LLVM for the Apollo Guidance Computer

Lewis Revill
About Me

- Studying at University of Bath, UK
- Joined Embecosm as a UKESF scholar in July 2018
- Working primarily on LLVM backends
- Started working on the AGC backend in October as a personal project
The Apollo Guidance Computer - Background

- Designed for use in NASA’s Apollo Program
- Used on command module & lunar lander
- Designed in three iterations:
  - block I: the prototype (Apollo 1 - 6)
  - block II: the release (Apollo 7 – 17)
  - block III: the ‘fantasy’ (never built)

The Apollo Guidance Computer - Architecture

• Memory-based registers

A  EQUALS  0
L  EQUALS  1  # L AND Q ARE BOTH CHANNELS AND REGISTERS
Q  EQUALS  2
EBANK  EQUALS  3
FBANK  EQUALS  4
Z  EQUALS  5  # ADJACENT TO FBANK AND BBANK FOR DXCH Z
BBANK  EQUALS  6  # (DTCB) AND DXCH FBANK (DTCF).
                  # REGISTER 7 IS A ZERO-SOURCE, USED BY ZL.
ARUPT  EQUALS  10  # INTERRUPT STORAGE
LRUPT  EQUALS  11
QRUPT  EQUALS  12
SAMPTIME  EQUALS  13  # SAMPLED TIME 1 & 2.
ZRUPT  EQUALS  15  # (13 AND 14 ARE SPARES.)

• Von Neumann architecture

• Erasable (read/write) & fixed (read-only) memory
The Apollo Guidance Computer - Architecture

• 16 bits per word
  - bits 15-1 represent a 15-bit ones-complement number or an instruction
  - bit 0 is an odd parity bit

<table>
<thead>
<tr>
<th>sign</th>
<th>data</th>
<th>parity</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
</tr>
</tbody>
</table>

• Adjacent words can form double words
• When interpreted as fixed point, the MSB represents $2^{-1}$
The Apollo Guidance Computer - ISA

- Accumulator-based instructions
- Complex multi-purpose instructions
  - e.g. CCS, EXTEND, INDEX
- Nowhere near GNU-like assembly syntax

```
OCT10000 = BIT13
OCT30000 = PRI030
OCT7777 = OCT 7777
STIKSTRT DEC 0.825268  # 20 D/S MAXIMUM COMMANDED RATE
60DEC DEC 60
RSFLGBTS OCT 20100
MAXDB OCTAL 03434
# 5 DEG ATTITUDE DEADBAND, SCALED AT 45.
LIGHTSET CAF BIT5
# CHECK FOR MARK REJECT AND ERROR RESET
EXTEND
RAND NAVKEYIN
EXTEND
BZF NONAVKEY  # NO MARK REJECT
```
AGC LLVM - Motivation

• Programming the AGC in C
• How well will LLVM cope?
  - can it be done while avoiding ‘hacks’ in generic code?
  - what is lacking that might be useful for other targets?
• How well will I cope?
  - first backend implemented from scratch

Image courtesy of MIT Museum., Public Domain
AGC LLVM – Register Definitions

• Special definitions for R0-R7

// Accumulator register. Used to store the result of operations done by the CPU. // Consists of 16 bits due to the addition of an overflow/underflow bit.
def R0 : AGCReg<0, "A">, DwarfRegNum<0>;

// Lower accumulator register. Used to store the lower half of the result of // operations done by the CPU when these operations are 'double precision'. Only // 15 bits used in contrast to the standard accumulator.
def R1 : AGCReg<1, "L">, DwarfRegNum<1>;

// Double accumulator register. Composed of both the upper and lower accumulator // registers. The effective value in this register is the concatenation of the // lower 14 bits of both registers, with the sign bits in each register set to // match each other. The overflow bit in the upper accumulator is set // accordingly.
let SubRegIndices = [subreg_upper, subreg_lower] in
def RD0 : AGCRegWithSubRegs<0, "AL", [R0, R1], DwarfRegNum<8>);

...  

