IBM Z back end in GCC vs. LLVM
A comparison

Dr. Ulrich Weigand
Senior Technical Staff Member
GNU/Linux Compilers & Toolchain

Date: Sep 9, 2018

© 2018 IBM Corporation

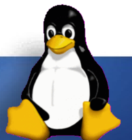
Agenda

- **IBM Z support in GCC vs. LLVM – Status**
  - Port history
  - General features
  - ISA exploitation
  - Microarchitecture tuning
  - Performance results
- **Compiler internals**
  - Back-end pass structure
  - Instruction definition (.md vs. .td files)
  - Platform ABI implementation

© 2018 IBM Corporation

IBM Z Support in LLVM and GCC
Status & Overview

© 2018 IBM Corporation



What is LLVM?

- **LLVM Compiler Infrastructure Project**
  - Collection of modular and reusable compiler and toolchain technologies
  - Provides libraries as well as several stand-alone tools
- **LLVM Core**
  - Based on well-specified code representation (LLVM IR)
  - Source- and target-independent optimizers
  - Code generators for various processor architectures
  - Object file utilities, assembler, disassembler
  - Just-in-time compile infrastructure (incl. module loader)

© 2018 IBM Corporation

What is LLVM? (cont.)

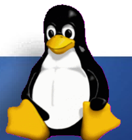
- **clang – C/C++/Objective-C compiler**
  - Fully C++17 standard compliant
  - Modular architecture supporting diverse clients
    - Refactoring tools, static analysis, code generation, …
  - Code instrumentation tools, e.g.
    - AddressSanitizer – detect out-of-bounds accesses
    - ThreadSanitizer – detect data races
    - MemorySanitizer – detect uninitialized reads
- **Additional components under development**
  - flang (Fortran compiler), LLDB (debugger), LLD (linker)
- **Various runtime libraries (C, C++, OpenMP, ...)**

© 2018 IBM Corporation

IBM Z port history

- **GCC**
  - Upstream since 3.0.1 (2001)
  - 18 major releases since
    - 1163 SVN commits
  - Main contributors:
    - Hartmut Penner
    - Andreas Krebbel
    - Robin Dapp
    - Dominik Vogt
    - Wolfgang Gellerich
    - Adrian Strätling
    - Ulrich Weigand

- **LLVM**
  - Upstream since 3.3 (2013)
  - 9 major releases since
    - 848 SVN commits
  - Main contributors:
    - Richard Sandiford
    - Jonas Paulsson
    - Zhan Jun Liau
    - Marcin Kościelnicki
    - Ulrich Weigand

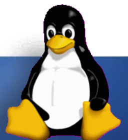


© 2018 IBM Corporation

General IBM Z feature set

- **GCC (& GNU toolchain)**
  - 31-bit & 64-bit
  - Linux, z/TPF
  - z900 … z14
  - C, C++, Objective-C, Fortran, Ada, Go
  - Vector extension
  - Runtime libraries: libstdc++, libgcc
  - Binutils assembler & disassembler
  - Binutils linker (ld, gold)
  - Debugger: GDB

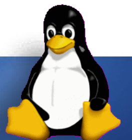
- **LLVM**
  - 64-bit only
  - Linux only
  - z10 … z14
  - Clang (C, C++, Objective-C), Swift, Rust
  - Vector extension
  - Runtime libraries: libc++, compiler-rt (not supp.)
  - Integrated assembler & disassembler
  - Linker: LDB (not supp.)
  - Debugger: LLDB
  - JIT compiler



© 2018 IBM Corporation

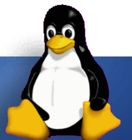
Main use cases

- **GCC**
  - System compiler
    - Linux distributions: Red Hat, SUSE, Debian/Ubuntu
    - z/TPF
  - Most ISV software
- **LLVM**
  - JIT compiler
    - 3D graphics (mesa/llvmpipe)
    - Various ISV software
  - Other languages
    - Swift, Rust
  - C/C++ compiler
    - User preference
    - Compile speed
    - Diagnostics
    - Sanitizers



