EMBECOSM®

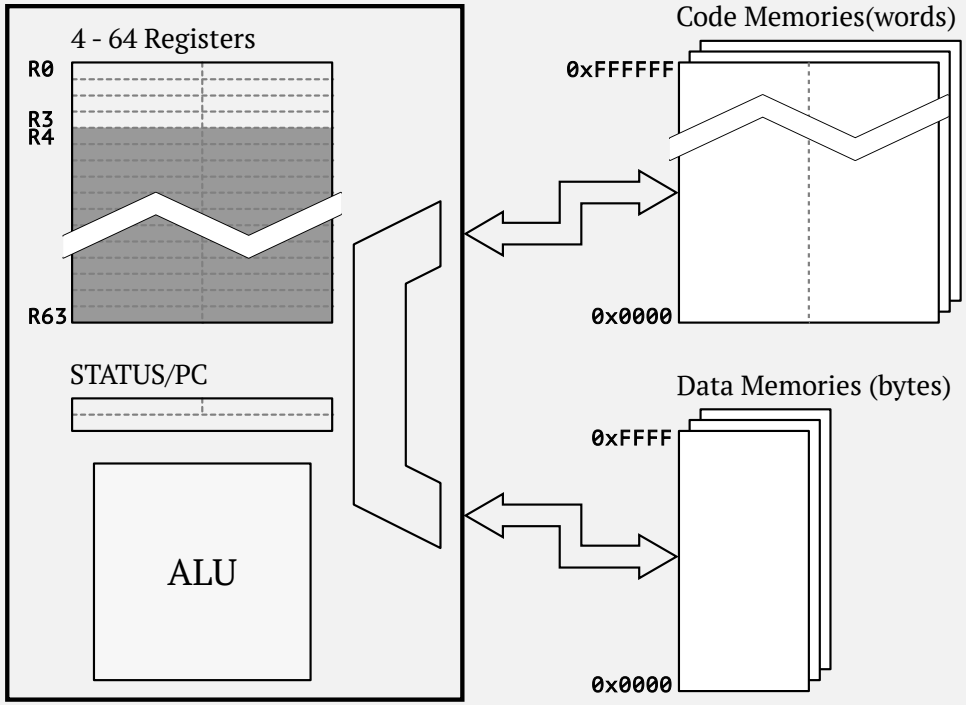
AAPSim: Implementing a LLVM Based Simulator

