Implementing Microprocessors from Simplified Descriptions

Nikhil A. Patil, Derek Chiou

The University of Texas at Austin

Asia South Pacific Design Automation Conference
January 2013
Simplify description of processor hardware

- Processors are becoming more and more complex
- Most complexity is added *on purpose*
- Better performance and power efficiency

Make processor hardware description *simpler* without reducing the ability to specify inherent complexity
What does *simplify* mean?

simplify: to make simple; reduce to basic essentials
What does *simplify* mean?

**simplify**: to make simple; reduce to basic essentials
What does *simplify* mean?

**simplify**: to make simple; reduce to basic essentials

```markdown
structure of rec is entangled in the description
```

```verilog
wire [96:0] rec;
wire [15:0] val;

```

```verilog
typedef struct packed {
    bit align;
    bit [15:0] x;
    bit [15:0] y;
    bit [63:0] theta;
} rec_t;

rec_t rec;
wire [15:0] val;

assign val = rec.align ? rec.x : rec.y;
```
What does *simplify* mean?

**simplify**: to make simple; reduce to basic essentials

- Compiler generates packing
- Simpler to read/debug
- Easy to add new fields
- Avoid bugs accessing fields

```verilog
typedef struct packed {
  bit align;
  bit [15:0] x;
  bit [15:0] y;
  bit [63:0] theta;
} rec_t;

wire [96:0] rec;
wire [15:0] val;


rec_t rec;
wire [15:0] val;

assign val = rec.align ? rec.x : rec.y;
```
Problem: disentangling functionality

Processor *design* clearly separates correctness and performance

- Instruction set: correctness
- Microarchitecture: power, performance

But processor *implementation* intimately entangles them

Make processor hardware description simpler without reducing the ability to specify inherent complexity by disentangling functionality from the description
Impact of disentangling functionality

- Compiler generates ISA-dependent logic:
  - Microcode table (control store)
  - Structure & encoding of control words
  - Logic controlled by microcode control bits
- Description becomes simpler to write, read and modify
- Avoid instruction implementation bugs
- Easy to add new instructions
Toolflow

Bluespec

Bluespec

SystemVerilog
(BSV)

Bluespec synthesis
toolflow
Toolflow

Two descriptions

BSV Template

Instruction Set
Toolflow

Holes

BSV Template → Compiler

Instruction Set → BSV Template

Compiler
Toolflow

Flow

BSV Template → Flow → Instruction Set

Compiler
Architectural State

μL language
- sequential
- static types
- type inference
- bit-width inference
- extensible structs

RegID ← defenum [EAX, ECX, EDX, ...]

State ← defstruct
[ (Bit 32 , PC)
 , (Array RegID (Bit 32) , RF)
 , (Array Addr (Bit 8) , MEM)
 , (CCode , CC)
 , (Bool , HLT) ]
Instruction Set

μL language
- sequential
- static types
- type inference
- bit-width inference
- extensible structs

```python
def NOP(inst):
    pc ← select (PC)
    update (PC, pc + 1)

def JMP(inst):
    pc ← select (PC)
    update (PC, pc + signExt(inst.IMM))

def POP(inst):
    pc ← select (PC)
    update (PC, pc + 2)
    sp ← read (RF, ESP)
    x ← read4 (MEM, sp)
    write (RF, ESP, sp + 4)
    write (RF, inst.DEST, x)
```
Instruction Set

It is easy to describe “functionality” without reference to “timing”

Limitation: we cannot directly specify
- memory model
- non-deterministic instructions
Microarchitectural Template: holes

It is difficult to describe microarchitecture without reference to ISA

- Details of the ISA manifest “all over the place”
- User can use holes to skip such details by using holes in BSV
- Compiler tries to fill in holes using ISA information

▷ A. Solar-Lezama et al, PLDI 2005

*Programming by Sketching for Bitstreaming Programs*
Holes: used in teaching

University of Texas at Austin

- EE 460N: Undergraduate Computer Architecture
- Simulate a 5-stage pipeline for LC3b ISA
- Given a detailed document describing microarchitecture
- Students fill in empty LOGIC boxes
LC3b pipeline: memory stage
Fig. 5 Memory Stage (MEM−Stage)
How big are the holes?

carefree user

- make the entire processor a single hole
- insert pipeline registers leaving each stage as a hole
- refine the fetch stage to use an instruction-cache
- ...

