Xtensa
A new ISA and Approach

Tensilica: www.tensilica.com

Earl Killian: www.killian.com/earl

Presentation Goals

- How Tensilica and Xtensa came to be
- What Xtensa is, with motivation for the decisions we made
  - Historical approach
- Get you thinking about a new paradigm
  - How do application-specific processors change the game?

- What are you interested in hearing about?

My Background

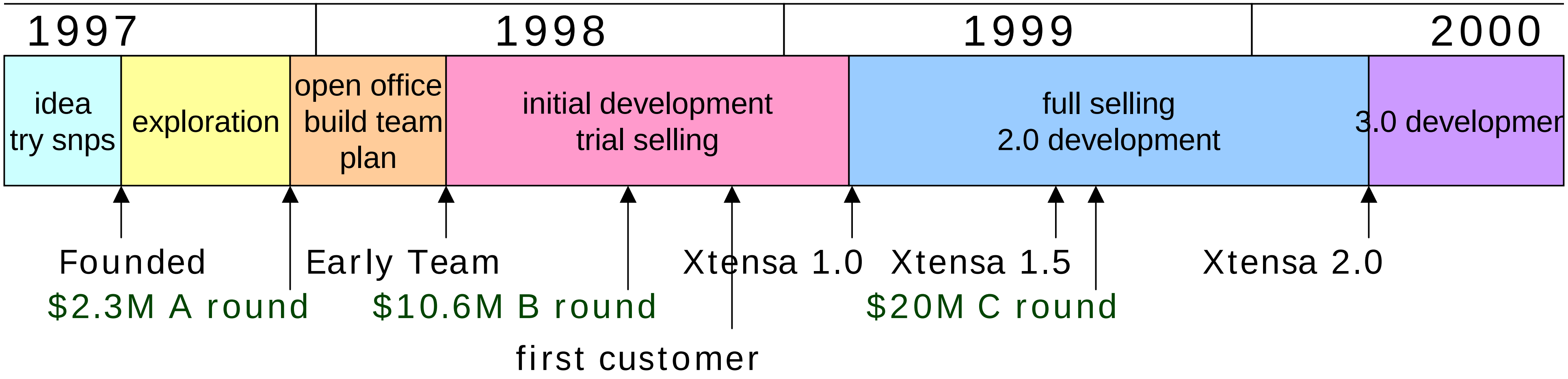
- Major Projects
  - 2 operating systems (not Unix)
  - 3 compilers (not gcc)
  - 1 satellite network
  - 4 processor instruction set designs
  - 6 processor micro-architectures
- Places
  - 1 University
  - 3 Start-ups (founder of one)
  - 1 Government lab
  - 2 Medium-sized companies

Outline

- About Tensilica
  - History, getting started, etc.
- Application-Specific Processors
  - What's different
- Xtensa ISA
  - What we did and why
- Extensibility via the TIE (Tensilica Instruction Extension) Language

Tensilica Background

- Tensilica is the brainchild of Chris Rowen
  - founder and CEO
  - formerly Intel, Stanford, MIPS, sgi, and Synopsys
  - an idea that wouldn't leave him alone: configurable processors



1997 1998 1999 2000

idea
try snps

exploration

open office
build team
plan

initial development
trial selling

full selling
2.0 development

3.0 developmer

Founded
$2.3M A round

Early Team
$10.6M B round

first customer

Xtensa 1.0

Xtensa 1.5
$20M C round

Xtensa 2.0

Outline

- About Tensilica
  - History, getting started, etc.
- Application-Specific Processors
  - What’s different
- Xtensa ISA
  - What we did and why
- Extensibility via the TIE (Tensilica Instruction Extension) Language

Tensilica's Mission

- From an early corporate overview:

