Tagged Unions and Pattern Matching

(a proposed System Verilog extension)

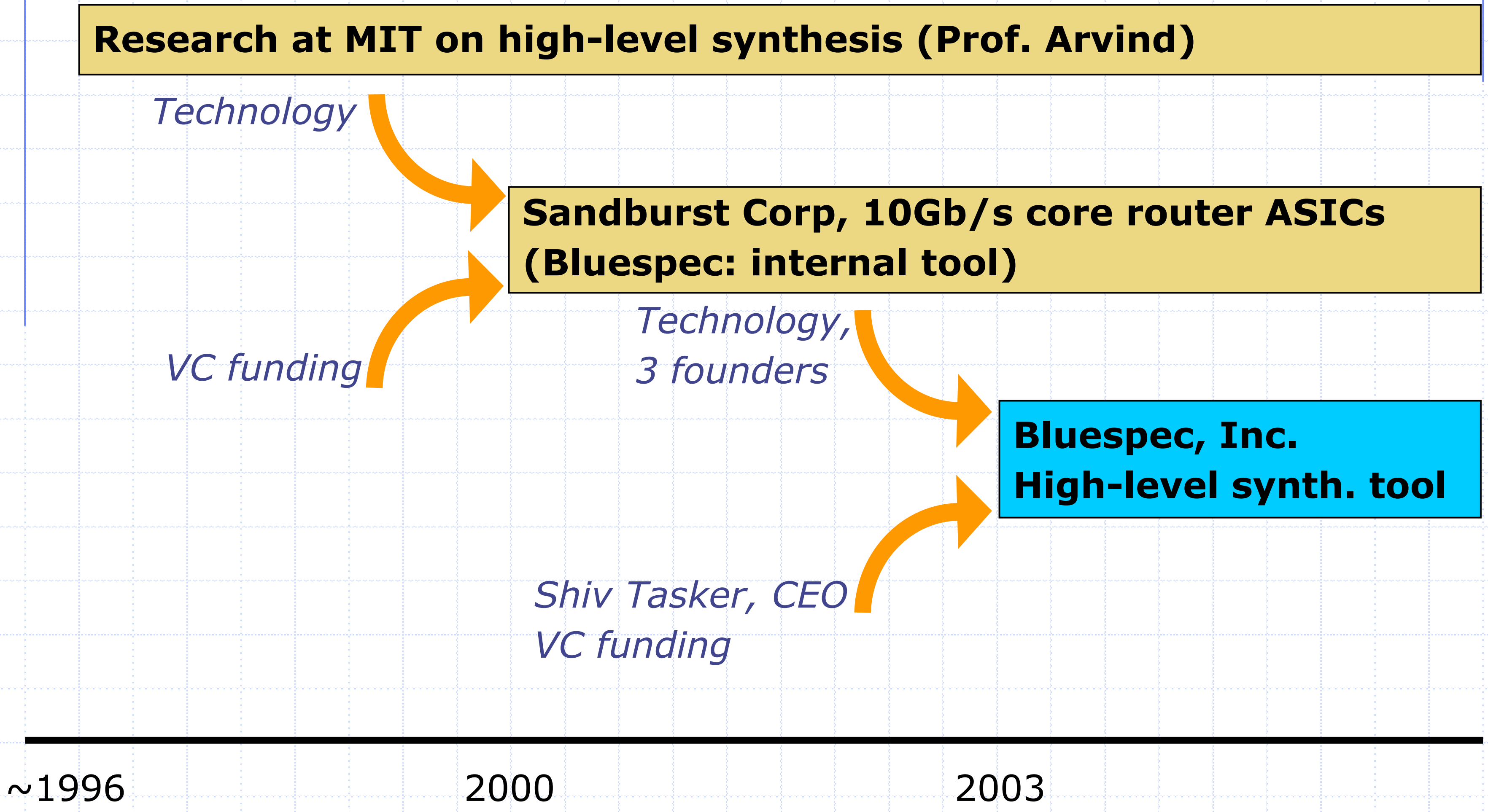
Rishiyur S. Nikhil, CTO
Bluespec, Inc.