© 2018 IBM Corporation

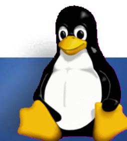
Instruction-Set Architecture Optimizations



© 2018 IBM Corporation

IBM Z instruction-set architecture overview

- **z/Architecture publicly documented by IBM**
  - z/Architecture Principles of Operation (SA22-7832-11)
- **Successor to prior architectures going back to 1960s**
  - System/360
  - System/370
  - System/370 extended architecture (370-XA)
  - Enterprise Systems Architecture/370 (ESA/370)
  - Enterprise Systems Architecture/390 (ESA/390)
- **Updated for each new processor generation**
  - Eighth Edition: z10
  - Ninth Edition: z196
  - Tenth Edition: zEC12
  - Eleventh Edition: z13
  - Twelfth Edition: z14



© 2018 IBM Corporation

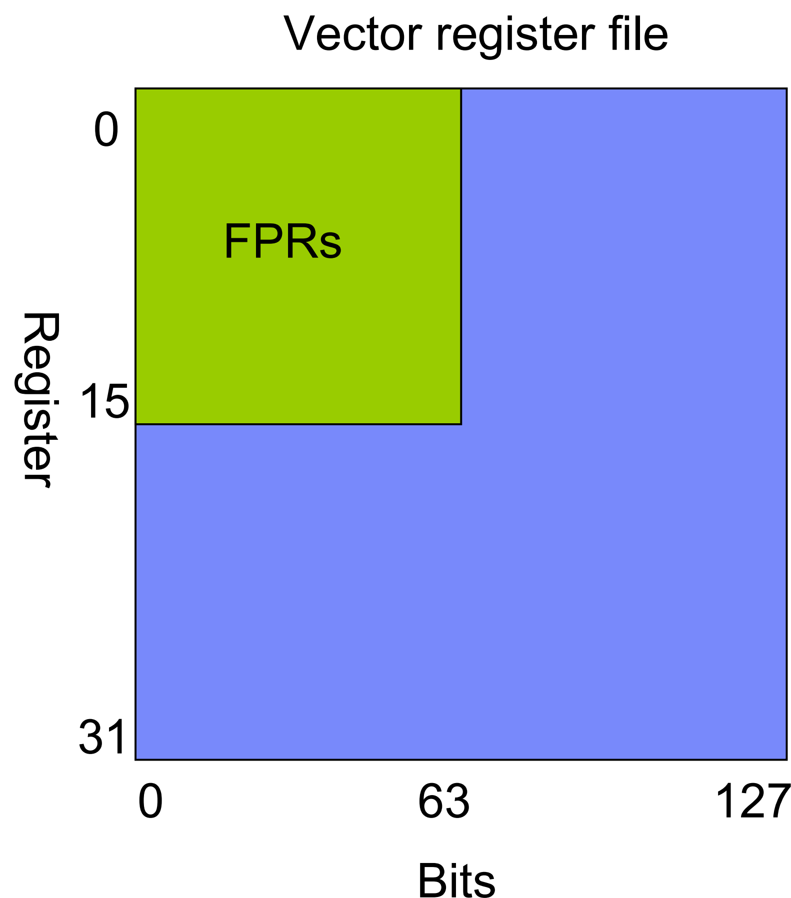
z/Architecture overview

- **Register file**
  - 16 64-bit general-purpose register
  - 16 64-bit floating-point registers
  - 32 128-bit vector registers (overlapping FPRs)
  - 16 32-bit access registers
  - Program Status Word (incl. PC and condition code)
- **Instruction set**
  - >1000 basic instructions, >2000 extended mnemonics
  - CISC operations (reg/reg, reg/mem, mem/mem, …)
  - IEEE floating-point, decimal FP, hexadecimal FP
  - Vector general, integer, floating-point, string instructions