  To be the leading provider of

  application-specific microprocessor solutions

  by delivering

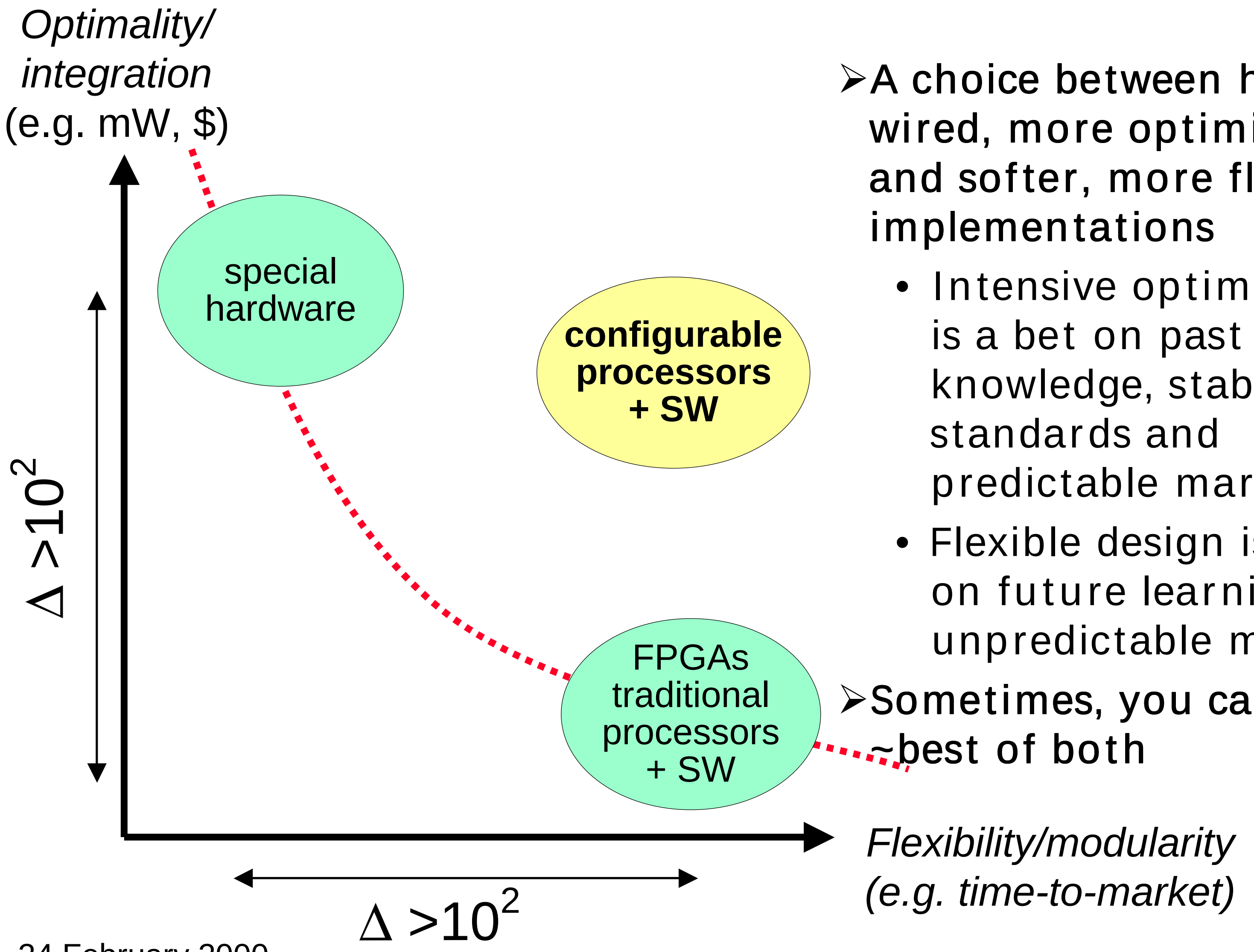
  configurable, ASIC-based cores

  and

  matching software development tools

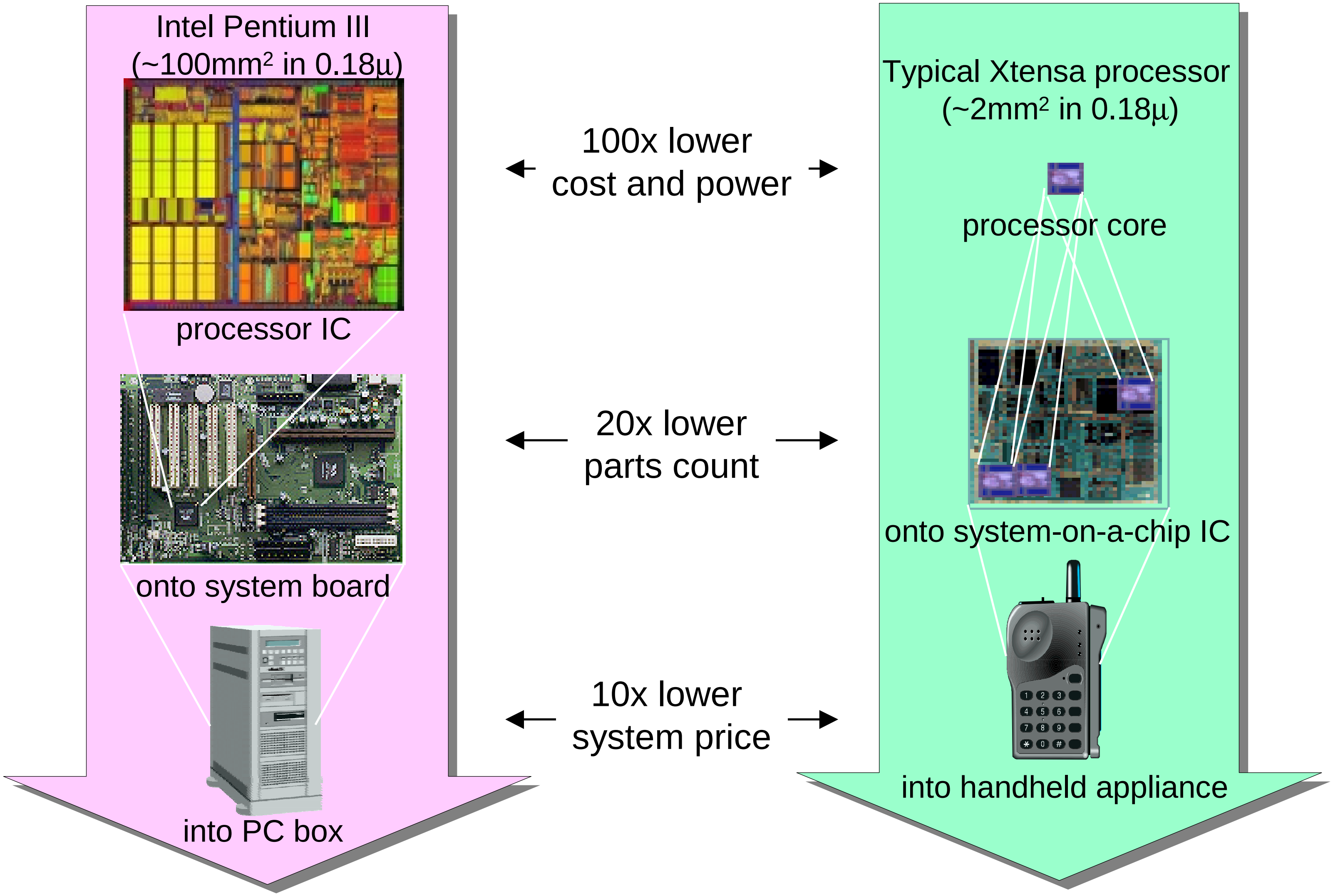
- Therefore
  - Synthesizable, configurable, embedded processors
    - Application is known at ASIC-design time!
    - Key is to exploit application specificity
  - Compiler and OS are as important as the processor
  - Customers are system designers
    - Very cost conscious customers — will only pay for what they need

The Opportunity



- A choice between hard-wired, more optimized and softer, more flexible implementations
  - Intensive optimization is a bet on past knowledge, stable standards and predictable markets
  - Flexible design is a bet on future learning and unpredictable markets
- Sometimes, you can get ~best of both

Not the Desktop Model



Intel Pentium III (~100mm^2 in 0.18µ)

processor IC

onto system board

into PC box

100x lower cost and power

20x lower parts count

10x lower system price

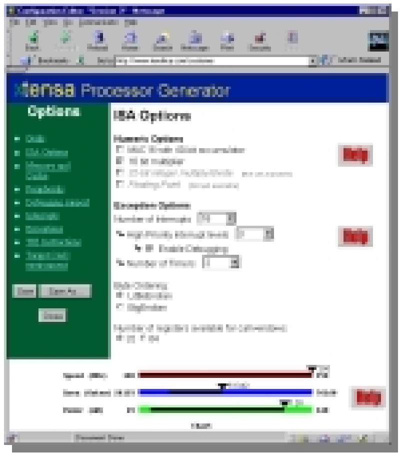
Typical Xtensa processor (~2mm^2 in 0.18µ)

processor core

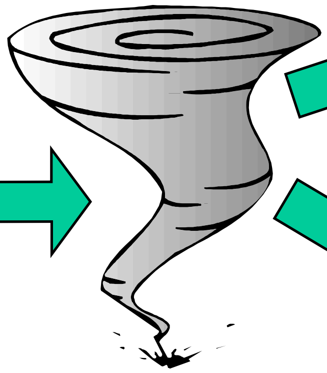
onto system-on-a-chip IC

into handheld appliance

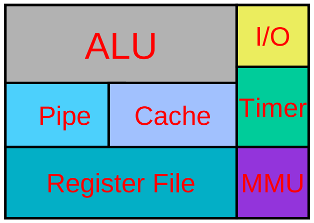
Technology Vision



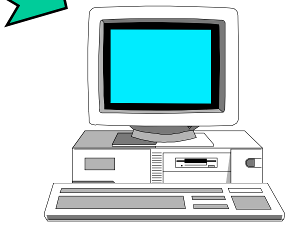
Select processor options and describe new instructions in Web interface



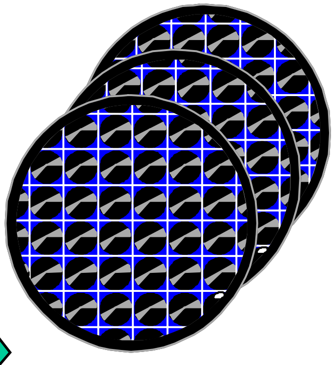
Using the Xtensa processor generator, create...

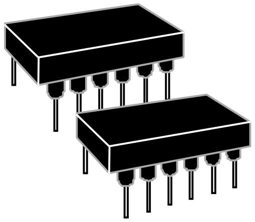


Tailored, HDL uP core



Customized Compiler, Assembler, Linker, Debugger, Simulator





Use standard library to target to the silicon

Types of Configurability

- Quantity, size, etc.
  - Often significant payback (e.g. cache size)
- Options (sort of quantity 0 or 1)
  - e.g. FP or not, MMU or not, DSP or not, …
- Parameters
  - e.g. addresses of vectors, memories, …
- Target specifications
  - e.g. synthesize for area at the cost of speed
  - Many applications don’t need the maximum processor performance
  - Process, standard cell library, etc.
- Extensibility
  - Adding things that the component supplier didn’t explicitly offer

Sample Xtensa Configurability

- Cost, Power, Performance
- ISA
  - Endianness
  - MUL16/MAC16
  - Various miscellaneous instructions
- Interrupts
  - Number of interrupts
  - Type of interrupts
  - Number of interrupt levels
  - Number of timers and their interrupt levels
  - more...
- Memories
  - 32 or 64 entry regfile
  - 32, 64, or 128b bus widths
  - Inst Cache
    - 1KB to 16KB
    - 16, 32, or 64B line size
  - Data Cache/RAM
    - ditto
  - 4-32-entry write buffer
- Debugging
  - No. inst addr breakpoints
  - No. data addr breakpoints
  - JTAG debugging
  - Trace port

Example .25µ Results

- 55 to 141MHz
- 28 to 84K gates
- 62 to 191mW power
- 2.0mm^2 to 8.3mm^2 including cache RAMs

Outline

- About Tensilica
  - History, getting started, etc.
- Application-Specific Processors
  - What's different
- Xtensa ISA
  - What we did and why
- Extensibility via the TIE (Tensilica Instruction Extension) Language

Early Planning

- Product/ISA discussion started ≈3/1998
  - Do our own ISA or MIPS/ARM?
  - What do we optimize for (performance, cost, code size, etc.)?
  - How low-end do we go (e.g. 16-bit)?
  - If our own ISA, do we need an “on-ramp”?
  - How much DSP?
- Issues
  - Only 8 months planned to do first product!
  - Legal issues using another’s ISA
  - Many standard processor tricks unavailable in synthesizable logic