// Define the accumulator as its own register class.
def Acc : RegisterClass<"AGC", [i16], 1, (add R0)>;

// Define the lower accumulator as its own register class.
def AccLower : RegisterClass<"AGC", [i16], 1, (add R1)>;

// Define the upper/lower accumulator pair as its own register class.
def AccPair : RegisterClass<"AGC", [i32], 1, (add RD0)>;
AGC LLVM – Register Definitions

• Generated definitions for other memory-registers

... 

foreach Index = 8-4095 in {
    def R#Index : AGCReg<Index, "R"#Index>, DwarfRegNum<[-1]>;
}

foreach Index = 1-4094 in {
    let SubRegIndices = [subreg_upper, subreg_lower] in
    def RD#Index : AGCRegWithSubRegs<Index, "RD"#Index,
        ![cast<AGCReg>("R"#Index),
        !cast<AGCReg>("R"#!add(Index, 1))],
        DwarfRegNum<[-1]>;
}

// Define a register class that maps directly to all memory locations.
def MM12 : RegisterClass<"AGC", [i16], 4096, (add
    (sequence "R%u", 0, 4095)
)>;

// Define a register class that maps directly to all memory pairs.
def MMD12 : RegisterClass<"AGC", [i32], 4095, (add
    (sequence "RD%u", 0, 4094)
)>;
AGC LLVM – Instruction definitions

- Extracodes denoted by an isExtracode bit

```plaintext
// Add value of general purpose memory location to accumulator. This instruction
// writes back the original value of the memory location, so memory locations
// which edit their contents are updated. Note that this occurs even though the
// operand is not necessarily erasable.
let Constraints = "$k = $k_wb" , isAdd = 1 in
def AD : ALUReadInst<0b110, (outs Acc:$a_dst, mem12:$k_wb),
( ins Acc:$a, mem12:$k), "ad">;

// Double precision add value of accumulator to general purpose memory location.
// This instruction writes the sign of the output to the accumulator, and writes
// zero to the lower accumulator.
let Constraints = "$k = $k_dst" , isAdd = 1 in
def DAS : ALUWriteInst<0b01000, (outs mem10:$k_dst, AccPair:$al_wb),
( ins mem10:$k, AccPair:$al), "das">;

// Subtract value of general purpose memory location from accumulator. This
// instruction writes back the original value of the memory location, so memory
// locations which edit their contents are updated.
let Constraints = "$k = $k_wb" , isExtracode = 1,
   DecoderNamespace = "Extracode" in
def SU : ALUWriteInst<0b11000, (outs Acc:$a_dst, mem10:$k_wb),
( ins Acc:$a, mem10:$k), "su">;
```

...
AGC LLVM – DecodeNullOps

- Useful for architectures with ‘hidden’ operands, like an accumulator

```cpp
class Instruction {
    ...
    /// DecodeNullOps - Whether this instruction has operands with no encoding,
    /// implying that decoding should produce null operands for this instruction
    /// when there are no bits to decode.
    bit DecodeNullOps = 0;
    ...
}
```

- Ensures MCInstrs have all DAG operands present when decoded.

```cpp
static bool populateInstruction(...){
    ...
    // For each operand, see if we can figure out where it is encoded.
    for (const auto &Op : InOutOperands) {
        ...
        // If this is an operand with no encoding but DecodeNullOps is set, let the
        // target decode a default 0-width operand.
        if (OpInfo.numFields() == 0 && Def.getValueAsBit("DecodeNullOps"))
            OpInfo.addField(0, 0, 0);
        ...
    }
    ...
}```
AGC LLVM – Directives

• Parsing an AGC directive like ‘${FILENAME}.agc’ requires extra work

```cpp
if (IDVal[0] == '$' && IDVal != "$")
    return parseAGCFileDirective(IDVal.drop_front());
```

```cpp
/// parseAGCFileDirective
/// ::= $filename
bool AsmParser::parseAGCFileDirective(StringRef Filename) {
    SMLoc DirectiveLoc = getTok().getLoc();