© 2018 IBM Corporation

Overlaid vector / floating point register file

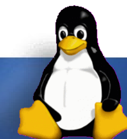
- **Overlaid register file**
  - Bits 0:63 of SIMD registers 0-15 will correspond to FPRs 0-15
  - When writing to an FPR, bits 64:127 of the corresponding vector register will become unpredictable
- **SIMD width 128 bits**
  - 1x128b, 2x64b, 4x32b, 8x16b, 16x8b integer
  - 1x128b, 2x64b, 1x64b, 4x32b, 1x32b floating-point



© 2018 IBM Corporation

z/Architecture: high-word register operations

- **64-bit GPRs treated as two independent 32-bit parts**
  - Intended to provide register relief (32 “registers” for many operations)
- **For example, to add an immediate:**
  - AGFI %r2, 1 → add 1 to full 64-bit register (64-bit ISA)
  - AFI %r2, 1 → add 1 to low 32-bit part (legacy 32-bit ISA)
  - AIH %r2, 1 → add 1 to high 32-bit part (high-word facility)
- **Modeled as sub-registers in LLVM**
  - GR64 → 64-bit GPRs
  - GR32 → 32-bit lower half GPRs
  - GRH32 → 32-bit upper half GPRs
  - GRX32 → union of GR32 and GRH32
    - Used in ISEL for operations supported on both halves
    - Post-RA expander selects final instruction depending on register
    - AFIMux (GRX32) pseudo → AFI or AIH



© 2018 IBM Corporation

z/Architecture: high-word register operations

- **Difficult to implement: instructions with two registers**
  - E.g. COMPARE could be modeled as CMux (GRX32, GRX32)
  - After register allocation, all four combinations possible
  - But ISA only has instructions for three of them:
    - Low/Low → CR
    - High/Low → CHLR
    - High/High → CHHR
  - Special handling for Low/High case required
    - May be convertible to High/Low by updating all users
    - Otherwise 2-3 instruction sequence involving rotates
- **Even more difficult: ADD with three register operands**
  - Only 3 combinations supported in ISA: LLL, HHL, HHH
  - Some cases would require up to 4 instructions to emulate
- **Should ideally be handled in RA directly (like GCC “alternatives”)**
  - But LLVM RA deliberately makes no instruction selection choices ...

© 2018 IBM Corporation

z/Architecture: high-word register operations

- **Currently not implemented in GCC**
  - No fundamental obstacle, but ...
  - Significant rework of register / mode support
  - Possible interference with CC instruction handling
  - Reload alternatives helpful to support all instructions
- **LLVM implementation shows not much benefit**
  - Even regressions in some cases
  - Still working on fixes / enhancements
- **Revisit GCC implementation option in the future**

© 2018 IBM Corporation

z/Architecture: conditional instructions

- **Condition code – two bits in the PSW**
  - Comparable to flags bits, but used as single value
  - Instructions may set any CC value, no fixed semantics
  - Branch instructions may test for any CC combination

Instruction examples	CC 0	CC 1	CC 2	CC 3
COMPARE (integer)	Equal	Low	High	-
COMPARE (floating-point)	Equal	Low	High	Unordered
ADD	Zero	< Zero	> Zero	Overflow
ADD LOGICAL	Zero, no carry	Not zero, no carry	Zero, carry	Not zero, carry
AND	Zero	Not zero	-	-
FIND LEFTMOST ONE	No one bit found	-	One bit found	
TEST UNDER MASK LOW	All zeros	Mixed, left bit zero	Mixed, left bit one	All ones
VECTOR COMPARE EQUAL	All elements equal	Some elts. equal	-	No elements equal
CONVERT UTF-8 TO UTF-32	Data processed	Destination full	Invalid UTF-8	Early exit

© 2018 IBM Corporation

IBM

z/Architecture: more conditional instructions

- **Instructions using the condition code**
  - LOAD ON CONDITION
    - Load from memory/register/immediate if CC in mask
  - (Conditional) trap instruction
    - Special form of (conditional) branch with invalid target
- **Instructions that do not use the CC**
  - COMPARE AND BRANCH / TRAP
    - Compare + conditional branch (or trap) as single insn
  - BRANCH ON COUNT
    - Decrement register and branch if not zero
  - LOAD AND TRAP
    - Load register from memory and trap if zero