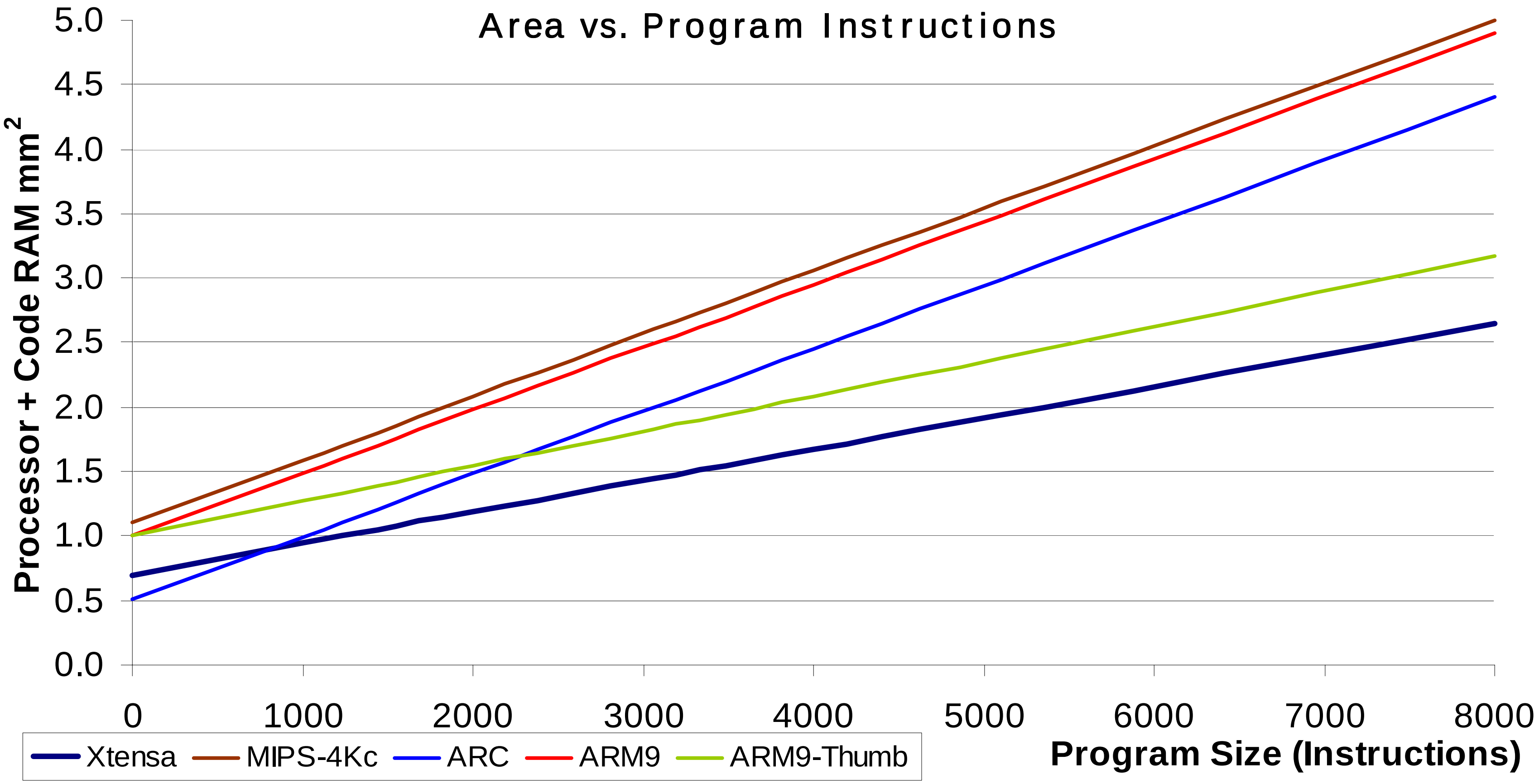
Our Guess at Our Customers' Priorities

- Solution
- System (not processor) cost
  - processor die area
  - code size
  - power
- Time-to-market
  - ease of use
  - verification
  - debugging
- Energy efficiency
- Performance
- Compatibility

Our Resulting ISA Priorities

- Code size
  - largest factor in system cost
- Configurability, Extensibility
  - provides best match to customer requirements, and so optimizes system cost
- Processor cost
  - a small factor in system cost
- Energy efficiency
  - minor influence on ISA, but listed for when it matters
- Performance
  - when all else is equal, this becomes important
- Scalability
- Features

The Importance of Code Size



- Based on base 0.18µ implementation plus code RAM or cache
- Xtensa code ~10% smaller than ARM9 Thumb, ~50% smaller than MIPS-Jade, ARM9 and ARC
- ARM9-Thumb has reduced performance
- RAM/cache density = 8KB/mm^2

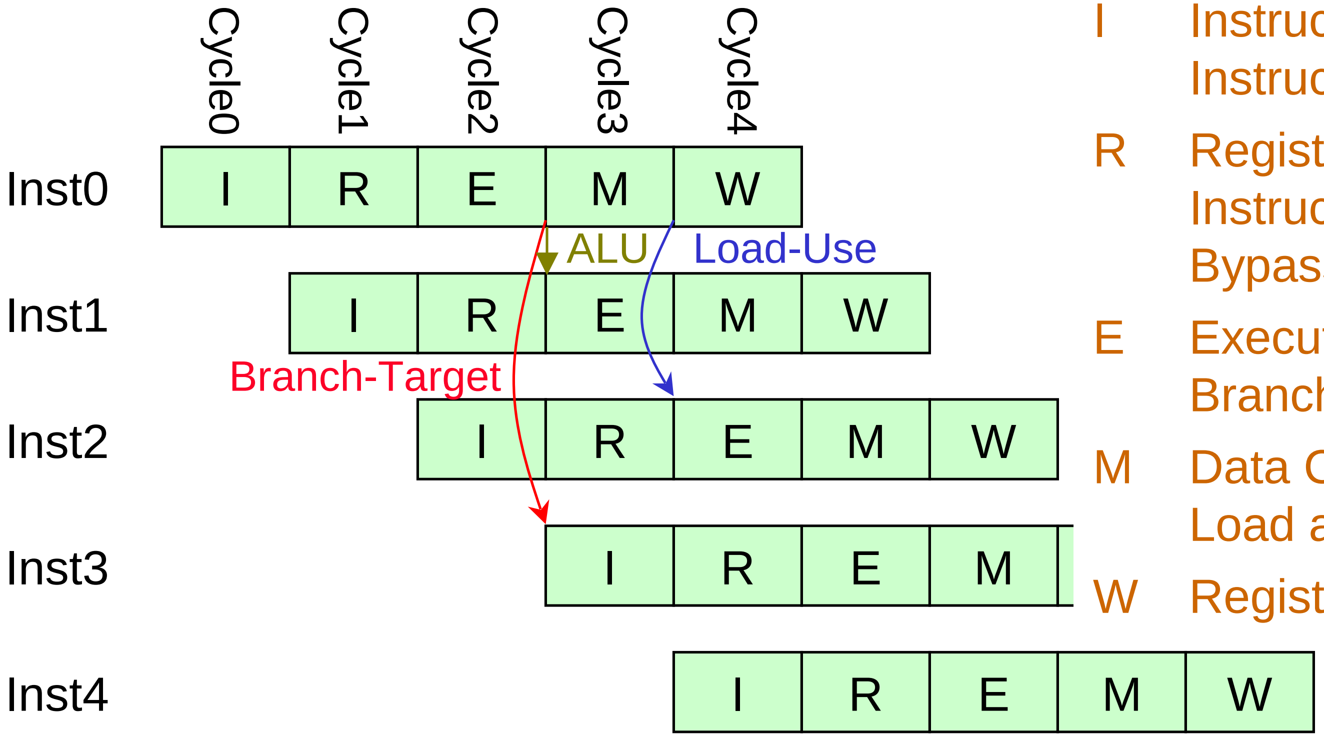
ISA Process

- Micro-architecture was firmer than ISA
- Created/circulated ISA alternatives
- Lots of arguing over alternatives
- Some data collected (but not much time!)
  - code size
  - performance
- Generally converged on solutions by consensus
- Generally followed our priority list

ISA Influences

- Major ISAs that influenced Xtensa
  - MIPS (e.g. compare-and-branch, MDMX, MIPS V)
  - IBM Power (ISA aids for ifetch, address modes)
  - Sun SPARC (register windows)
  - ARM Thumb (code size)
  - HP Playdoh (speculative loads)
  - DSPs (loop instructions)
- Other ISAs that shaped my thinking
  - CDC 6600, Cray-1
  - DEC PDP10
  - DEC PDP11, Motorola 68000
  - Multics, LLNL S-1, S-2
  - Cydrome, Multiflow

Target Pipeline



	Cycle0	Cycle1	Cycle2	Cycle3	Cycle4			
Inst0	I	R	E	M	W			
Inst1		I	R	E	M	W		
Inst2			I	R	E	M	W	
Inst3				I	R	E	M	
Inst4					I	R	E	M	W

I	Instruction Cache Access Instruction Align
R	Register Read Instruction Decode Bypass, Issue decision
E	Execute (ALU, TIE) Branch decision
M	Data Cache Access Load align
W	Register write

- ➢ One clock, rising-edge triggered flip-flops
  - no time borrowing between stages
- ➢ Use RAM-compiler generated Instruction and Data RAMs
  - registered address input

Pipeline Issues

- Why not superscalar?
  - Cost/benefit not right for this market
    - 2× register file read and write ports
    - Typical dual-issue adds 20-30% performance boost, not 2×
  - Design/verification time
  - Balance
    - Should add branch prediction or branches cost too much
- Why 5-stage (1980’s RISC in 2000)?
  - Cycle time cost too high for < 5 stages
  - Energy and cost issues for > 5 stages

Pipeline Implications

- Branches will be expensive
  - lack of time borrowing, edge-triggered RAM
  - try to compensate in ISA with more powerful branches
- Symmetry of I an M stages allows time for variable length instruction alignment
- Standard RISC principles:
  - Instructions must be simple to decode, issue, bypass
  - Register file read addresses must from fixed instruction fields

