Much Ado about Almost Nothing: Compilation for Nanocontrollers

LCPC2003

11:40-12:00, Saturday, October 4, 2003

Hank Dietz, Shashi Arcot, and Sujana Gorantla
Electrical and Computer Engineering Department
University of Kentucky
Lexington, KY 40506-0046
`http://aggregate.org/hankd/`

Why Nanocontrollers?

- Nanofabrication techniques allow:
  - Features as small as ˜30 nanometers
  - Micrometer-scale devices: sensors, actuators, etc.
  - Low-temperature processing (can build *over* circuitry)
- How can we intelligently control thousands to millions of micrometer-scale devices on a single chip?
  - LOTS of signals with off-chip processing?
  - LOTS of signals multiplexed by an on-chip processor?
  - A massively-parallel computer on a chip:
    LOTS of tiny nanocontrollers, one per device!

What Must A Nanocontroller Be Able To Do?

- Minimal Circuit Size
- Predictable Real-Time Behavior
- Localized Input/Output
- Coordination as a Parallel Computer
- Each nanocontroller independently programmable (MIMD)
- Reprogramability

Nanoprocessor/Nanocontroller Architecture

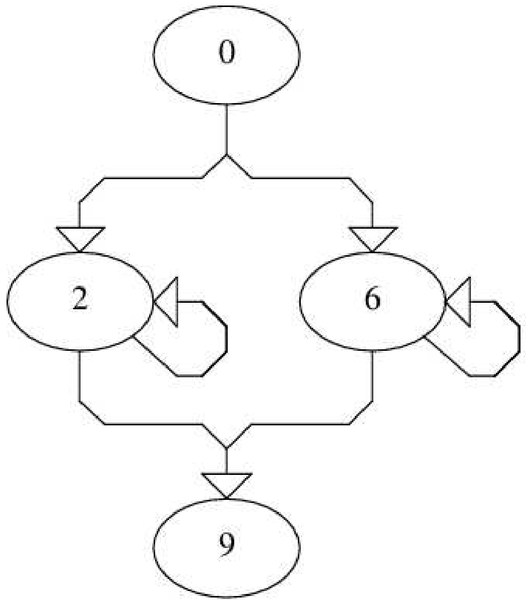
- No previous architectural model fits:
  - Radical new architectures aren’t sufficiently developed
  - MIMD is close, but circuit complexity is too high... especially for program memory
  - SIMD is very close...
    but PEs are not independently programmable
  - SIMD executing MIMD code by interpretation also needs too much local memory for programs
- *Kentucky Architecture* is SIMD-like hardware with compiler technology that transforms MIMD code into pure SIMD code so *control flow* is entirely replaced by *selection*

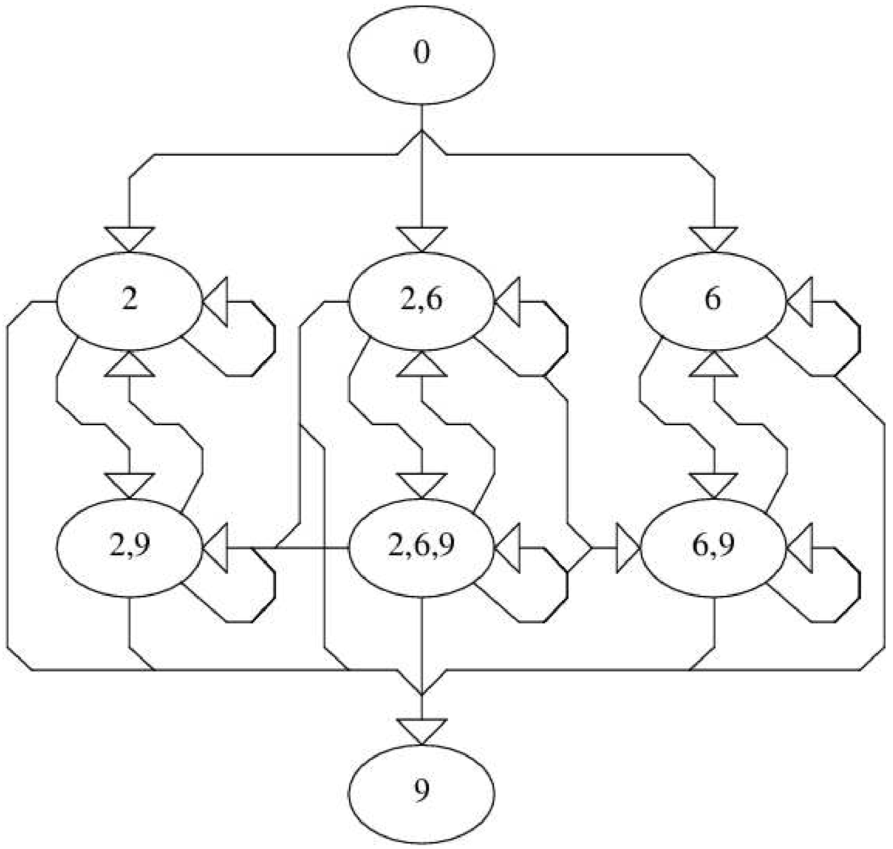
Meta-State Conversion (MSC)