    // Since the filename is not escaped it may appear as multiple tokens.
    // Retrieve the full filename by combining the strings.
    if (getTok().isNot(AsmToken::EndOfStatement)) {
        unsigned Length = Filename.size() + getTok().getString().size() +
            Lexer().LexUntilEndOfStatement().size();
        Filename = StringRef(Filename.data(), Length);
    }

    // Attempt to switch the lexer to the included file before consuming the
    // end of statement to avoid losing it when we switch.
    if (check(enterIncludeFile(Filename.str()), DirectiveLoc,
        "Could not find file " + Filename.str() + "")
        return true;
    }

    return false;
}
AGC LLVM – Directives

- AGC also uses ‘identifier EQUALS value’ for assignment

```c
... case AsmToken::Equal:
    if (!getTargetParser().equalIsAsmAssignment())
        break;
    // identifier '=' ... -> assignment statement
    Lex();
    return parseAssignment(IDVal, true);

/* BEGIN AGC CODE */
case AsmToken::Identifier: {
    StringRef NextVal = Lexer.getTok().getString().lower();
    if (NextVal != "equals")
        break;
    // identifier 'equals' ... -> assignment statement
    Lex();
    return parseAssignment(IDVal, true);
}
/* END AGC CODE */
...```
AGC LLVM – Directives

- ELF sections were a good fit for some AGC directives

```c
void AGCMCELFStreamer::EmitInstruction(...) {
  switch (Inst.getOpcode()) {
    ...
    case AGC::DirectiveBANK: {
      // The bank directive informs the assembler to output to another bank.
      // This is implemented by switching to a new section and letting the
      // linker handle it.
     StringRef SectionName = "BANK" + itostr(Inst.getOperand(0).getImm());
      MCSection *Section = ((MCSection *)getContext().getELFSection(
        SectionName, ELF::SHT_PROGBITS, ELF::SHF_EXECINSTR | ELF::SHF_ALLOC));
      SwitchSection(Section, nullptr);
      return;
    }
    case AGC::DirectiveSETLOC: {
      // The setloc directive informs the assembler to output to an explicit
      // address. This is implemented by switching to a new section encoding the
      // address and letting the linker handle it.
     StringRef SectionName = "ADDR" + itostr(Inst.getOperand(0).getImm());
      MCSection *Section = ((MCSection *)getContext().getELFSection(
        SectionName, ELF::SHT_PROGBITS, ELF::SHF_EXECINSTR | ELF::SHF_ALLOC));
      SwitchSection(Section, nullptr);
      return;
    }
  }
  MCELFStreamer::EmitInstruction(Inst, STI);
}
```
AGC LLVM – Directives

- Other directives were simple to handle

```c
void AGCMCCodeEmitter::encodeInstruction(...) const {
  unsigned Opcode = MI.getOpcode();
  switch (Opcode) {
    default:
      break;
    case AGC::DirectiveERASE: {
      int64_t NumWords = MI.getOperand(0).getImm();
      for (; NumWords > 0; --NumWords)
        emitBitsWithParity(OS, 0x0000);
      return;
    }
    case AGC::DirectiveOCT:
      emitBitsWithParity(OS, (uint16_t)MI.getOperand(0).getImm());
      return;
    }
  ...
}
AGC LLVM – Parity

- Instructions should be emitted with an odd parity bit

```cpp
// Determine the odd parity bit that applies to the given 15-bit instruction code.
static uint16_t getParityBitForEncoding(uint16_t Enc) {
    // Accumulate the correct parity bit for odd parity.
    uint16_t ParityBit = 1;
    for (int i = 0; i < 15; i++)
        ParityBit ^= ((Enc >> i) & 1);

    return ParityBit;
}

// Append an odd parity bit to a 15-bit binary code.
static uint16_t getEncodingWithParity(uint16_t Bits) {
    return (Bits << 1) | getParityBitForEncoding(Bits);
}

void AGCMCCodeEmitter::emitBitsWithParity(raw_ostream &OS, uint16_t Bits) {
    uint16_t Encoding = getEncodingWithParity(Bits);
    support::endian::write<uint16_t>(OS, Encoding, support::big);
}
```
AGC LLVM – Extracodes

