Topics

- CPU selection
- Application-specific processors in SoCs
- Instruction set design
- Compilers

Figures of Merit in CPU Selection

- Performance on the application:
  - Average case
  - Worst-case
- Power/energy consumption
- Interrupt handling latency
- Context switch time
- Other issues:
  - Code compatibility
  - Development environment
  - Fab support
CPU Families Example: ARM

- **Low end:**
  - No cache
  - No floating point
  - No MMU

- **High end:**
  - Cache
  - Floating-point
  - MMU
Configurable vs. Reconfigurable

- **Configurable:**
  - CPU architectural features are selected at design time

- **Reconfigurable:**
  - Hardware can be reconfigured in the field
  - May be dynamically reconfigured during execution
Tensilica Configurable Processors

- Configurability:
  - Processor parameters (cache size, etc.)
  - Instructions

- Result:
  - HDL model for processor
  - Software development environment
Xtensa Configurability

- **Instruction set:**
  - ALU extensions, coprocessors, wide instructions, DSP-style, function unit implementation

- **Memory:**
  - I cache config, D cache config, memory protection/translation, address space size, mapping of special-purpose memories, DMA access

- **Interface:**
  - Bus width, protocol, system register access, JTAG, queue interfaces to other processors

- **Peripherals:**
  - Timers, interrupts, exceptions, remote debug
TIE Extensions

- Tensilica Instruction Extension (TIE) language used to define instruction set definitions
  - State declarations
  - Instruction encodings and formats
  - Operation descriptions
Regfile LR 16 128

Operation add128

{out LR sr, in LR ss, in LR st}

{ assign sr = st + ss;}
main() {
    int i;
    LR src1[256], src2[256], src3[256];
    for (i=0; i<256; i++)
        dest[i] = add128(src1[i], src2[i]);
}
Performance Improvement

• Compare Xtensa optimized vs. Xtensa out-of-the-box:
  • Compare performance/MHz
• EEMBC ConsumerMark:
  • Xtensa optimized: 2.02
  • Xtensa out-of-the-box: 0.66
• EEMBC TeleMark:
  • Xtensa optimized: 0.47
  • Xtensa out-of-the-box: 0.23
• EEMBC NetMarks:
  • Xtensa optimized: 0.123
  • Xtensa out-of-the-box: 0.03
In-Class Exercise

- Operations for which an instruction extension may be useful:
  - Example 1: bit reversal
  - Example 2: majority function
  - Example 3: class decision

- Write a C function to implement these
  - How long would it take to execute?
- Design an extension instruction
  - How complex of a functional unit would be required?
Introduction to ASIPs

- Application-Specific Instruction Set Processor (ASIP)
  - A stored-memory CPU whose architecture is tailored for a particular set of applications
  - Programmability allows changes to implementation, use in several different products, high datapath utilization
  - Application-specific architecture provides smaller silicon area, higher speed
ASIP Enhancements

- Performance/cost enhancements:
  - Special-purpose registers and busses to provide the required computations without unnecessary generality
  - Special-purpose function units to perform long operations in fewer cycles
  - Special-purpose control for instructions to execute common combinations in fewer cycles
ASIP Co-Synthesis

• Given:
  - A set of characteristic applications
  - Required execution profiling

• Automatically generate:
  - Microarchitecture for ASIP core
  - Optimizing compiler targeted to the synthesized ASIP

• Implement application using core + compiler
ASIP Design Problems

- Processor synthesis
  - Choose an instruction set
  - Optimize the datapath
  - Extract the instruction set from the register-transfer design

- Compiler design
  - Drive compilation from a parametric description of the datapath and instruction set
  - Bind values to registers
  - Select instructions for code matched to parameterized architecture
  - Schedule instructions
Instruction Set Selection

- [1] – Choose instruction set based on application program set
- Assumes that datapath is given
- Inputs: datapath architecture, execution traces of benchmarks, live register analysis
- Instruction selection based on N% rule: instruction accepted only if it improves performance by N%

Instruction Selection Process

- Code is divided into segments at random
  - (Segments may contain jumps.)
- Symbolic execution turns segments into symbolic form: outputs as a function of beginning program state
- Use heuristic search to find minimal-time microoperation sequence for each symbolic state transition
- Selected instructions must cover all required operations
  - Use N% rule to evaluate coverings
Instruction Selection Process (cont.)