September 18, 2003

My plan

Bluespec, Inc. (who?)	1 slide
Context of proposal	3 slides
The proposal	8 slides

Bluespec, Inc.: who?


Research at MIT on high-level synthesis (Prof. Arvind)
Technology
Sandburst Corp, 10Gb/s core router ASICs (Bluespec: internal tool)
VC funding
Technology, 3 founders
Bluespec, Inc. High-level synth. tool
Shiv Tasker, CEO VC funding
~1996
2000
2003

Context of proposal

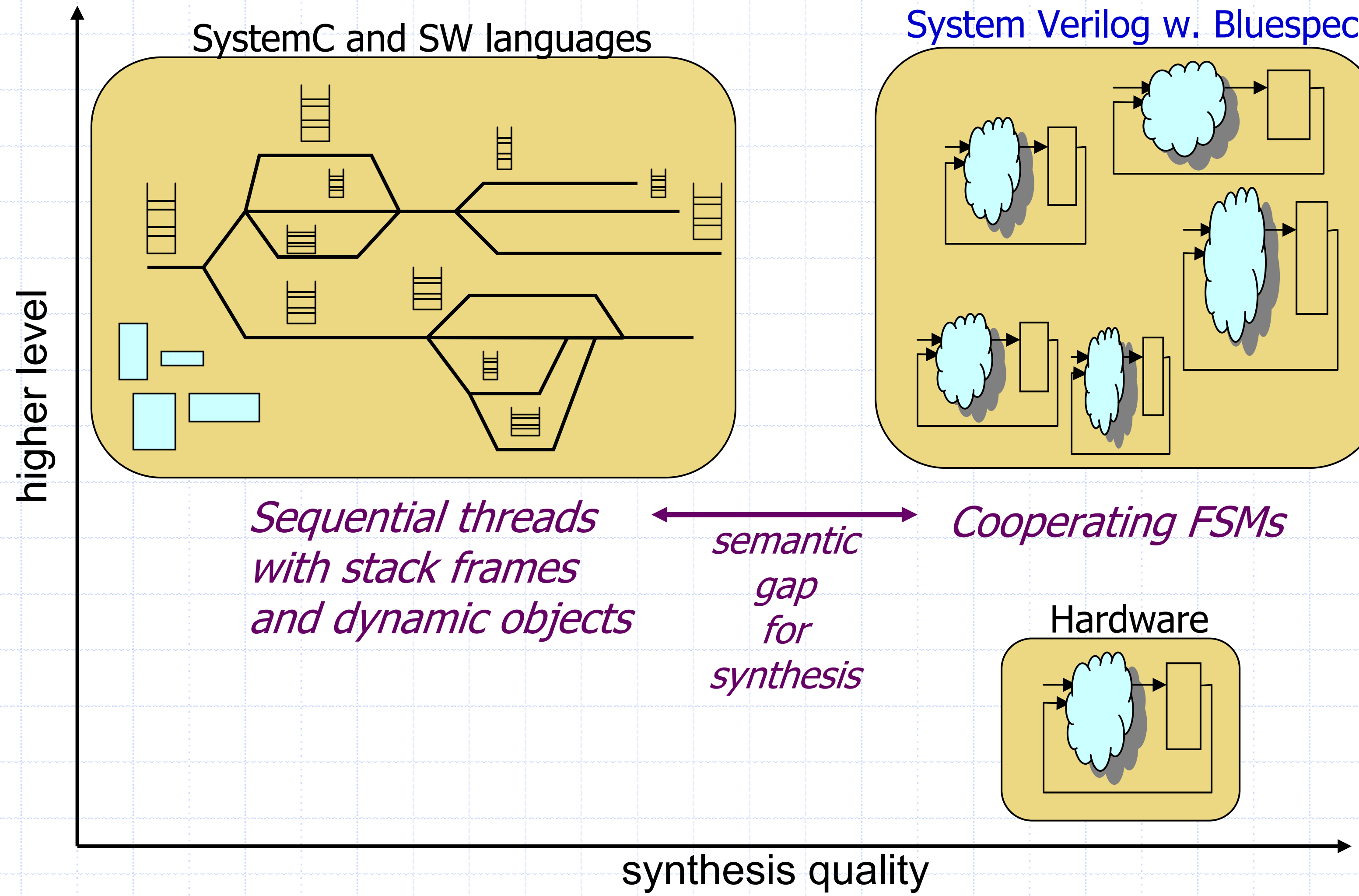
- Bluespec, a technique for high-level *synthesis,* has been developed for > 3 years.
- In an apples-to-apples comparison with a *product ASIC* (180nM, 200 MHz, 1.5Mgates) originally coded in Verilog, we've demonstrated:
  - 5x-13x reduction in source code (66K Lines of Verilog)
  - 66% reduction in verification bugs
  - Matched performance (clock speed, area)
  - Enabled major design space explorations within time budgets