Early Controversies

- Performance/scalability vs. code size
- Multiple instruction sizes and instruction ≠ 32b
- Register windows
- How to handle the small size of immediate operands
- Instruction mnemonics
- DSP

Performance vs. Code Size

- Traditional performance-oriented ISA
  - Fixed 32b instruction word
    - supports 3 or 4 5-6b register fields
    - supports easy superscalar growth path
- Code-size oriented ISA
  - Most instructions < 32b (usually 16b)
    - 2 or 3 3-4b register fields (extra spills or moves)
  - Multiple instruction sizes
    - superscalar more difficult
- Considered 32/16, 24/12, and 24/16
  - Two sizes differentiated by a single bit
- Tensilica chose 24/16 in line with our priorities
  - best code size of the choices
  - good performance from 3 4b register fields

Register Windows

- Code size savings from elimination of save/restore
  - savings very application dependent
  - our estimate was 6-10%
- Issues
  - larger register file (adds to processor area)
    - especially with standard cell implementation
  - may impact real-time applications
  - windows not well-liked (colored by SPARC)
- Tensilica chose windows as per our priorities
  - fixed SPARC problems

Xtensa Instruction Formats

op2	op1	r	s	t	op0

E.g. `AR[r] ← AR[s] + AR[t]`

imm8	op1	s	t	op0

E.g. `if AR[s] < AR[t] goto PC+imm8`

imm12	s	t	op0

E.g. `if AR[s] = 0 goto PC+imm12`

imm16	t	op0

E.g. `AR[t] ← AR[t] + imm16`

imm18	n	op0

E.g. `CALL0 PC+imm18`

r	s	t	op0

E.g. `AR[r] ← AR[s] + AR[t]`

Code Size

- Bits per instruction reduction (0.62)
  - 24-bit encoding (25%)
  - 16-bit optional encodings (12%)
- Instruction count (0.91)
  - Compound instructions

    -15% from compare-and-branch
    -2% from shift add/subtract
    -2% from shift mask (extract)
    -2% from L32R vs. 2-instruction 32-bit immediate synthesis
  - Register windows

    -6% from elimination of functional call overhead (save/restore)
  - 24-bit encoding

    +10% from register spill
    +8% from small immediates
- Combined 0.91 × 0.62 = 0.56

Code Size Comparison — ARM

```
for (i=0; i < NUM; i++)
    if (histogram[i] != NULL)
        insert (histogram[i], &tree);
```

Xtensa code

```
L16: addx4 a2, a3, a5
     l32i  a10, a2, 0
     beqz  a10, L15
     add   a11, a4, a7
     call8 insert
L15: addi  a3, a3, 1
     bge   a6, a3,L16
```

ARM code

```
J4:ADD   a1,sp,#4
   LDR   a1,[a1,a3,LSL#2]
   CMP   a1,#0
   MOVNE a2,sp
   BLNE  insert
   ADD   a3,a3,#1
   CMP   a3,#&3e8
   BLT   J4
```

Thumb code

```
L4: LSL r1,r7,#2
    ADD r0,sp,#4
    LDR r0,[r0,r1]
    CMP r0,#0
    BEQ L13
    MOV r1,sp
    BL  insert
L13:ADD r7,#1
    CMP r7,r4
    BLT L4
```

7 instructions
17 bytes

8 instructions
36 bytes

10 instructions
20 bytes

Xtensa ISA Summary

- 80 base instructions
  - Load and Store (8 instructions)
  - Move (5 instructions)
  - Shift (13 instructions)
  - Arithmetic Operations (12 instructions)
  - Logical Operations (AND , OR , XOR)
  - Jump and Branch (29 instructions)
  - Zero Overhead Loops (3 instructions)
  - Pipeline Control (7 instructions)

Xtensa ISA Features

	Code size	Energy efficiency	Performance	Extensibility	Scalability
24-bit encoding	3			3	
16-bit encoding	3	3			
Register windows	3	3	3		
Compare and branch	3		3		
Bit test/mask and branch	3		3		
No branch delay	3				3
Funnel shifts			3		
Right shift and mask	3		3		
Conditional moves			3		3
Speculative loads			3		3
Zero-overhead loop		3	3		3
TIE	3	3	3	3	
Multiprocessor			3		3
DSP option			3		
FP option			3		

Compare and Branch

C

```
if (a < b) {
        c = 0;
}
```

SPARC

```
        cmp     %o0, %o1
        bge     L1
        <<delayslot>>
        or      %g0, 0, %o2
L1:
```

2 cycle branch untaken or taken
(3 if nop in delay slot)

Xtensa

```
        bge     a2, a3, L1
        movi    a4, 0
L1:
```

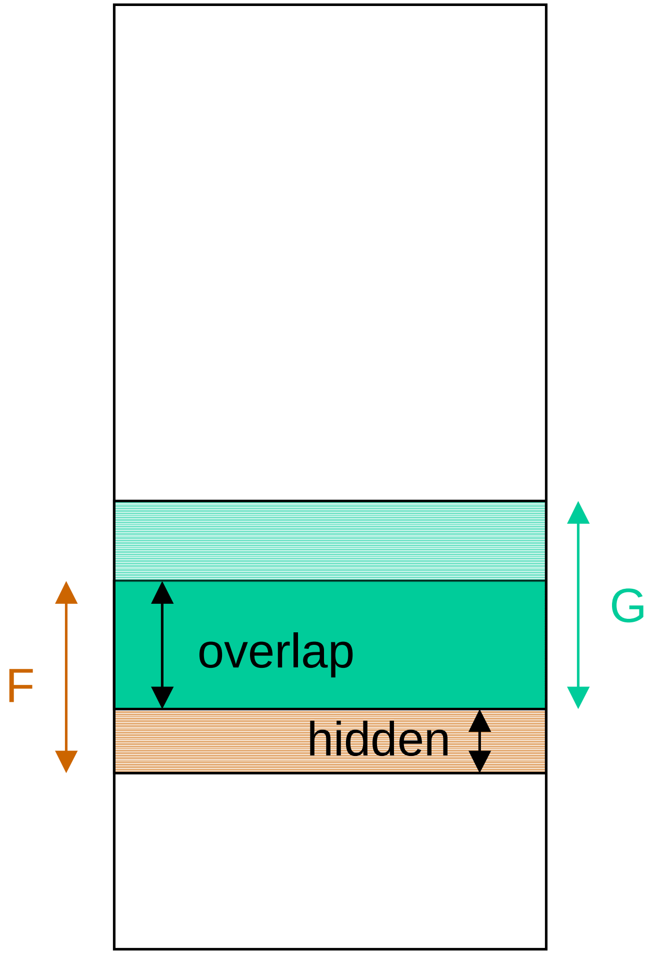
1 cycle branch if untaken,
3 cycle branch if taken

Zero-Overhead Loops

```
    loopgtz a0, endloop
loop:
    body0
      •
      •
      •
    bodyN
endloop:
```

- Processor automatically branches to body0 after executing bodyN the number of times in a0
- No branch penalty in most cases
- Implemented with the LBEG, LEND, and LCOUNT special registers

Overlapped Register Windows



- Routine F calls routine G incrementing register file pointer by 4, 8, or 12
- F and G's windows into the physical register file overlap
- F can pass register parameters to G by writing its high registers
- The register file pointer increment hides 4-12 of F's registers
- No save or restores required unless pointer wraps

Window Code Example

```
Foo:
   entry sp, 16
   movi  a6, 1      // a6 will become a2 in Bar after entry
   l32i  a7, a2, 4  // a7 will become a3 in Bar after entry
   call4 Bar        // call Bar, request increment of 4
   addi  a2, a6, 1  // a6 is Bar’s a2 before the retw
   retw

Bar:
   entry sp, 16     // move window by caller’s increment
   add   a2, a2, a3// add our arguments, with result
                    // to return value register
   retw             // move window back (decrement)
```

Window Code Comparison

Traditional

```
f: addi  sp, sp, -framesize
   s32i  a0, framesize-12(sp)
   s32i a12, framesize-8(sp)
   s32i a13, framesize-4(sp)
   …
   l32i  a0, framesize-12(sp)
   l32i a12, framesize-8(sp)
   l32i a13, framesize-4(sp)
   addi  sp, sp, framesize
   ret
```