- Parsed extracode instructions must be preceded by EXTEND

```cpp
bool AGCAsmParser::ParseInstruction(...) {
    ParsingExtracode = ParsedExtend;
    ParsedExtend = false;
    ...
}

unsigned AGCAsmParser::checkEarlyTargetMatchPredicate(...) {
    unsigned Opcode = Inst.getOpcode();

    // Indicate that the next instruction parsed should be an extracode when an
    // EXTEND instruction is encountered.
    if (Opcode == AGC::EXTEND) {
        ParsedExtend = true;
        return Match_IgnoredExtend;
    }

    bool InstIsExtracode = MII.get(Opcode).TSFlags & AGCII::IsExtracode;

    // Check that extracode instructions are preceded by an EXTEND instruction.
    if (!ParsingExtracode)
        return InstIsExtracode ? Match_ExtracodeFail : Match_Success;

    // Check that non-extracode instructions are not preceded by an EXTEND
    // instruction.
    return InstIsExtracode ? Match_Success : Match_NonExtracodeFail;
}
```
AGC LLVM – Extracodes

- Emitted extracode instructions must be preceded by extend

```cpp
void AGCMCCodeEmitter::encodeInstruction(...) const {

    ... 

    if (Desc.TSFlags & AGCII::IsExtracode) {
        // Prefix this instruction with an EXTEND instruction.
        emitBitsWithParity(OS, 0x0006);
        ++MCNumEmitted;
    }

    uint16_t Bits = getBinaryCodeForInstr(MI, Fixups, STI);
    emitBitsWithParity(OS, Bits);
    ++MCNumEmitted;
}
```
AGC LLVM – Extracodes

- Extracode instructions share encodings with non-extracodes

```cpp
DecodeStatus AGCDisassembler::getInstruction(...) const {
    uint16_t Instruction;
    DecodeStatus Result;

    // Try non-extracode instructions first.
    Instruction = support::endian::read16be(Bytes.data());
    // Mask off the parity bit.
    Instruction = (Instruction & 0xFFF) >> 1;
    Result =
        decodeInstruction(DecoderTable16, MI, Instruction, Address, this, STI);

    if (MI.getOpcode() != AGC::EXTEND) {
        Size = 2;
        return Result;
    }

    // Try parsing the following instruction using the decoder table for extracodes.
    Instruction = support::endian::read16be(Bytes.drop_front(2).data());
    // Mask off the parity bit of the second instruction.
    Instruction = (Instruction & 0xFFF) >> 1;
    Result = decodeInstruction(DecoderTableExtracode16, MI, Instruction, Address, this, STI);

    Size = 4;
    return Result;
}
```
AGC LLVM – ALU Lowering

... Generic pattern classes

class PatAccMem12<SDPatternOperator OpNode, AGCInst12 Inst>
    : Pat<(OpNode Acc:$src1, MM12:$src2), (Inst Acc:$src1, mem12:$src2)>;
class PatAccMem10<SDPatternOperator OpNode, AGCInst10 Inst>
    : Pat<(OpNode Acc:$src1, MM10:$src2), (Inst Acc:$src1, mem10:$src2)>;

def : PatAccMem12<add, AD>;
def : PatAccMem10<sub, SU>;
def : PatAccMem12<and, MASK>;
...
AGC LLVM – ALU Lowering

AGCTargetLowering::AGCTargetLowering(...) {
    ...
    setOperationAction(ISD::MUL, MVT::i32, Custom);
    setOperationAction(ISD::SDIV, MVT::i32, Custom);
    setOperationAction(ISD::SREM, MVT::i32, Custom);
}