Simon Cook

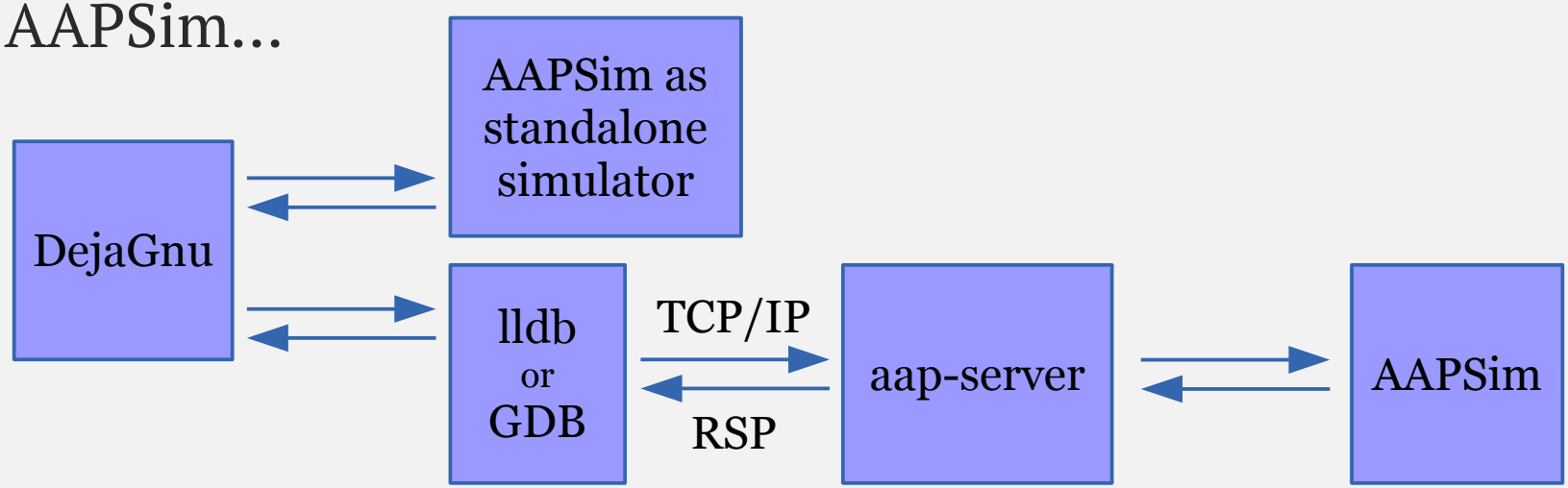
simon.cook@embecosm.com

Quick Recap: AAP

- 16-bit Harvard architecture
  - 16-bit byte addressed data
  - 24-bit word addressed code
- 4-64 16-bit registers
- LLVM tool chain
- FPGA Implementation
- MC based Simulator



- Run-time testing primarily via GCC Regression Suite
- Executed either directly or via lldb/GDB
- Both need a model of the processor to validate execution
- So I wrote AAPSim…



An LLVM Simulator? The case for MC

- Steps required for a simulator
  - Disassemble instruction
  - Carry out meaningful instruction
- LLVM MC
  - TableGen'd disassemble tables
  - Useful container for instructions (`MCInst`)
- Rapid development of Architecture
  - Need to only change one place to affect assembly and simulation

Instruction Example

```
class Inst_rrr<bits<4> opclass, bits<8> opcode,
               dag outs, dag ins, string asmstr,
               list<dag> pattern>
    : InstAAP<opclass, opcode, outs, ins,
              asmstr, pattern> {
  bits<6> rD;
  bits<6> rA;
  bits<6> rB;
  let Inst{8-6} = rD{2-0};
  let Inst{24-22} = rD{5-3};
  let Inst{5-3} = rA{2-0};
  let Inst{21-19} = rA{5-3};
  let Inst{2-0} = rB{2-0};
  let Inst{18-16} = rB{5-3};
}
```

```
multiclass ALU_r<bits<8> opcode, string opname,
                 SDNode OpNode> {
  def _r : Inst_rrr
    <0x1, opcode, (outs GR64:$rD),
                  (ins GR64:$rA, GR64:$rB),
      !strconcat(opname, "\t$rD, $rA, $rB"),
      [(set GR64:$rD,
        (OpNode GR64:$rA, GR64:$rB))]>;
}

let Defs = [PSW] in  defm ADD : ALU_r<0x1, "add",
                                      add>;
```

Implementation

Simulator State

- Need to hold processor state
- For memory, can just new whole array as memories only 128KiB
- Simulator state flag

```
class AAPSimState {
  // Registers
  uint16_t base_regs[64];
  uint32_t pc_w : 24;

  // Special registers
  uint16_t exitcode;      // Exit code register
  uint16_t overflow : 1;  // Overflow bit register
  SimStatus status;       // Simulator status

  // One code and data namespace each
  // LLVM expectes 8-bit addressed memory, so
  // provide as such
  uint8_t *code_memory;
  uint8_t *data_memory;
};
```

Fetch/Decode

- With the program memory as an `ArrayRef<uint8_t>`
  - Get instruction from `MCDisassembler`
  - Calculate default new program counter (PC + Size)
- If decode fails, return `SIM_INVALID_INSN`

```
SimStatus AAPSimulator::fetch() {
  MCInst Inst;
  uint64_t Size;
  uint32_t pc_w = State.getPC();
  ArrayRef<uint8_t> *Bytes = State.getCodeArray();

  if (DisAsm->getInstruction(Inst, Size, Bytes->slice(pc_w << 1),
                             (pc_w << 1), nulls(), nulls())) {
    // Decode was successful, calculate default new PC
    uint32_t newpc_w = pc_w + (Size >> 1);
    SimStatus status;
    // TODO Execute instruction
    State.setPC(newpc_w);

    return status;
  }
  else {
    // Unable to read/decode an instruction. If the memory raised an
    // exception, pass this on, otherwise return invalid instruction.
    if (State.getStatus() != SimStatus::SIM_OK)
      return SimStatus::SIM_INVALID_INSN;
    return State.getStatus();
  }
}
```