- Developed in 1992-1993 for MasPar MP-1 distributed-memory SIMD to execute shared-memory MIMD code
- State-space transformation, like NFA-to-DFA conversion; in practice, number of states grows slowly
- A SIMD Meta State corresponds to each *set of possibly temporally co-existant MIMD node states*
- Code within a Meta State is executed with processors enabled only for the code they should execute
- **Common Subexpression Induction (CSI)** is used to create common instruction sequences, thereby minimizing disable time for each processing element

A Simple MSC Example

```
if (A) { do {B} while (C); else do {D} while (E); } F
```

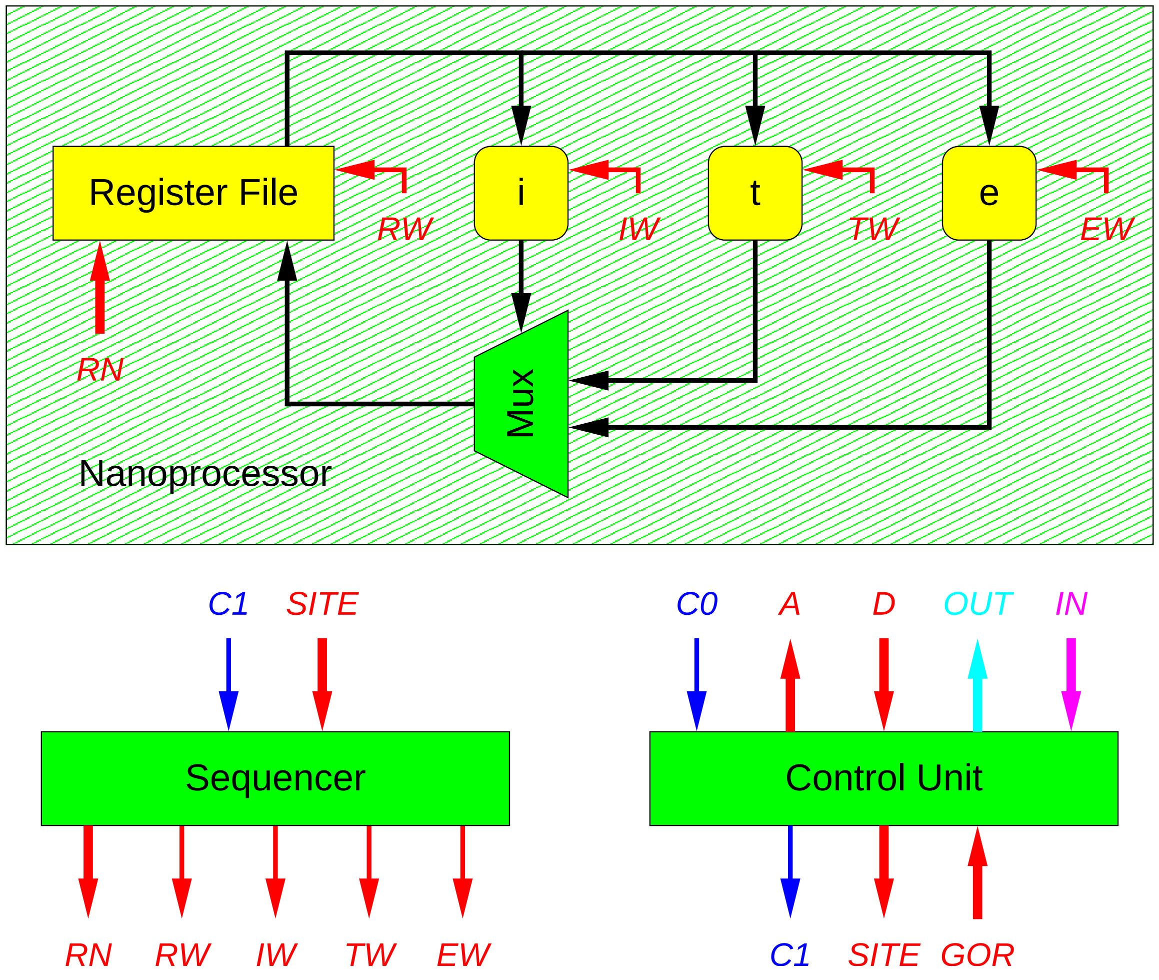




The Current Paper's Approach

- Implement nanocontrollers using:
  - Simplifi ed Kentucky Architecture design, **KITE**, which reduces ALU to a single 1-of-2 Multiplexor
  - Compilation of word-level operations directly into bit-level *If-Then-Else (ITE)* operations optimized by techniques borrowed from logic optimization
  - Greatly simplifi ed MSC and CSI due to the fact that ITEs directly implement enable masking
- Disclaimers: no KITE hardware has been built and the compiler system described is a research proof-of-concept

KITE: Kentucky If-Then-Else



KITE: Kentucky If-Then-Else

- Only instruction is *SITE: Store-If-Then-Else*
- Control unit like VLIW multiway branch unit, not SIMD CU:
  - No scalar instructions in CU
  - Fancy management for fetch and caching of compressed basic blocks of instructions tagged with multiway exit arcs
  - Clock rate determined by (possibly off-chip) instruction memory
- Sequencers fed SITEs at intermediate clock rate, locally broadcast control signals at full clock rate
- Nanoprocessor/Nanocontroller runs at full clock rate, 4 cycles/SITE

Programming Language: BitC

- A very small C dialect
- Minor extensions to C data types:
  - Explicit precision; e.g., `int:3 a;`
  - I/O and communications; e.g., `int:1 adc@5;`
- All applicable C operators plus a few others:
  `?<` (min), `?>` (max), `$` (population count), etc.
- The usual control flow, but no recursive functions

Transforming Word-Level To Bit-Level

- BitC code:

```
unsigned int:2 a, b, c;
c = a + b;
```

- Function on arbitrary-precision values:
  (in fact, a 3-bit result is computed and top bit is ignored)