© 2018 IBM Corporation

LLVM code generation for conditional instructions

LLVM pass	z/Architecture ISA handling
Instruction selection	Select appropriate compare instructions Generate TEST UNDER MASK Generate LOC/LOCR from selects
Early if-conversion	Generate LOCR from if blocks Speculative execution of both sides
Peephole optimizer	Optimize explicit uses of CC (e.g. builtins) Generate LOCHI for immediates
Post-RA pseudos (including z specific pass)	Select low/high instructions Expand mixed LOCRMux cases
Late if-conversion	Detect conditional trap, conditional return, conditional sibling call
Optimize comparison against zero (z specific pass)	Detect branch-on-count, load-and-trap Convert load to load-and-test Update CC users with CC mask for other insn
Fuse compare operations (z specific pass)	Detect compare-and-branch, compare-and-trap, compare-and-return, compare-and-sibcall

© 2018 IBM Corporation

GCC code generation for conditional instructions

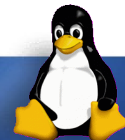
GCC pass	z/Architecture ISA handling
Expand	Select appropriate compare instructions Emit movcc pattern from COND_EXPR
Early if-conversion	Emit movcc pattern from if blocks Detect conditional trap
Doloop	Generate branch-on-count
Combine	Generate TEST UNDER MASK Convert load to load-and-test Detect load-and-trap Detect compare-and-branch, compare-and-trap Detect branch-on-index Optimize explicit uses of CC (e.g. builtins)
LRA / reload	Choose LOC/LOCR/LOCHI for movcc pattern
Post-reload splitters	Fix up branch-on-count, branch-on-index

© 2018 IBM Corporation

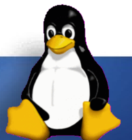
Misc. code generation differences

- **Decimal floating-point**
  - GCC supports _Decimal<N> types, LLVM does not
- **Inline expansion of string/memory functions**
  - LLVM only expands if length known at compile time
  - GCC generates EXECUTE loops
- **Rotate-then-insert/and/or/xor-selected-bits**
  - LLVM has C++ code to detect many cases
  - GCC only uses combine, misses some cases
- **Supported data types in vector registers**
  - LLVM does not support integer types in FPRs / VRs
  - GCC supports 64-bit integers in FPRs (moves only), 128-bit integers in VRs (moves and VAQ/VSQ)

© 2018 IBM Corporation



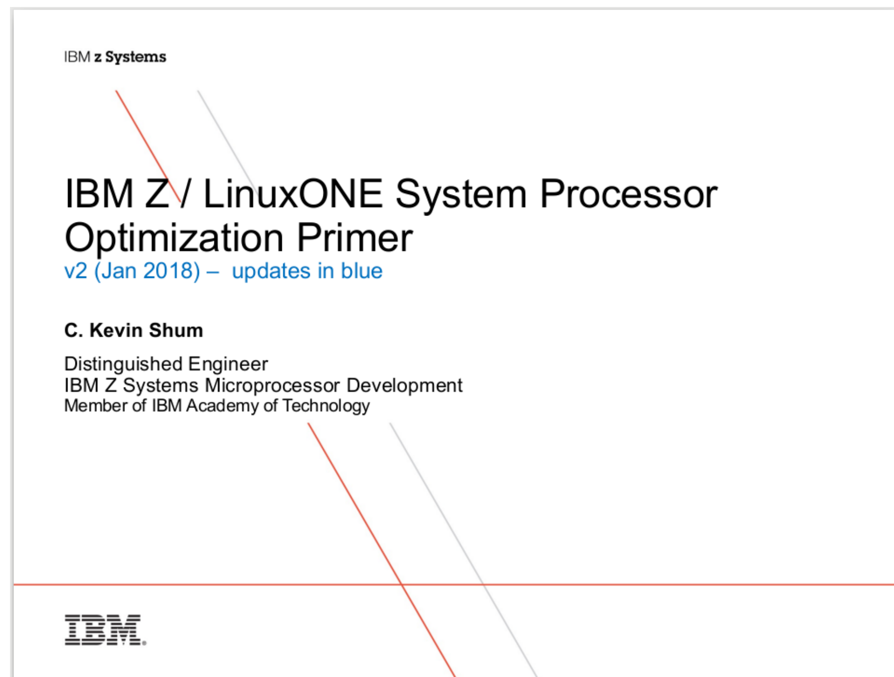
Processor Micro-Architecture Optimizations

