;> Restore original %s8...
27          ADD     %sD, -1
28          FETCH   %s8, (%sD)          ;> from the stack.
29          LOAD    %sF, %sE            ;> epilog: deallocate locals.
30          FETCH   %sE, (%sF)          ;> epilog: restore old frame ptr.
31          ADD     %sF, 1              ;> epilog: deallocate its storage
32          RETURN
33  .Ltmp0:
34          .size   testmain, .Ltmp0-testmain
```