SDValue AGCTargetLowering::LowerMUL(SDValue Op, SelectionDAG &DAG) const {
    assert(Op.getValueType() == MVT::i32 && "MUL should be i32 only");

    if (Op.getOperand(0).getOpcode() != ISD::SIGN_EXTEND ||
        Op.getOperand(1).getOpcode() != ISD::SIGN_EXTEND)
        report_fatal_error("cannot lower MUL");

    SDValue Arg0 = Op.getOperand(0).getOperand(0);
    SDValue Arg1 = Op.getOperand(1).getOperand(0);

    if (Arg0.getValueType() != MVT::i16 || Arg1.getValueType() != MVT::i16)
        report_fatal_error("cannot lower MUL");

    // Replace the node (mul (sign_extend arg0), (sign_extend arg1)) with
    // (MP arg0, arg1).
    return SDValue(DAG.getMachineNode(AGC::MP, SDLoc(Op), MVT::i32, Arg0, Arg1), 0);
}
AGC LLVM – Materializing Constants

- Requires consideration as no instruction takes an immediate operand

```c
void AGCDAGToDAGISel::Select(SDNode *Node) {
...
for (auto &Op : Node->ops()) {
    switch (Op.getNode()->getOpcode()) {
    default:
        break;
    case ISD::Constant: {
        SelectAGCConstant(Op.getNode());
        break;
    }
    }
}
...}
```

```c
void AGCDAGToDAGISel::SelectAGCConstant(SDNode *Node) {
    assert(Node->getValueType(0) == MVT::i16 &&
           "Non-i16 constants cannot yet be materialized");

    SDLoc DL(Node);

    int64_t ConstImm = cast<ConstantSDNode>(Node)->getSExtValue();
    SDValue ConstValue = CurDAG->getTargetConstant(ConstImm, DL, MVT::i16);
    // Wrap constant nodes with a pseudo instruction that we can materialize later
    // for the appropriate data representation.
    SDNode *New =
        CurDAG->getMachineNode(AGC::PseudoCONST, DL, MVT::i16, ConstValue);
    ReplaceNode(Node, New);
}
bool AGCExpandPseudo::expandPseudoCONST(...) {
    ...
    unsigned DestReg = MI.getOperand(0).getReg();
    int64_t Constant = MI.getOperand(1).getImm();

    // Get the constant as represented in 15-bit ones complement.
    if (!toAGCConstant(Constant))
        report_fatal_error("Could not represent constant");

    MachineBasicBlock *NewMBB = MF->CreateMachineBasicBlock(MBB.getBasicBlock());
    MF->insert(++MBB.getIterator(), NewMBB);

    // Emitting the following sequence will allow the constant to be materialized
    // in the correct location:
    //
    // SETLOC DestReg
    // OCT Constant
    BuildMI(NewMBB, DL, TII->get(AGC::DirectiveSETLOC)).addReg(DestReg);
    BuildMI(NewMBB, DL, TII->get(AGC::DirectiveOCT)).addImm(Constant);

    // Now we need to return to the original output location of the function.
    const TargetMachine &TM = MF->getTarget();
    StringRef SectionName =
        (((MCSectionELF *)TM.getObjFileLowering()->SectionForGlobal( &MF->getFunction(), TM))
            ->getSectionName());

    BuildMI(NewMBB, DL, TII->get(AGC::DirectiveBANK))
        .addImm(toBankNumber(SectionName));
    ...
}
AGC LLVM – Future Work

• Generating correct IR from front & middle ends
  - don’t assume 8-bit bytes
  - don’t assume twos complement
  - allow only a small subset of C to compile

• Eliminate GNU/ELF directives from assembly output
  - output files should assemble with yaAGC
    (github.com/virtualagc/virtualagc)

• Implement a linker

• Emulate a stack with INDEX instructions

• Lower control flow statements to CCS
LLVM pain points

- AsmParser is not flexible to non-GNU-like assembly
- FixedLenDecoderEmitter requires DecodeNullOps patch to function for MCInsts with hidden operands
- The lexer converts octal immediates itself without the target knowing
Questions?

www.embecosm.com

github.com/lewis-revill/agc-llvm.git