With Windows

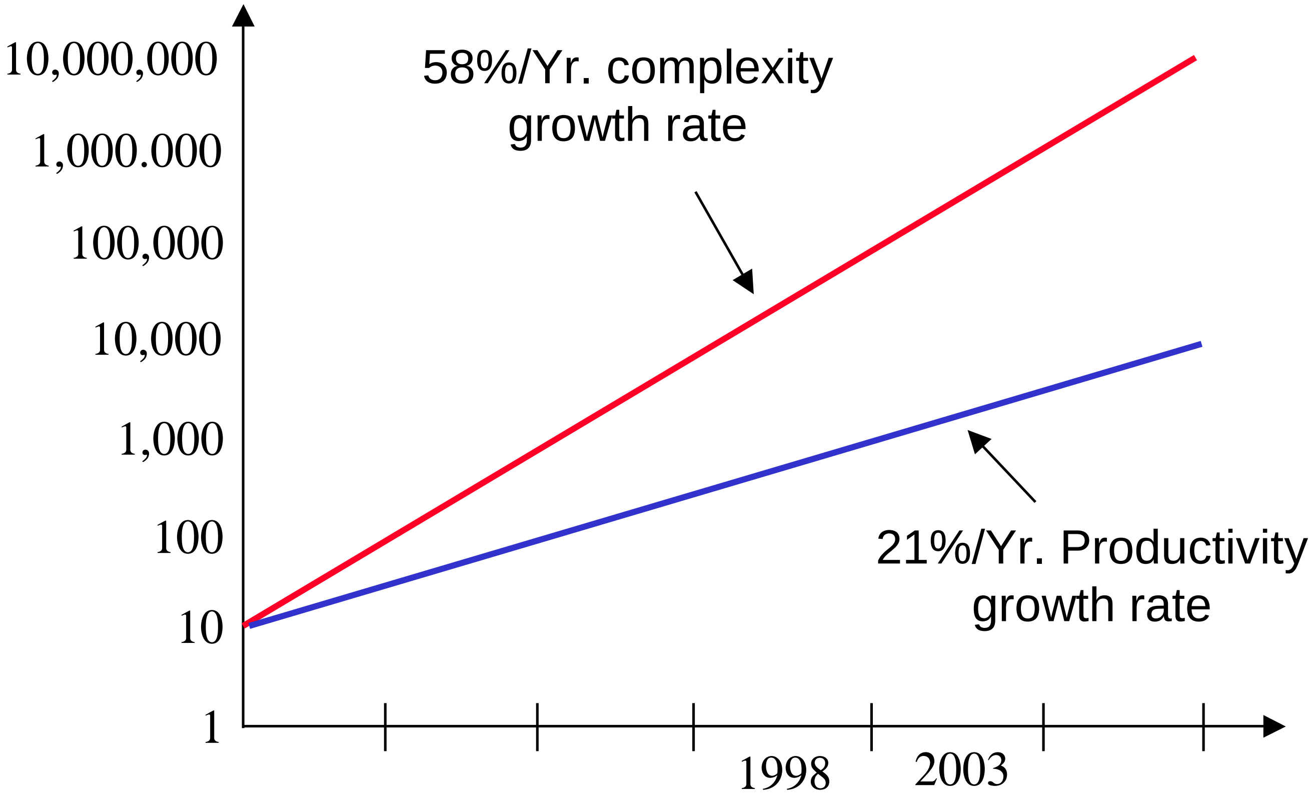
```
f: entry sp, framesize
   …
   retw
```

- ➢ Smaller
- ➢ Faster

Outline

- ➢ About Tensilica
  - History, getting started, etc.
- ➢ Application-Specific Processors
  - What’s different
- ➢ Xtensa ISA
  - What we did and why
- ➢ Extensibility via the TIE (Tensilica Instruction Extension) Language

Productivity Gap



58%/Yr. complexity growth rate

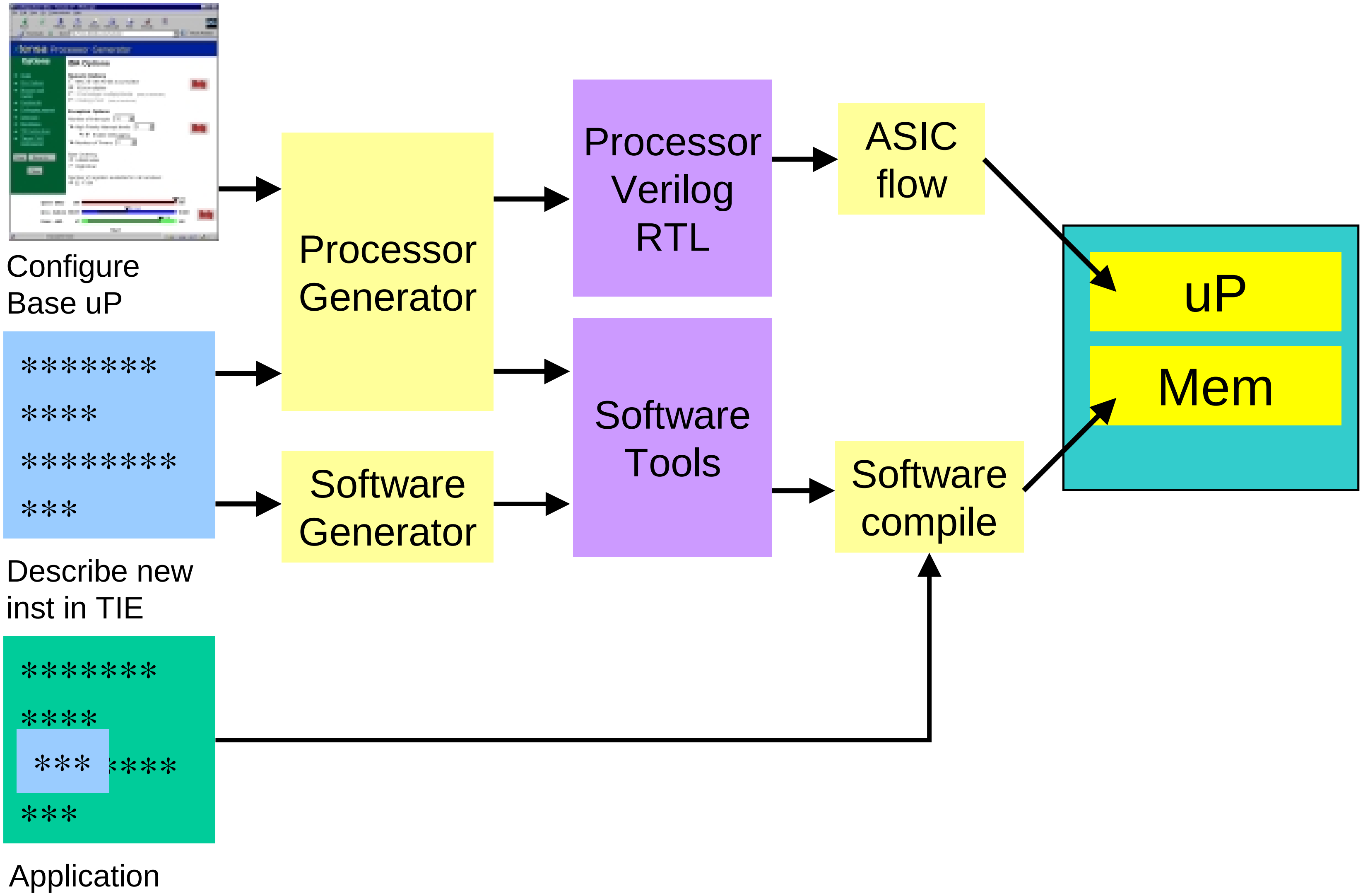
21%/Yr. Productivity growth rate

Logic Transistor / Chip (K)