describe detailed hardware with several small holes all over

power user → many, small, combinational holes
Holes in BSV

- Manually specified as a pure function: `#hole(⋯)`
- Explicit inputs, but not necessarily minimal
- Automatically defined using ISA information
- Synthesized to combinational logic

**Example**

```plaintext
dcache.request(#ldst_p(uop),
    phyAddr(#addr(uop)),
    #st_data(uop));
```

- ▶ load/store
- ▶ address
- ▶ store-data
Filled-in BSV is a valid microarchitecture for this ISA

- Makes hole synthesis at least as “hard” as verification
- Too strong a notion for correctness
- How can we weaken this?
Flow: an intermediate spec

Flow describes the **functional** execution of a single instruction through the microarchitecture template.
Flow: example

μL language
- sequential
- static types
- type inference
- bit-width inference
- extensible structs

while (true):
    pc ← select (PC)
    inst₀ ← fetch (pc)
    update (PC, pc + #ilen(inst₀))
    x ← read (RF, #src(inst₀))
    inst₁ ← inst₀ +: (DATA, x)
    z ← #add_x(inst₁) + #add_y(inst₁)
    inst₂ ← inst₁ +: (RESULT, z)
    pupdate (PC, #jmp(inst₂), #target(inst₂))
pwrite (RF, #wr_en(inst₂), #dest(inst₀), z)
Flow: divide problem into two

**User**
BSV template must “implement” flow for every type-correct hole definition

**Compiler**
Flow must be equivalent to ISA for the generated hole definition
Flow restricts holes

Holes are general enough to disentangle the instruction set

But, a hole cannot contain

- intrinsically time-dependent logic
- pipeline control logic
- cache control logic
- inter-instruction dependency logic
Compiler overview

- Generate hole definitions such that flow is equivalent to ISA
- Equivalence determined by a term rewriting system $\mathcal{R}$
Compiler problem formulation

\[ \exists \theta. \forall \Sigma. \theta(\text{flow}(\Sigma)) \equiv \text{isa}(\Sigma) \]

Second-order \( R \)-matching
Problem formulation

\( R \)-matching & Lazy Narrowing

- Problem formulated as second-order \( R \)-matching
- Narrowing is a systematic way to solve such problems
- Lazy narrowing is a refinement that works outside-in
- Heuristic-guided systematic search, backtracking as necessary

Practical concerns

- Solver can be very slow and timeout
- Need limits on hole size
- Equation solver errors are cryptic
Evaluation target

Y86-inorder

- CMU, Introduction to Computer Systems
- 27 instructions (including variants)
- 5-stage inorder pipeline
### Y86-inorder: Compile times

<table>
<thead>
<tr>
<th>Compiler stage</th>
<th>Time</th>
<th>Instruction</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Typecheck</td>
<td>0.2 s</td>
<td>RRMMOV</td>
<td>0.42 s</td>
</tr>
<tr>
<td>Normalize</td>
<td>1.5 s</td>
<td>RMMOV</td>
<td>0.33 s</td>
</tr>
<tr>
<td>Solver</td>
<td>4.9 s</td>
<td>MRMOV</td>
<td>0.31 s</td>
</tr>
<tr>
<td>Microcode</td>
<td>0.1 s</td>
<td>ADD</td>
<td>0.32 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>JXX</td>
<td>0.16 s</td>
</tr>
<tr>
<td>Total</td>
<td>6.7 s</td>
<td>PUSH</td>
<td>0.29 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>POP</td>
<td>0.39 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CALL</td>
<td>0.24 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RET</td>
<td>0.43 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LEAVE</td>
<td>0.35 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
## Y86-inorder: Lines of code

<table>
<thead>
<tr>
<th></th>
<th>Vanilla BSV flow</th>
<th>Input to compiler</th>
<th>Output of compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Full processor</td>
<td>Core pipeline</td>
<td>Microcode</td>
</tr>
<tr>
<td></td>
<td>3300 lines BSV</td>
<td>400 lines BSV</td>
<td>20 lines BSV</td>
</tr>
<tr>
<td></td>
<td>Core pipeline</td>
<td>ISA 180 lines µL</td>
<td>510 bits BSV</td>
</tr>
<tr>
<td></td>
<td>450 lines BSV</td>
<td>Flow 63 lines µL</td>
<td>300 lines BSV</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Functions 730 gates BSV</td>
</tr>
</tbody>
</table>
Summary

BSV Template

Flow

Instruction Set

Compiler

Implementing Microprocessors from Simplified Descriptions
Thank you!