Execute

Using MCInst from previous step, switch on opcode

- For each instruction implement any state changes the instruction carries out.

```
SimStatus AAPSimulator::exec(MCInst &Inst, uint32_t pc_w, uint32_t &newpc_w) {
  switch (Inst.getOpcode()) {
    // Unknown instruction
    default:
      llvm_unreachable("No simulator support for this instruction");
      break;

    // ADD
    case AAP::ADD_r:
    case AAP::ADD_r_short: {
      int RegDst = getLLVMReg(Inst.getOperand(0).getReg());
      int RegSrcA = getLLVMReg(Inst.getOperand(1).getReg());
      int RegSrcB = getLLVMReg(Inst.getOperand(2).getReg());
      EXCEPT(uint32_t ValA = signExtend16(State.getReg(RegSrcA)));
      EXCEPT(uint32_t ValB = signExtend16(State.getReg(RegSrcB)));
      uint32_t Res = ValA + ValB;
      EXCEPT(State.setReg(RegDst, static_cast<uint16_t>(Res)));
      // Test for overflow
      int32_t Res_s = static_cast<int32_t>(Res);
      State.setOverflow(static_cast<int16_t>(Res_s) != Res_s ? 1 : 0);
      break;
    }

    // ... other cases not shown ...
  } // end opcode switch

  // By default, we executed the instruction
  return SimStatus::SIM_OK;
}
```

Nop

AAP uses the NOP instruction for simulator control

- NOP 0 – Breakpoint
- NOP 1 – Nop
- NOP 2 – Exit with Retcode in Rd
- NOP 3 – Write char Rd to stdout
- NOP 4 – Write char Rd to stderr

```
switch (Inst.getOpcode()) {
  // NOP Handling
  // 0: Breakpoint
  case AAP::NOP:
  case AAP::NOP_short: {
    int Reg = getLLVMReg(Inst.getOperand(0).getReg());
    uint16_t Command = Inst.getOperand(1).getImm();
    // Load register value and char for NOPs that require it
    EXCEPT(uint16_t RegVal = State.getReg(Reg));
    char c = static_cast<char>(RegVal);
    switch (Command) {
      case 0: return SimStatus::SIM_BREAKPOINT;
      default:  // Treat unknown values as NOP
      case 1: break;
      case 2:
        State.setExitCode(RegVal);
        return SimStatus::SIM_QUIT;
      case 3:
        outs() << c;
        break;
      case 4:
        errs() << c;
        break;
    }
    break;
  }

  // ... other cases not shown ...
} // end opcode switch
```

Running programs: aap-run

- step()
  - Function to drive one cycle of processor
  - Returns simulator status value based on CPU exception, etc.

- aap-run: Loads object and executes
  - Core: `while (true) step();`

```
int main(int argc, char **argv) {
   // Load Binary
  LoadBinary(Sim, InputFilename);
  while (status == SimStatus::SIM_OK) {
    status = Sim.step();
  }
  // Deal with the final simulator status
  switch (status) {
    default: break;
    case SimStatus::SIM_INVALID_INSN:
      dbgs() << " *** Invalid instruction ***\n";   break;
    case SimStatus::SIM_BREAKPOINT:
      dbgs() << " *** Breakpoint hit ***\n";        break;
    case SimStatus::SIM_TRAP:
      dbgs() << " *** Simulator trap ***\n";        break;
    case SimStatus::SIM_EXCEPT_MEM:
      dbgs() << " *** Invalid memory trap ***\n";   return 1;
    case SimStatus::SIM_EXCEPT_REG:
      dbgs() << " *** Invalid register trap ***\n"; return 1;
    case SimStatus::SIM_QUIT:
      dbgs() << " *** EXIT " << Sim.getState().getExitCode() << " ***\n";
      return Sim.getState().getExitCode();
  }
  return 0;
}
```