Transistor/Staff-month

Source: NTRS’97

TIE Overview



Configure Base uP

Describe new inst in TIE

Application

Processor Generator

Software Generator

Processor Verilog RTL

Software Tools

ASIC flow

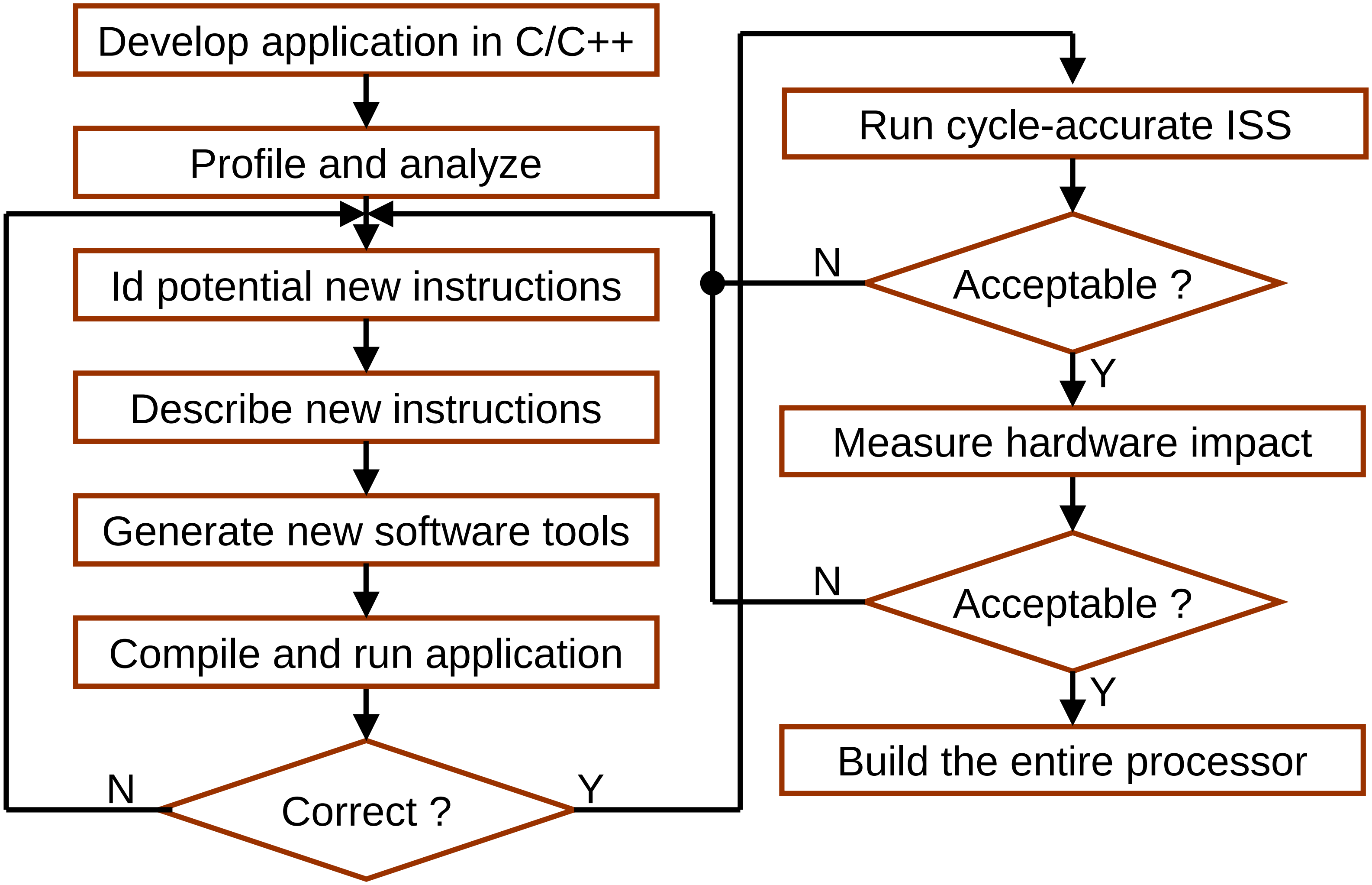
Software compile

uP

Mem

****
****
****
***

TIE Design Cycle


Develop application in C/C++
Profile and analyze
Id potential new instructions
Describe new instructions
Generate new software tools
Compile and run application
N
Correct ?
Y
Run cycle-accurate ISS
N
Acceptable ?
Y
Measure hardware impact
N
Acceptable ?
Y
Build the entire processor

Tensilica Instruction Extension

- No micro-architecture (implementation) details
  - same TIE will work with new base
  - decode, interlock, bypass, and pipelining automatic
- Automatic configuration of software tools
  - compiler
  - instruction-set simulator
  - debugger
  - etc.
- Automatic synthesis of efficient hardware compatible with the base processor
- Extension language, not a language to describe a complete CPU

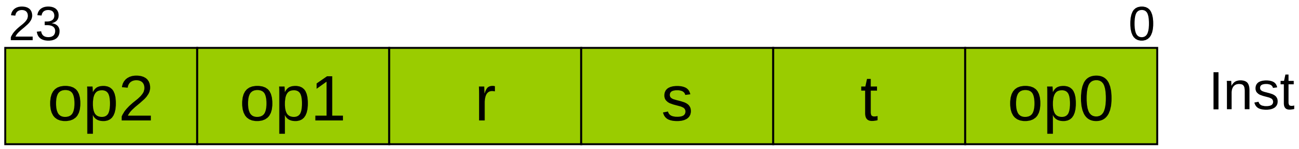
Major sections in TIE

- Instruction fields
- Opcode
- Operands
- Instruction semantics

Instruction Field Definition

- TIE code:

```
field        op0  Inst[3:0]
field        op1  Inst[19:16]
field        op2  Inst[23:20]
field        r    Inst[15:12]
field        s    Inst[11:8]
field        t    Inst[7:4]
```



23 0

op2 op1 r s t op0 Inst

Opcode Definition

- TIE code:

```
opcode  QRST   op0=4’b0000
opcode  CUST0  op1=4’b1100     QRST
opcode  ADD4   op2=4’b0000     CUST0
```

- TIE compiler generates decode logic

23					0	
0000	1101	r	s	t	0000	ADD4 Instruction

Operand Definition

- TIE code:

```
operand  ars   s     {AR[s]}
operand  art   t     {AR[t]}
operand  arr   r     {AR[r]}
iclass   rrr   {ADD4}{out arr, in ars, in art}
```

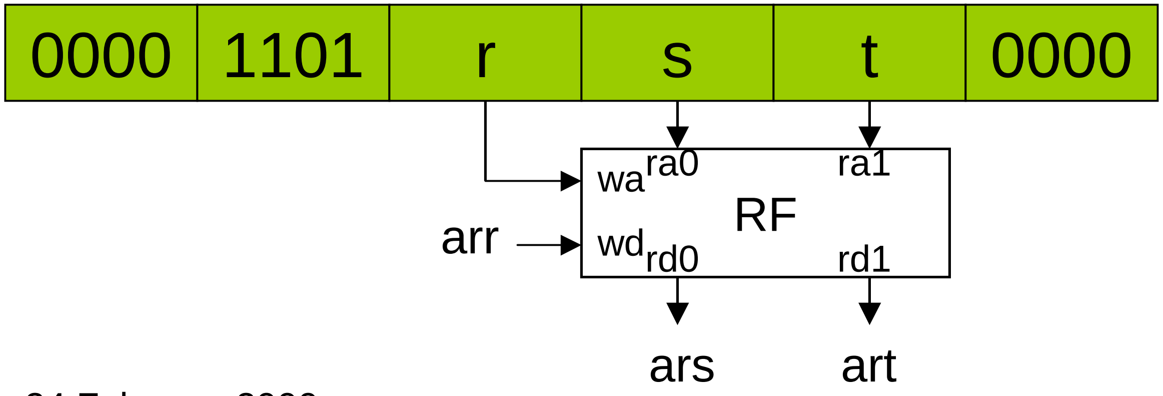
- Assembly example:

```
ADD4     a2, a3, a5
```

- C example:

```
X = ADD4(y, z);
```

- TIE compiler generates interlock and bypass logic

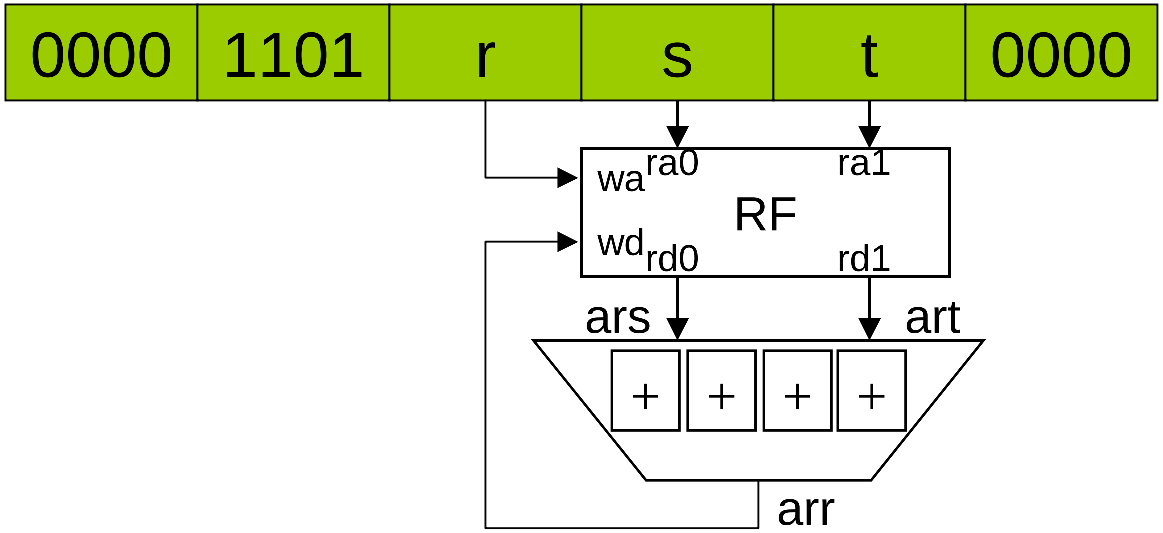


ADD4 Instruction

Semantic Description

- TIE code:

```
semantic add4_semantic {ADD4} {
    wire [7:0] arr0 = ars[ 7: 0] + art[ 7: 0];
    wire [7:0] arr1 = ars[15: 8] + art[15: 8];
    wire [7:0] arr2 = ars[23:16] + art[23:16];
    wire [7:0] arr3 = ars[31:24] + art[31:24];
    assign arr = {arr3, arr2, arr1, arr0}; }
```