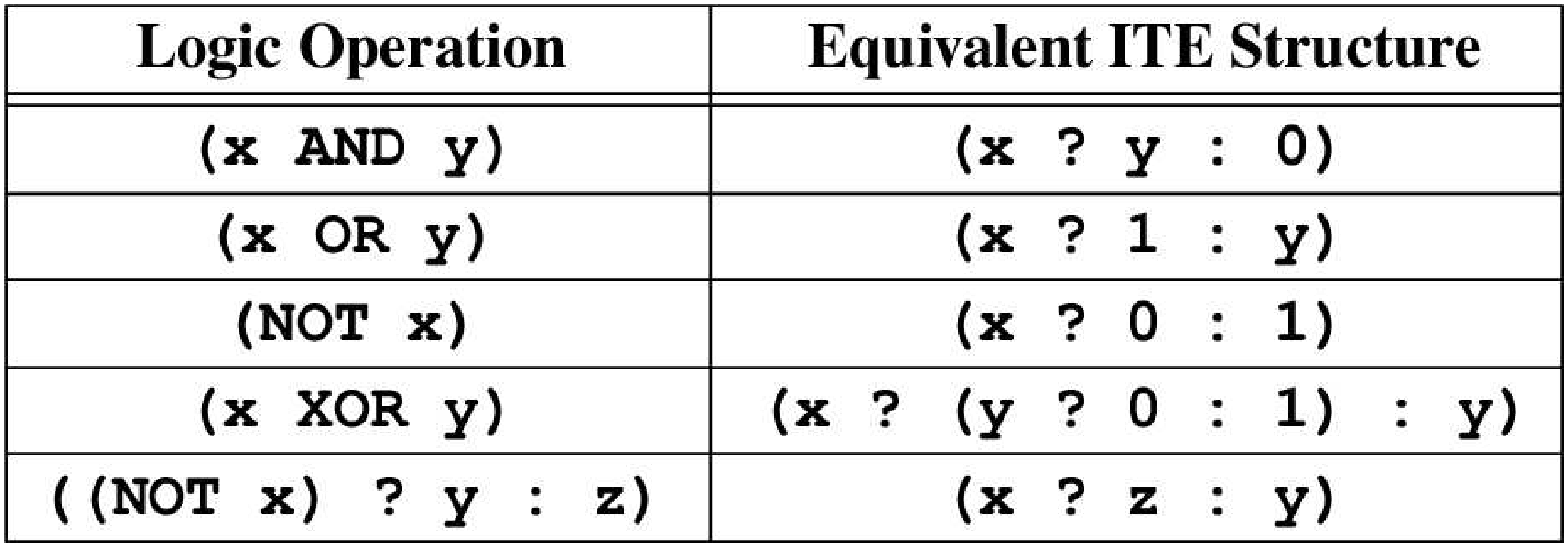
```
{c1, c0} = {a1, a0} + {b1, b0}
```

- Bitwise logic expressions:

```
c0 = (a0 XOR b0)
c1 = ((a1 XOR b1) XOR (a0 AND b0))
```

ITE Equivalents For Familiar Logic Operations

- Like NAND, ITEs are complete
- XORs are not effi ciently represented using ITEs



Logic Operation	Equivalent ITE Structure
`(x AND y)`	`(x ? y : 0)`
`(x OR y)`	`(x ? 1 : y)`
`(NOT x)`	`(x ? 0 : 1)`
`(x XOR y)`	`(x ? (y ? 0 : 1) : y)`
`((NOT x) ? y : z)`	`(x ? z : y)`

Transformation Into ITEs

- Bitwise logic expressions:

```
c0 = (a0 XOR b0)
c1 = ((a1 XOR b1) XOR (a0 AND b0))
```

- ITE equivalents:

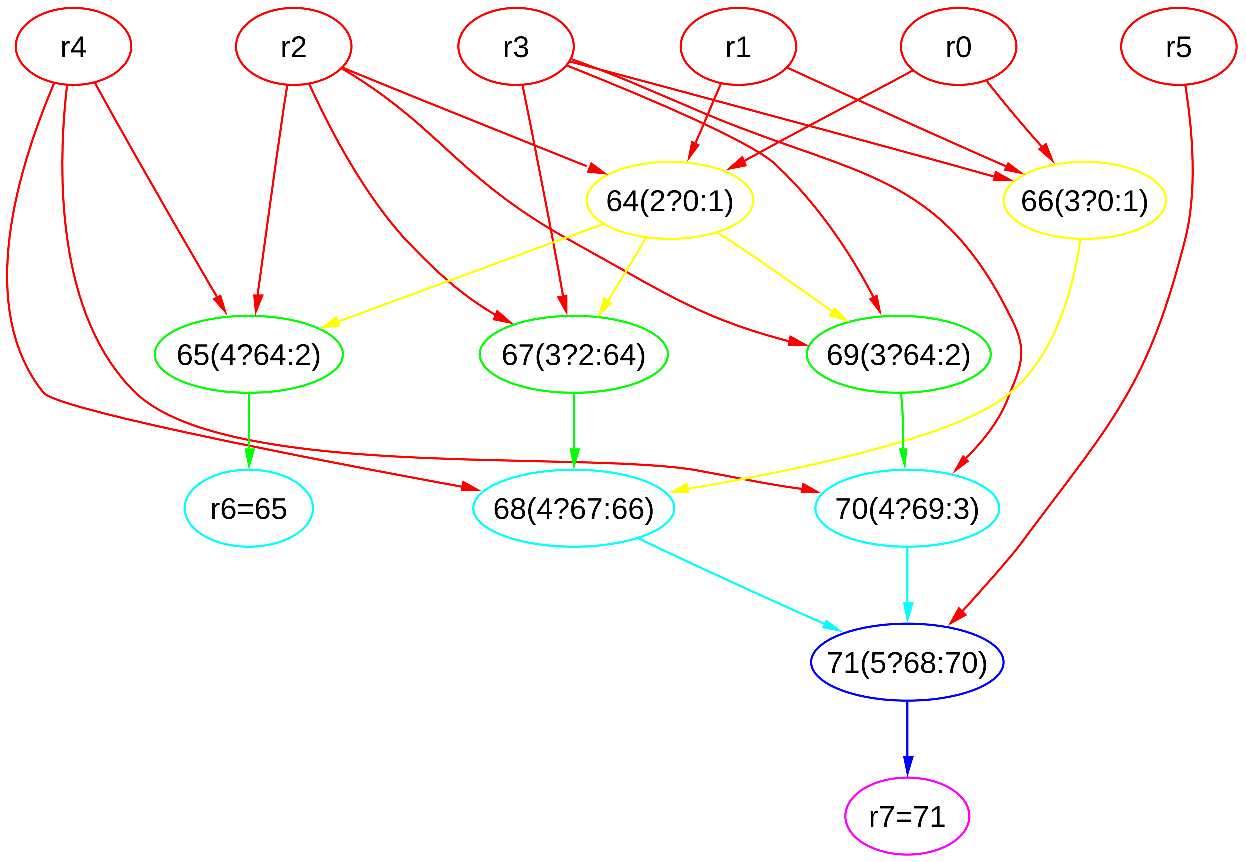
```
c0 = (a0 ? (b0 ? 0 : 1) : b0)
c1 = ((a1 ? (b1 ? 0 : 1) : b1) ?
      ((a0 ? b0 : 0) ? 0 : 1) : (a0 ? b0 : 0))
```

- But those aren’t the ITEs we actually generate...