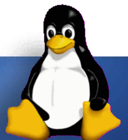


© 2018 IBM Corporation

z14 processor micro-architecture overview

- **Full micro-architecture is not publicly documented**
- **Overview targeted at compiler developers here:**
  - IBM Z / LinuxONE System Processor Optimization Primer





© 2018 IBM Corporation

z14 high-level instruction & execution flow

Branch prediction
Instruction cache / buffer
3 instructions
3 instructions
Instruction decode/ crack / dispatch / map
3 instructions
3 instructions
Issue queue side0
V. Branch Q
Issue queue side1
RBU0
RBU1
LSU 0
FXUb 0
FXUa 0
MP0
DV0
VFU0
SIMD0
BFU0
BFU2
DFU0
DFX0
FPD0
VFU1
SIMD1
BFU1
BFU3
DFU1
DFX1
FPD1
FXUa 1
FXUb 1
LSU 1
MP1
DV1
port 0
Data cache
port 1

(zEC12) new instruction flow and execution units for relative branches

(z13) additional instruction flow for higher core throughput

(z13) additional execution units for higher core throughput

(z13) new execution units to accelerate business analytics workloads

(z14) new SIMD floating-point units to provide 2x execution on single/double precisions

© 2018 IBM Corporation
IBM

z14 execution engine pipelines

Only 1 of 2 issue sides shown

- **Typical pipeline depths and bypass capabilities shown**
- **Some instructions may take longer to execute or bypass results**
- **Access registers not shown**

ACC – GR access
WB – GR write back

V-ACC – FPR/VR access
VWB – FPR/VR write back

CC – condition code calculation

BYP – data bypass network cycle

FPD, DFU – functions, e.g. divide, square-root, may take multiple passes through the pipeline

G2F – GR to VR/FPR moves
F2G – VR/FPR to GR moves

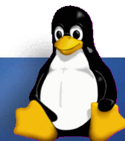
1 of 2 sides shown
ACC
WB
VWB
LSU
ACC
CC
WB
WB
FXA
ACC
CC
F2G
ACC
G2F
FXB (multiply/divide engines not shown)
VFU
DFX
Issue Queue
VWB
XS/PM
V-ACC
ST
SIMD
B Y P
XM
BFU (x2 in z14)
FPD
DFU
V Branch Queue
RBU

© 2018 IBM Corporation

IBM

Instruction scheduling

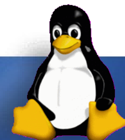
- **Goals of scheduling for z13 and z14**
  - **No** exact modeling of OOO execution phase possible
    - But: execution latencies still determine critical path length
  - Optimize decoder grouping
    - Sequence instructions so that decoder groups can be as large as possible (3 instructions) to optimize dispatch bandwith
  - Resource balancing
    - Sequence instructions so that over time, usage of the various execution units is as evenly balanced as possible
  - FPd side steering
    - Distribute long-running instructions (e.g. FP divide) evenly over both execution pipeline sides
  - FXU side steering
    - Distribute dependent instructions to the same side to enable result bypassing with reduced latency