Context of proposal (contd.)

- We want to align with System Verilog
- We'd like to contribute Bluespec language ideas to System Verilog
- Current proposal (*Tagged Unions and Pattern Matching*) is the first contribution

- We have more potential contributions

Why System Verilog?


SystemC and SW languages
System Verilog w. Bluespec
higher level
Sequential threads with stack frames and dynamic objects
semantic gap for synthesis
Cooperating FSMs
Hardware
synthesis quality

Proposal: background

- structs and unions are often nested. Example:

A 32b *instruction* is

either an *Add* instruction

with two sources *reg1* & *reg2*

and a destination *regd*

struct

or a *Jump* instruction, which is

union

either an *Unconditional* jump

with an immediate addr

scalar

nested union

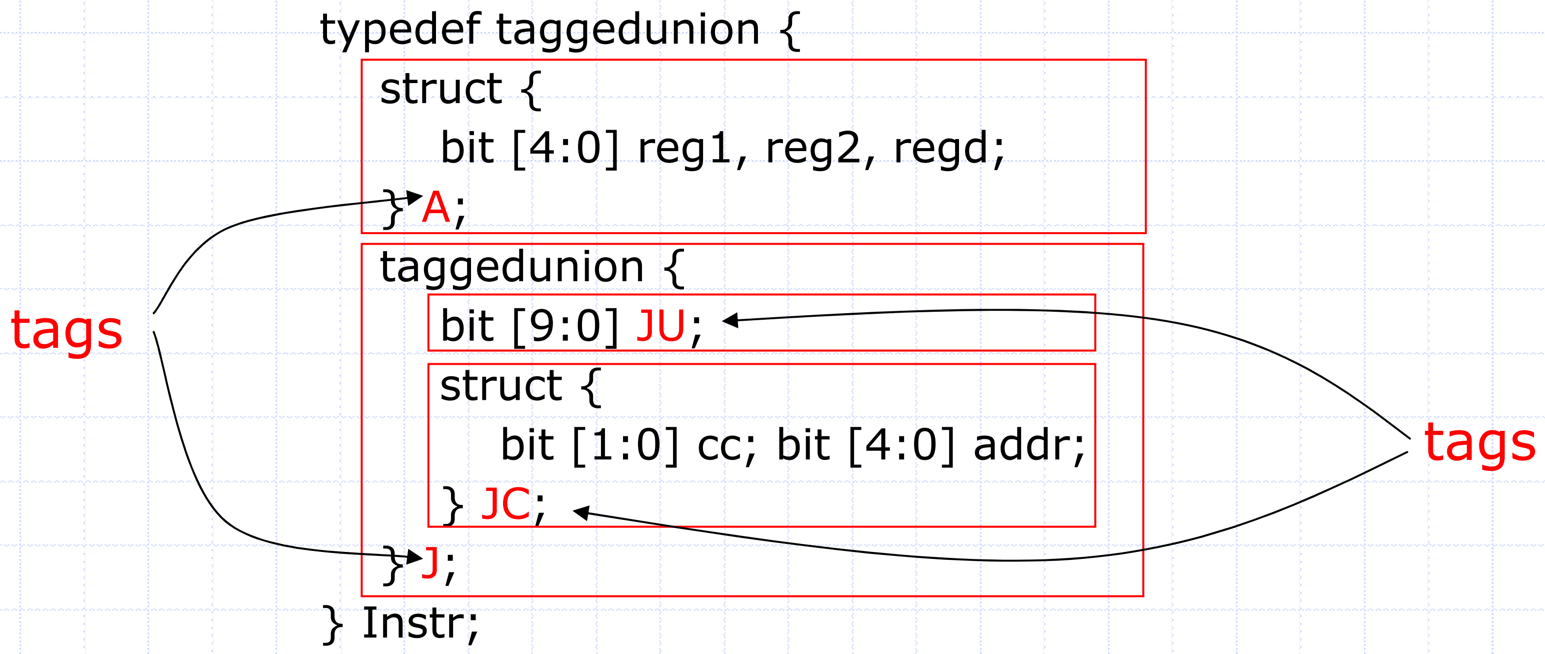
or a *Conditional* jump

with a condition-code *cc*

and offset *addr*

struct

Using tagged unions



```
typedef taggedunion {
    struct {
        bit [4:0] reg1, reg2, regd;
    } A;
    taggedunion {
        bit [9:0] JU;
        struct {
            bit [1:0] cc; bit [4:0] addr;
        } JC;
    } J;
} Instr;
```

Pattern matching

- Example usage:

```
case (instr)
 A{r1,r2,rd}: rf [rd] = rf [r1] + rf [r2];
 J{j}:          case (j)
                    JU{a}: pc+= a;
                    JC{cc,ra}: if (cf [cc]) pc = rf [ra];
endcase
```

- or (nested patterns)

```
case (instr)
 A{r1,r2,rd}:   rf [rd] = rf [r1] + rf [r2];
 J{JU{a}}:      pc+= a;
 J{JC{cc,ra}}: if (cf [cc]) pc = rf [ra];
endcase
```

Other aspects of the proposal (details in the document)

- Tagged union expressions: to directly construct a tagged union *value*
  - in any expression context
  - look just like patterns
- Pattern matching in if statements
- Canonical bit representations