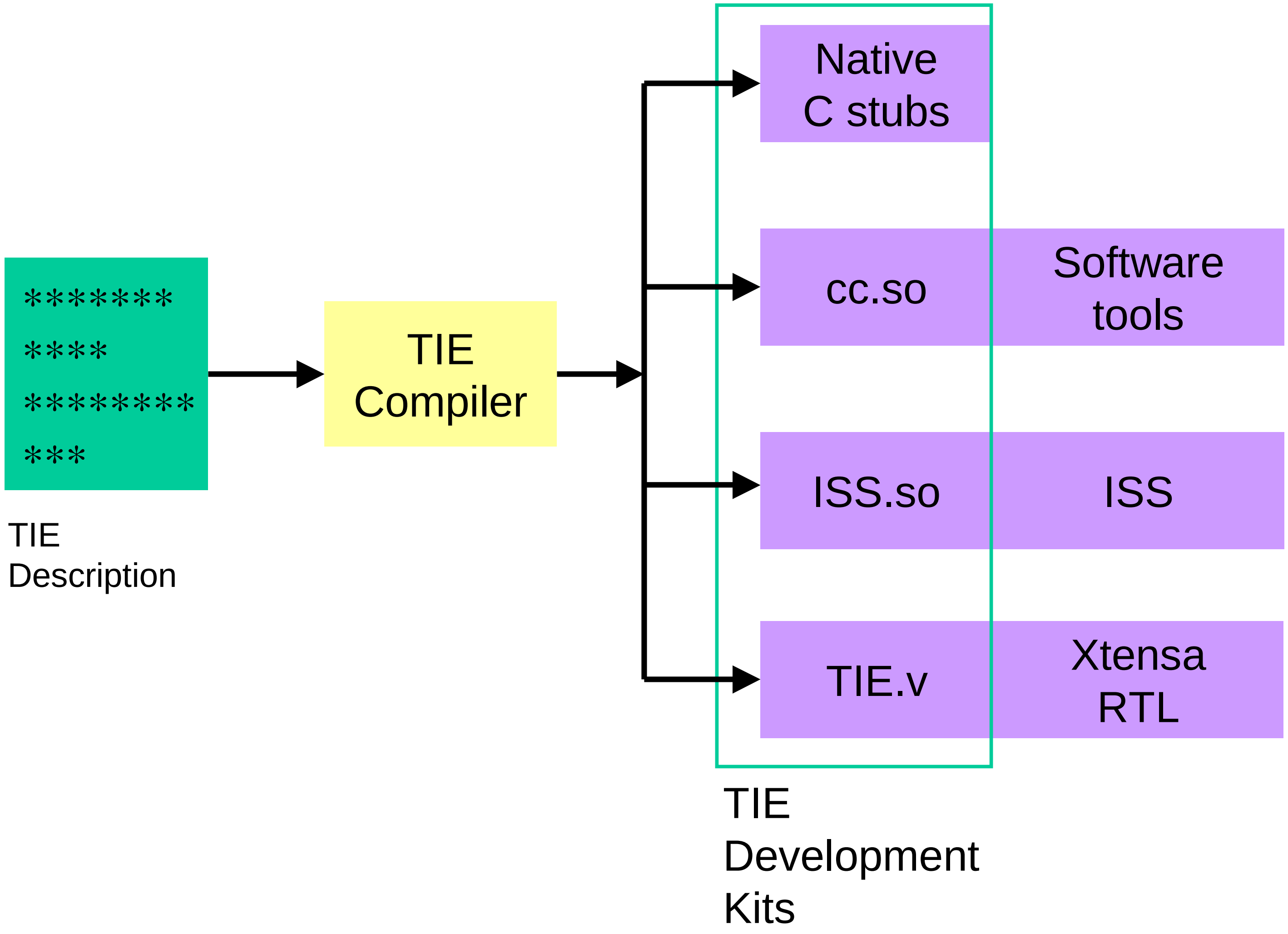
0000	1101	r	s	t	0000

ADD4 Instruction

Complete Example

```
opcode   ADD4  op2=4’b0000 CUST0
iclass   rrr   {ADD4}      {out arr, in ars, in art}
semantic add4_semantic {ADD4} {
    wire arr0 = ars[ 7: 0] + art[ 7: 0];
    wire arr1 = ars[15: 8] + art[15: 8];
    wire arr2 = ars[23:16] + art[23:16];
    wire arr3 = ars[31:24] + art[31:24];
    assign arr = {arr3, arr2, arr1, arr0};
}
```

TIE Development Process



Using TIE Instruction in C

```
#ifdef NATIVE
#include ADD4_cstub.c
#endif

int a[ ], b[ ], c[ ];
char *x=a, *y=b, *z=c;
...
read(x);
read(y);
for (i = 0; i < n; i++) {
    c[i] = ADD4(a[i], b[i]);
}
write(z);
...
```

Testing new instructions on the host

```
shell> gcc -o app –DNATIVE app.c
shell> app
```

- Objectives
  - Verify TIE description
  - Verify application code
- Advantage
  - Short iteration cycle

Testing new instructions on Xtensa simulator

```
shell> xt-gcc -o app app.c
shell> iss app
```

- ➢ Objectives
  - Testing TIE description
  - Testing application
  - Measuring performance
- ➢ Advantage
  - Cycle-accurate

Checking the Hardware

```
shell> vi app.dcsh
shell> dc_shell -f app.dcsh
shell> vi app.report
```

- ➢ Objectives
  - • Measuring cycle-time impact
  - • Measuring area impact
- ➢ Advantage
  - • Time-accurate
  - • Cost-accurate

Data Encryption Standard

- Initial step

  (R, L) = Initial_permutation(Din_{64})

- Iterate 16 times
  - Key generation

    (C, D) = PC1(k)
    n = rotate_amount (function of iteration count)
    C = rotate_right(C, n)
    D = rotate_right (D, n)
    K = PC2(D, C)

  - Encryption

    $R_{i+1} = L_i \oplus$ Permutation (S_Box ($K \oplus$ Expansion (R)))
    $L_{i+1} = R_i$

- Final step

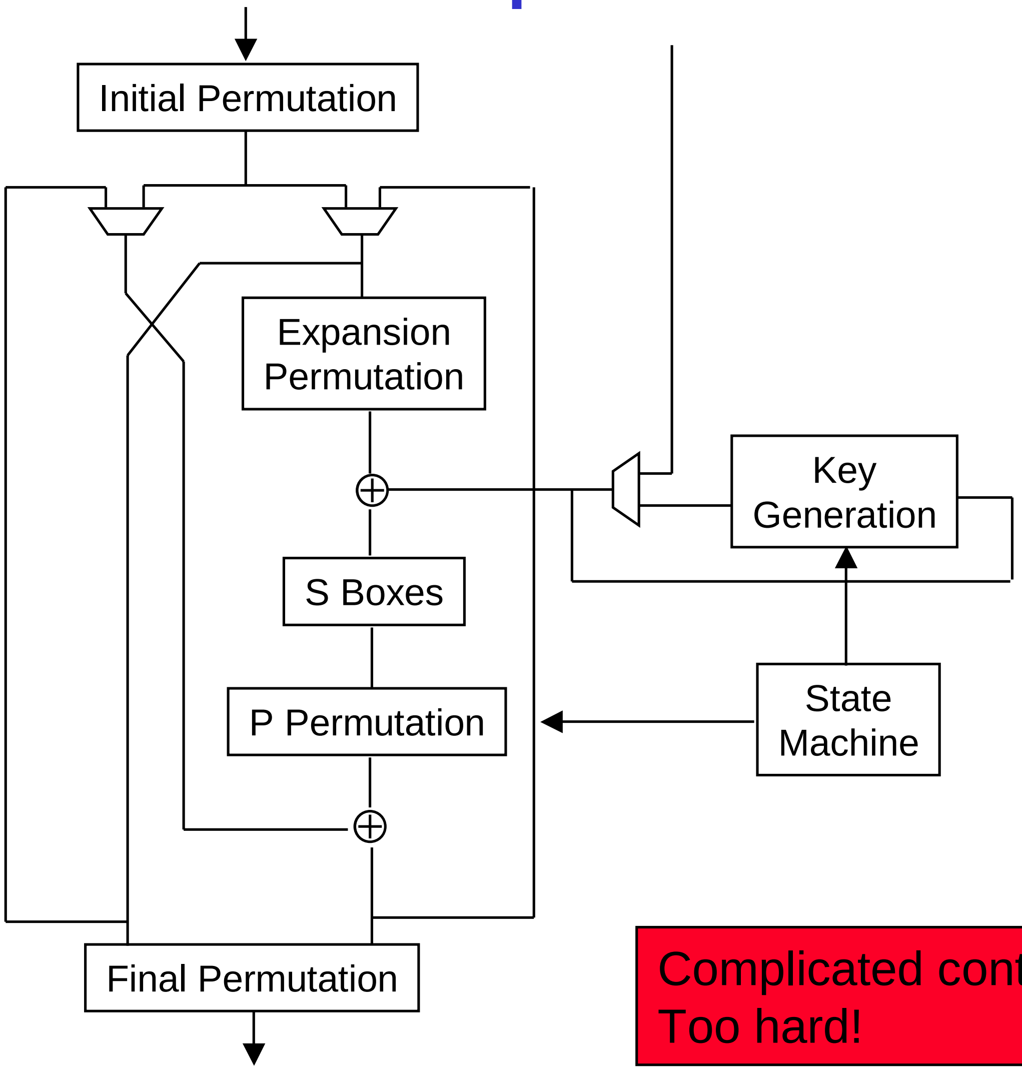
  $Dout_{64}$ = Final_permutation(L, R)