© 2018 IBM Corporation

Instruction scheduling

- **Current LLVM implementation**
  - Post-RA scheduler as very last MI pass
    - Using new SchedStrategy and HazardRecognizer
    - Decoder grouping, resource balancing, FPd steering
    - FXU steering not yet implemented due to regressions
  - Pre-RA MI scheduler
    - Only considers latencies, resource balancing
    - “Mix up” register usage to get more freedom post-RA
      - Better decoder grouping; better FXU side steering; ...
    - But must be careful to not cause spilling!
    - Only slight performance benefit so far ...
- **Current GCC implementation**
  - Similar to LLVM; uses sched1 (pre-RA) and sched2 (post-RA)
  - Decoder grouping, FPd steering done post-RA
  - No FXU steering (yet)
  - Also does in-order scheduling for z10 and older processors



© 2018 IBM Corporation

Tuning code generation

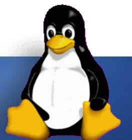
- **Caveat: performance results hard to predict**
  - Positive effects often dominated by negative second-order issues
  - E.g. increasing use of branch-on-count caused overall performance regression
  - Important goal: tune to avoid “second-order” effects
- **Loop unrolling**
  - Important to eliminate small loops which are sensitive
    - Loops should preferably be >12 instructions
  - LLVM: Enables "everything" to get rid of small loops, including forced unrolling
  - But: limit on number of stores to avoid running out of store tags
- **Loop strength reduction**
  - z13 supports only 12-bit unsigned offsets for vector memory accesses
  - Scalar code uses 20-bit signed offsets → try to avoid regressions in vectorizer
  - LLVM: New hook isFoldableMemAccessOffset() to handle this
- **Cost functions to tune vectorizer decisions**
- **Align functions to 16-byte boundary**
  - Helps reduce PHT aliasing second-order effects

© 2018 IBM Corporation

IBM

Performance History

© 2018 IBM Corporation

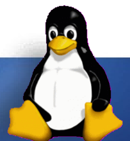


LLVM vs. GCC – performance history on z13

SPECint2006_base

Date axis
06/17/13 11/17/13 04/17/14 09/17/14 02/17/15 07/17/15 12/17/15 05/17/16 10/17/16 03/17/17 08/17/17 01/18/18 06/17/18

GCC
LLVM



© 2018 IBM Corporation

IBM

LLVM vs. GCC – performance history on z13

8: Loop distribution

5: Vectorizer tuning

8: Inliner tuning

5: Vector ISA &
z13 scheduling

7: Vectorizer tuning

GCC
LLVM

SPECint2006_base

4: Scheduling &
micro-arch tuning

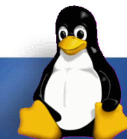
3.7: Vector ISA

3.4: ISA exploitation

06/17/13 11/17/13 04/17/14 09/17/14 02/17/15 07/17/15 12/17/15 05/17/16 10/17/16 03/17/17 08/17/17 01/18/18 06/17/18

© 2018 IBM Corporation

IBM



LLVM vs. GCC on IBM Z – optimization opportunities

LLVM pre-8.0 vs. GCC 8.1 (Jul 2018)

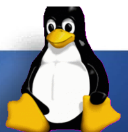
30%
20%
10%
0%
-10%
-20%
-30%
-40%
-50%
-60%

400.perlbench
401.bzip2
403.gcc
429.mcf
445.gobmk
456.hmmer
458.sjeng
462.libquantum
464.h264ref
471.omnetpp
473.astar
483.xalancbmk

LLVM uses **SLP** in one case where GCC does not, but the resulting code is worse. Possible cost function problem?

GCC (8) uses **loop distribution** in one case where LLVM does not. Can be "fixed" by adding a #pragma clang loop distribute(enable)

© 2018 IBM Corporation



Working on LLVM vs. GCC

Some observations from a back-end developer's perspective

© 2018 IBM Corporation

LLVM vs. GCC intermediate representations

- **Intermediate representations**
  - LLVM IR: High-level optimizations
    - GCC: gimple (& some aspects of trees ?)
  - SelectionDAG: Used for instruction selection within basic block
    - GCC: RTL ops semantics, e.g. in expand & combine
  - MI: Describes target instructions
    - Pre-RA form (in SSA) and post-RA form
    - GCC: RTL control and data flow only
  - MC: Represent binary code (assembler/disassembler)
    - Not in GCC; would be part of binutils
- **A note on documentation:**
  - LLVM IR very well documented, SelectionDAG/MI less so
  - GCC RTL very well documented, gimple less so