  - zero implementation overhead (compared to coding with unions and structs)
- Arbitrary bit representations, with automated packing/unpacking

Compare w. unions/structs

```
typedef struct {
    Opcode op;                  // A or J
    union {
        struct {
            bit [4:0] reg1, reg2, regd;
        } A_operands;
        struct {
            JumpOpcode jop;       // JC or JU
            union {
                bit [9:0] JU_operand;
                struct {
                    bit [1:0] cc; bit [4:0] addr;
                } JC_operands;
            } J_suboperands;
        } J_operands;
    } operands;
} Instr;
```

Using unions/structs

- Example usage:

```
case (instr.op)
 A: rf [instr.operands.A_operands.regd] =
          rf [instr.operands.A_operands.reg1] +
          rf [instr.operands.A_operands.reg2];
 J: case (instr.operands.J_operands.jop)
     JU: pc+= instr.operands.J_operands.J_suboperands.JU_operand;
     JC: if (cf [instr.operands.J_operands.J_suboperands.JC_operands.cc])
            pc = rf [instr.operands.J_operands.J_suboperands.JC_operands.addr;
endcase
```

Note: such deep “dot-selections” are often encapsulated in macros (`define/#define)

unions/structs: issues

- Not type-safe
  - So, adds a verification obligation
    - e.g., prove that the *regd* field is never accessed in a *Jump* instruction
- Not concise
  - too many intermediate names
- Not too readable
  - deeply nested dot-selections

Tagged unions and Pattern matching: Bottom line

- Type-safe (improves verification)
- Concise
- Readable (patterns)
- Small extension to BNF
- Synthesizable
- Zero implementation overhead
- Language concepts well tested for ~3 decades
- Synthesis well tested for ~ 3 years

We have more potential contributions

- *parametric polymorphism, higher-order functions, atomic state transitions, …*