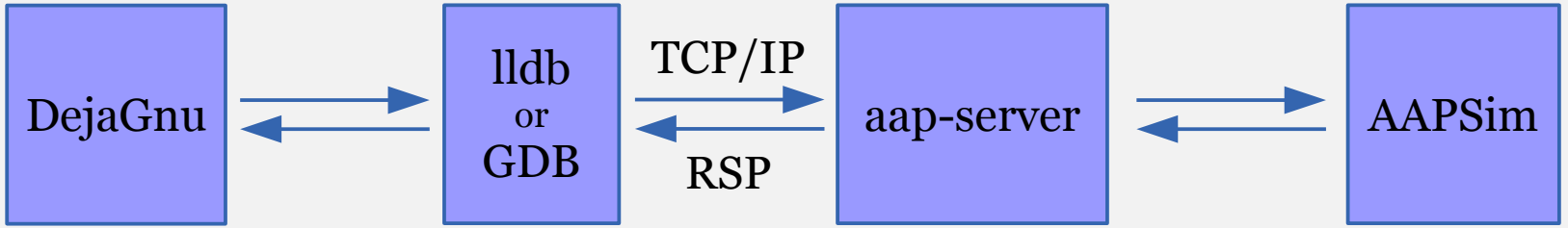
Running programs: aap-server

- RSP: textual packet protocol
  - Read/write regs: `g/G`
  - Read/write mem: `m/M`
  - Step/Cont: `s/c`
  - Status: ?

```
void
GdbServer::rspClientRequest ()
{
  switch (pkt->data[0])
    {
    case 's':
      rspReportException (mSim->step ());
      return;

    // ... other cases not shown ...

    }
} // rspClientRequest ()
```



Demo

The future

The Future... llvm-tblgen -gen-simulator?

- Most semantic meaning already exists in some form
  - SelectionDAG patterns could be mapped?
  - Already know register sizes
  - Pipeline information known for scheduling
- Need to add information about memories, custom DAG nodes

The Future... llvm-tblgen -gen-simulator?

- Most semantic meaning already exists in some form
  - SelectionDAG patterns could be mapped?
  - Already know register sizes
  - Pipeline information known for scheduling
- Need to add information about memories, custom DAG nodes
- Already exists in GNU binutils-gdb
  - CGEN can be used in TableGen style to build assembler/disassemble (libopcodes)
  - Semantic information added to allow a simulator to be built
  - Many targets already do this

Resources

- AAP Specification
  - www.embecosm.com/EAN13
- Source
  - github.com/embecosm/aap-llvm
  - github.com/embecosm/aap-*
- Simulator
  - lib/Target/AAPSimulator
  - tools/aap-run

Thank You

www.embecosm.com

simon.cook@embecosm.com

getLLVMReg (Reg)

```
static int getLLVMReg(unsigned Reg) {
  switch (Reg) {
    default: llvm_unreachable("Invalid register");
#define REG(x) case AAP::R##x: return x;
    REG(0)  REG(1)  REG(2)  REG(3)  REG(4)  REG(5)  REG(6)  REG(7)
    REG(8)  REG(9)  REG(10) REG(11) REG(12) REG(13) REG(14) REG(15)
    REG(16) REG(17) REG(18) REG(19) REG(20) REG(21) REG(22) REG(23)
    REG(24) REG(25) REG(26) REG(27) REG(28) REG(29) REG(30) REG(31)
    REG(32) REG(33) REG(34) REG(35) REG(36) REG(37) REG(38) REG(39)
    REG(40) REG(41) REG(42) REG(43) REG(44) REG(45) REG(46) REG(47)
    REG(48) REG(49) REG(50) REG(51) REG(52) REG(53) REG(54) REG(55)
    REG(56) REG(57) REG(58) REG(59) REG(60) REG(61) REG(62) REG(63)
#undef REG
  }
}
```

EXCEPT (...)

```
// Register and memory exception handlers
#define EXCEPT(x) x; \
  if (State.getStatus() != SimStatus::SIM_OK) \
    return State.getStatus()
```

LoadObject (Sim, o)

```
static void LoadObject(AAPSimulator &Sim, ObjectFile *o) {
  for (const SectionRef &Section : o->sections()) {
    uint64_t Address = Section.getAddress();
    uint64_t Size = Section.getSize();
    bool Text = Section.isText(); bool Data = Section.isData();
    bool TextFlag = Address & 0x1000000;
    StringRef BytesStr;
    Section.getContents(BytesStr);
    // FIXME: Should  load based on LOAD segment flag
    if (Text || Data) {
      if (TextFlag) {
        Address = Address & 0xffffff;
        Sim.WriteCodeSection(BytesStr, Address);
      } else {
        Address = Address & 0xffff;
        Sim.WriteDataSection(BytesStr, Address);
      }
    }
  }
  // Set PC (should load from ELF)
  Sim.setPC(0x0);
}
```