- [2] – View instruction set design as scheduling of microoperations (MOPs)
- Objective: \((100/N)\ln(\text{perf}) + \text{cost}\)
- Application code is divided into basic blocks
  - User weights basic blocks by importance
- Constraints on combining MOPs: instruction word width, data dependencies, timing constraints

Synthesis Procedure

• Schedules operations using simulated annealing, as constrained by data dependencies, timing of multi-cycle events, and max # opcodes
• Instruction manipulation operations:
  • unify/split two register operands
  • make a register implicit
  • make implied operands explicit
• Instruction move operations:
  • swap MOPs in time
  • move MOP in time
  • add/delete empty time step
Another Instruction Set Definition

- [3]: semi-automatically derive instruction set
- Designer provides an initial collection of datapath components and application program
- Application code is expanded onto given components. Operations are bundled into interconnected sets
- Scheduling of operations gives occupation graph
- Datapath components can be modified to improve occupation and datapath resource sharing

Architecture Template

• Designer specifies architecture template; synthesis fills in the template
• Template is specified in terms of MOPs and timing parameters
• Typical MOP specification:
  • name, R1 <- R1 + R2; format cost; hardware cost; execution stages used
• Typical timing parameters:
  • data path module, latency
Retargetable Compilation

for (i=0; i<N; i++)
c[i] = foo(a[i], b[i]);

- application code
- front end
- code generation
- object code
- instruction set definition

from ASIP core synthesis

microarchitectural model
Microarchitectural Model

- Microarchitectural model is structural
- Basic elements: registers, function units, RAM/ROM
Resource Scheduling

- Timmer et al: model all possible conflicts, then use those conflicts in scheduling
- Register transfer: path from one register to another
- Overall conflict graph (OCG) has edge between RTs if those register transfers use same resource in different modes. Add extra edges for instruction conflicts
Scheduling for Spill Minimization

• Liao et al: schedule operations to minimize number of register spills. Particularly important for accumulator architectures such as TMS32010

• Given DAG of basic block, find linear ordering of operations to minimize register spill. Solve by branch-and-bound, constructing partial schedule. Lower bounds improve efficiency:
  - outputs of basic block must be spilled
  - multiple fanouts must be spilled
  - some multiple-input instructions require spill
Template Matching

expression

+ 

- 

a b

- 

op1 op2 

op1 op2 

plus minus

instruction templates
Tree Covering

step 1

step 2
Dynamic Programming Approach

- Contiguous evaluation property: optimal evaluation of expression tree comes from evaluating subtrees into memory, then combining results
- Three-step dynamic programming algorithm (Aho, Sethi, Ullman):
  - Compute costs for each node, proceeding bottom-up; cost c[i] is optimal cost of subtree assuming that i registers are available
  - Use costs to determine which subtrees must be computed into memory
  - Traverse tree to generate code
Liem et al: generalize traditional pattern-matching code generation to handle irregular datapath structures

Patterns:

- read/write variable
- data operation
- read/write array
- control flow
Patterns and Code Generation

- Build patterns for data flow, control flow
- Arrange each in DAG for search. Descendants are supersets of ancestor patterns. Pseudo-patterns organize tree by type
- Match patterns to tree
  - Can use dynamic programming for simple cost functions
  - Need more complex matching algorithm for other cost functions
Register Allocation

- Many DSPs/ASIPs have irregular register organizations – few/no general-purpose registers
- Divide registers into classes. Register may belong to more than one class. Class may be divided into subclasses
- Initially determine candidate register sets for each data flow operation
- Assign values to registers using variation of left-edge algorithm, based on variable lifetimes
MIMOLA Approach

- Major steps in MIMOLA code generation:
  - Program transformation—choose variable layout in memory, transform loops into conditional jumps
  - Preallocation—initial assignment of hardware function units to operations
  - Code generation—pattern matching
  - Scheduling—pack microoperations into microinstructions
Instruction Set Extraction

- Circuit representation models datapath structure
- Every microoperation assigns an expression to a target storage module, creating a condition tree
- A condition tree description may be described in terms of intermediate modules—must expand each condition tree to storage nodes
- Final checks: condition refers to memory or register; conflicts among common subranges of instruction word; consistent condition
Bootstrapped Microcode Generation

- Phase 1: generate possible control
  - Generate control for each possible instruction
  - Generate microcontrol ROM for available instructions using MIMOLA
- Phase 2: generate actual control
  - Add microcontrol ROM to the microarchitectural model
  - Generate code for application, making use of microcontrol instructions