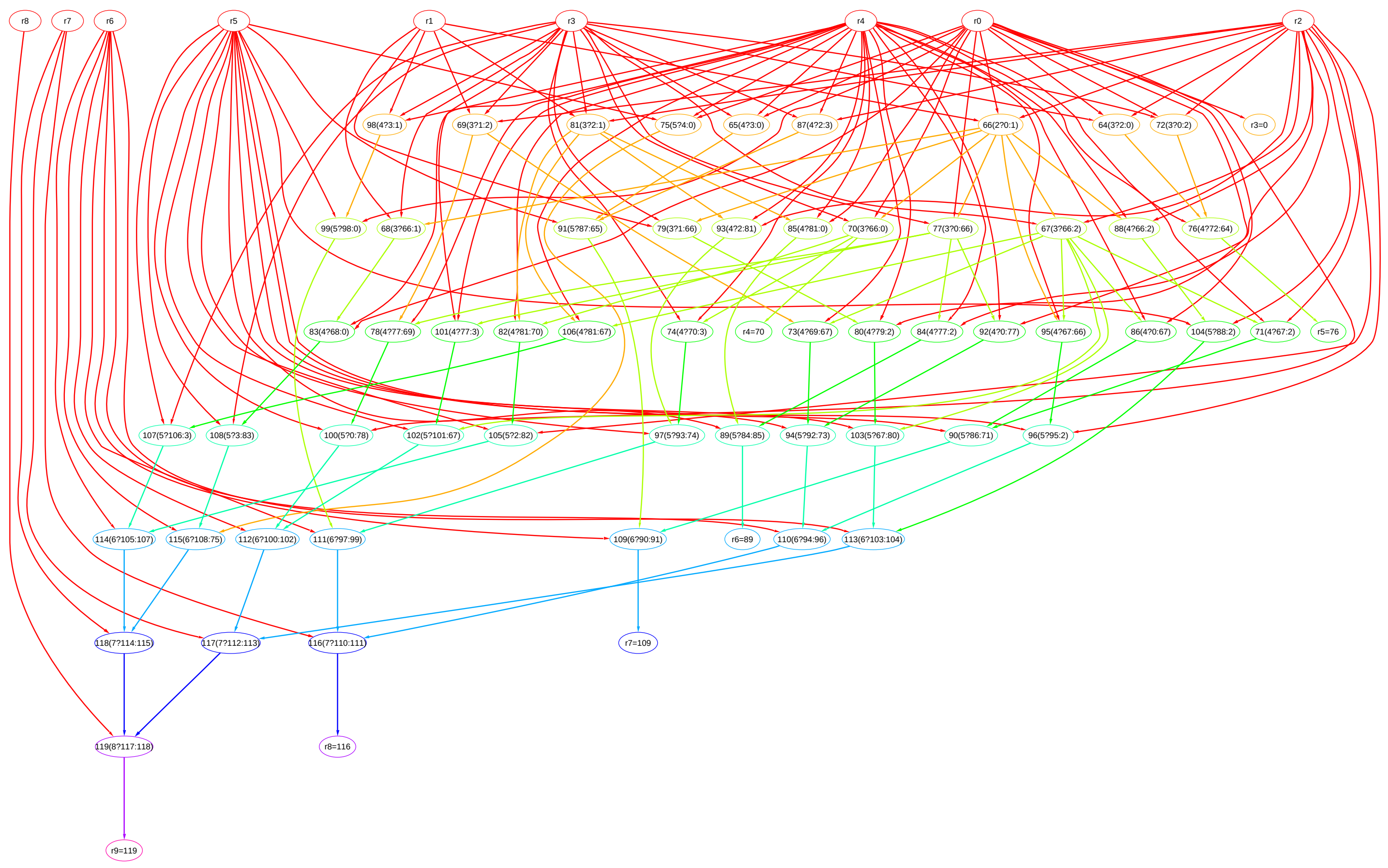
Optimization Of ITEs

- Can use *Multi-level Multi-value Logic Minimization*
- Normal form is identical for equivalent circuits
- Bryant Normal Form for BDDs of the form (a ? b : c):
  - Require a is an input, lexically before inputs in b and c
- Karplus improvements to Bryant’s normalization:
  - More direct production of normal form...
- Karplus Normal Form:
  - Allow NOT of a, b, or c (to make form more compact)
  - Require inputs in a, b, and c to be lexically ordered

Bryant Normal Form



A Larger Example: `int:8 a; a=a*a;`



Predication (Selection) Is Trivial Using ITEs

- Consider:

```
if (a) { b=c; if (d) e=f; else g=h; i=j; } k=l;
```

- By simple if-conversion, we get:

```
b = (a ? c : b);
e = ((a ? d : 0) ? f : e);
g = ((a ? (d ? 0 : 1) : 0) ? h : g);  /* typo in paper swaps h, g */
i = (a ? j : i);
k = l;
```

- This is precisely what MSC+CSI guards look like!

Preliminary Results

- Compiler speed is not a problem
- Complexity of high-precision XOR-based arithmetic: Bryant’s form for `int:12 a,b; a=a*b;` has 156,392 ITEs; use multiple-state sequence for high precision?
- Normal form *perfectly* recognizes word-level identities, although it wasn’t told about any of them... e.g., `int a,b; a=a+b; a=a-b;` generates no ITEs!
- Normal form could be used to disambiguate indirect references...

Conclusion

- ITE-based multi-level multi-value logic minimization can be used to optimize compiler’s bit-level coding of word-level operations, while dramatically simplifying MSC+CSI
- SITE-based KITE architecture viable as a nanocontroller; circuit complexity potentially ˜100 transistors/processor
- ITE normal forms have benefi cial side-effects in recognizing equivalence of expressions
- Yet to be done: register allocation, instruction block encoding, implementation of KITE system