© 2018 IBM Corporation

LLVM vs. GCC back end

- **Many things look similar**
  - Sequence of passes
  - .td files vs .md files
  - C++ as implementation language
- **Differences**
  - LLVM seems to provide more flexibility in adding target-specific passes / overriding common passes
    - But LLVM back-end typically requires more custom C++ code than GCC back-end
  - No “reload” infrastructure (predicate vs. constraint distinction)
    - Common code instruction selection final after SelectionDAG
    - Any later insn changes (e.g. reg vs. mem variants) done by custom code
  - Support for machine code in LLVM
    - Can make .td files look more complex
    - The GCC/binutils split causes some semantical duplication
  - Platform ABI implementation
    - GCC back-end defines ABI directly in terms of tree types (mostly C types)
    - LLVM/clang split ABI definition between clang (C → LLVM IR) and LLVM (LLVM IR → assembler)

© 2018 IBM Corporation

IBM

LLVM vs. GCC – Back-end passes

GCC	LLVM
expand & combine	SelectionDAGISel
early split	EmitInstrWithCustomInserter
Early RTL opt passes	MachineSSAOptimization
sched	EmitSchedule
ira / lra / reload	RegAllocPass
n/a	addPreRegAlloc / addPostRegAlloc
thread_prologue_and_epilogue	PrologEpilogCodeInserter
Late RTL opt passes	MachineLateOptimization
late split	ExpandPostRAPseudos
sched2	PostRAScheduler
reorder_blocks	MachineBlockPlacements
machine_dependent_reorg	addPreEmitPass
final	EmitFile / EmitObjectCode

© 2018 IBM Corporation

LLVM vs. GCC – Machine definition

- **GCC .md file**

```
(define_insn "bswap<mode>2"
  [(set (match_operand:GPR                    "nonimmediate_operand" "=d,d,T")
        (bswap:GPR (match_operand:GPR 1 "nonimmediate_operand" " d,T,d")))]
  ""
  "@
   lrv<g>r\t%0,%1
   lrv<g>\t%0,%1
   strv<g>\t%1,%0"
  [(set_attr "type" "*,load,store")
   (set_attr "op_type" "RRE,RXY,RXY")
   (set_attr "z10prop" "z10_super")])
```

- **LLVM .td file**

```
def LRVR  : UnaryRRE<"lrvr",  0xB91F, bswap, GR32, GR32>;
def LRVGR : UnaryRRE<"lrvgr", 0xB90F, bswap, GR64, GR64>;
def LRV   : UnaryRXY<"lrv",  0xE31E, z_lrv,  GR32, 4>;
def LRVG  : UnaryRXY<"lrvg", 0xE30F, z_lrvg, GR64, 8>;
def STRV  : StoreRXY<"strv",  0xE33E, z_strv,  GR32, 4>;
def STRVG : StoreRXY<"strvg", 0xE32F, z_strvg, GR64, 8>;
```

© 2018 IBM Corporation
IBM

Hands-on code generation demo

- **Test code**

```
int test (int flag, int *ptr)
{
  if ((flag & 2) && !(flag & 4))
    *ptr = 1;

  return 0;
}
```

- **Resulting assembler code (both compilers)**

```
        tml     %r2,6
        jnl     .L2
        mvhi    0(%r3),1
.L2:
        lghi    %r2,0
        br      %r14
```


© 2018 IBM Corporation

Summary

- **Critical use cases for both GCC and LLVM**
  - Need to maintain good support for both compilers across IBM hardware platforms
- **Competition is healthy**
  - No longer any clear favorite w.r.t. performance
  - Both compilers benefit from performance comparison
- **GCC vs. LLVM code base**
  - Similar concepts, different implementation ...
  - GCC back-end developers should be able to work on LLVM back end with little difficulties and vice versa

© 2018 IBM Corporation

Questions

© 2018 IBM Corporation