DES Software Implementation

```
static unsigned permute(unsigned char *table, int n,
                        unsigned hi, unsigned lo)
{
    int ib, ob;
    unsigned out = 0;
    for (ob = 0; ob < n; ob++) {
        ib = table[ob] - 1;
        if (ib >= 32) {
            if (hi & (1 << (ib-32)))   out |= 1 << ob;
        } else {
            if (lo & (1 << ib))        out |= 1 << ob;
        }
    }
    return out;
}
```

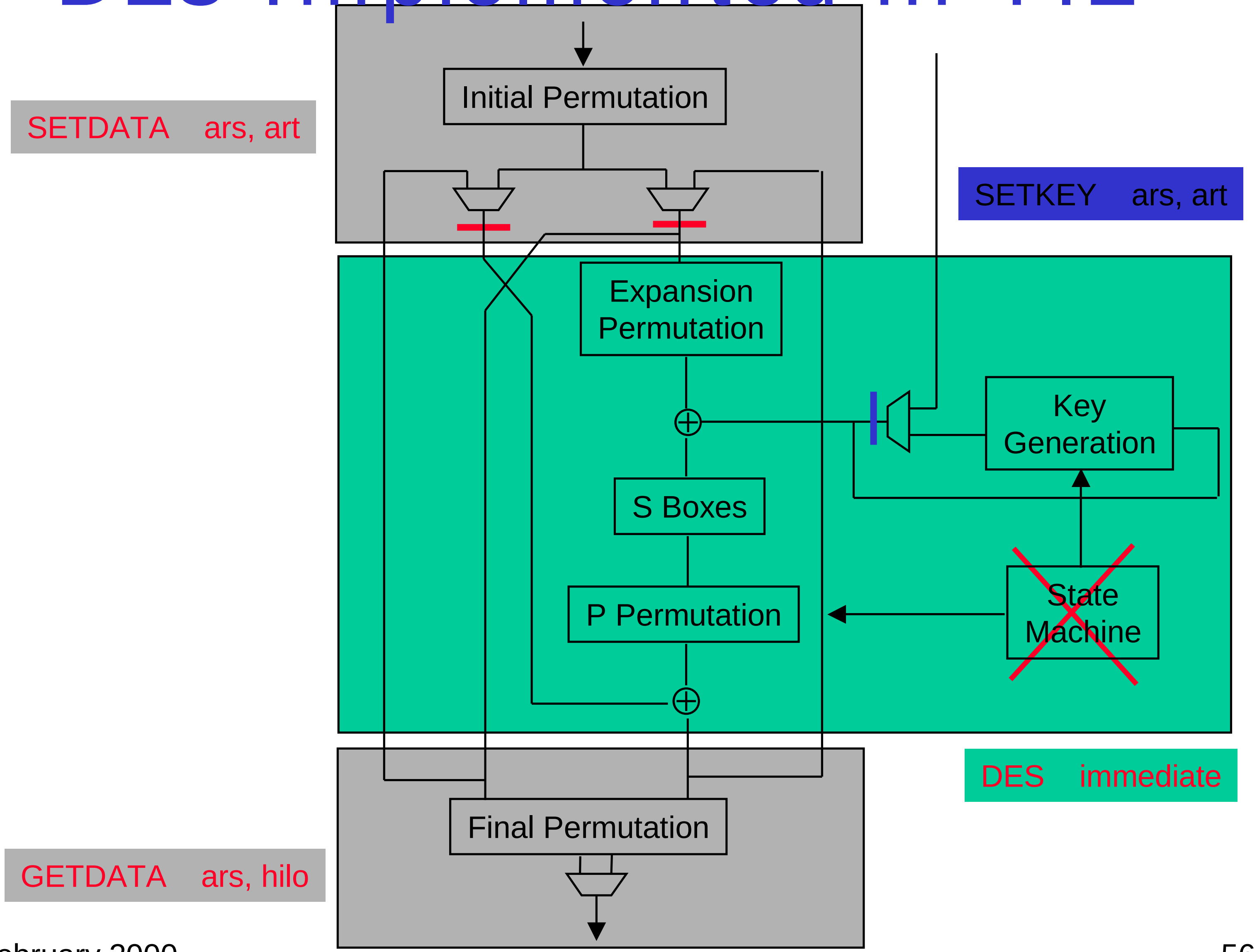
Too much computation!
Too slow!

DES Hardware Implementation



Complicated control logic! Too hard!

DES Implemented in TIE



DES Program

Encryption

```
SETKEY(K_hi, K_lo);
for (;;) {
    … /* read data */
    SETDATA(D_hi, D_lo);
    DES(ENCRYPT1);
    DES(ENCRYPT1);
    DES(ENCRYPT2);
    DES(ENCRYPT2);
    DES(ENCRYPT2);
    DES(ENCRYPT2);
    DES(ENCRYPT2);
    DES(ENCRYPT2);
    DES(ENCRYPT1);
    DES(ENCRYPT2);
    DES(ENCRYPT2);
    DES(ENCRYPT2);
    DES(ENCRYPT2);
    DES(ENCRYPT2);
    DES(ENCRYPT2);
    DES(ENCRYPT1);
    E_hi = GETDATA(hi);
    E_lo = GETDATA(lo);
    … /* write encrypted data */ }
```

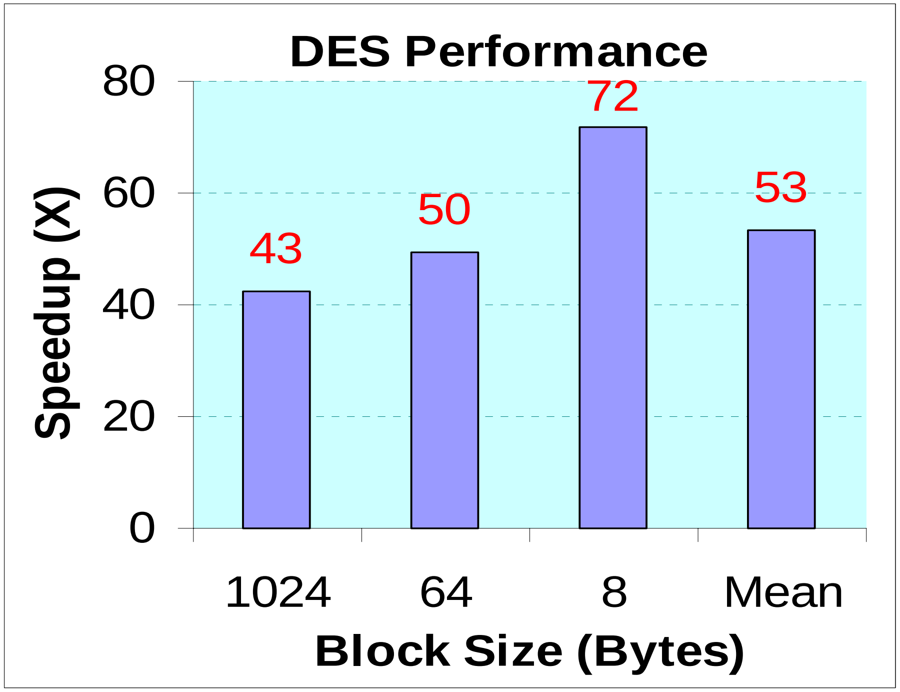
Decryption

```
SETKEY(K_hi, K_lo);
for (;;) {
    … /* read encrypted data */
    SETDATA(D_hi, D_lo);
    DES(DECRYPT1);
    DES(DECRYPT2);
    DES(DECRYPT2);
    DES(DECRYPT2);
    DES(DECRYPT2);
    DES(DECRYPT2);
    DES(DECRYPT2);
    DES(DECRYPT1);
    DES(DECRYPT2);
    DES(DECRYPT2);
    DES(DECRYPT2);
    DES(DECRYPT2);
    DES(DECRYPT2);
    DES(DECRYPT2);
    DES(DECRYPT1);
    DES(DECRYPT1);
    E_hi = GETDATA(hi);
    E_lo = GETDATA(lo);
    … /* write data */ }
```

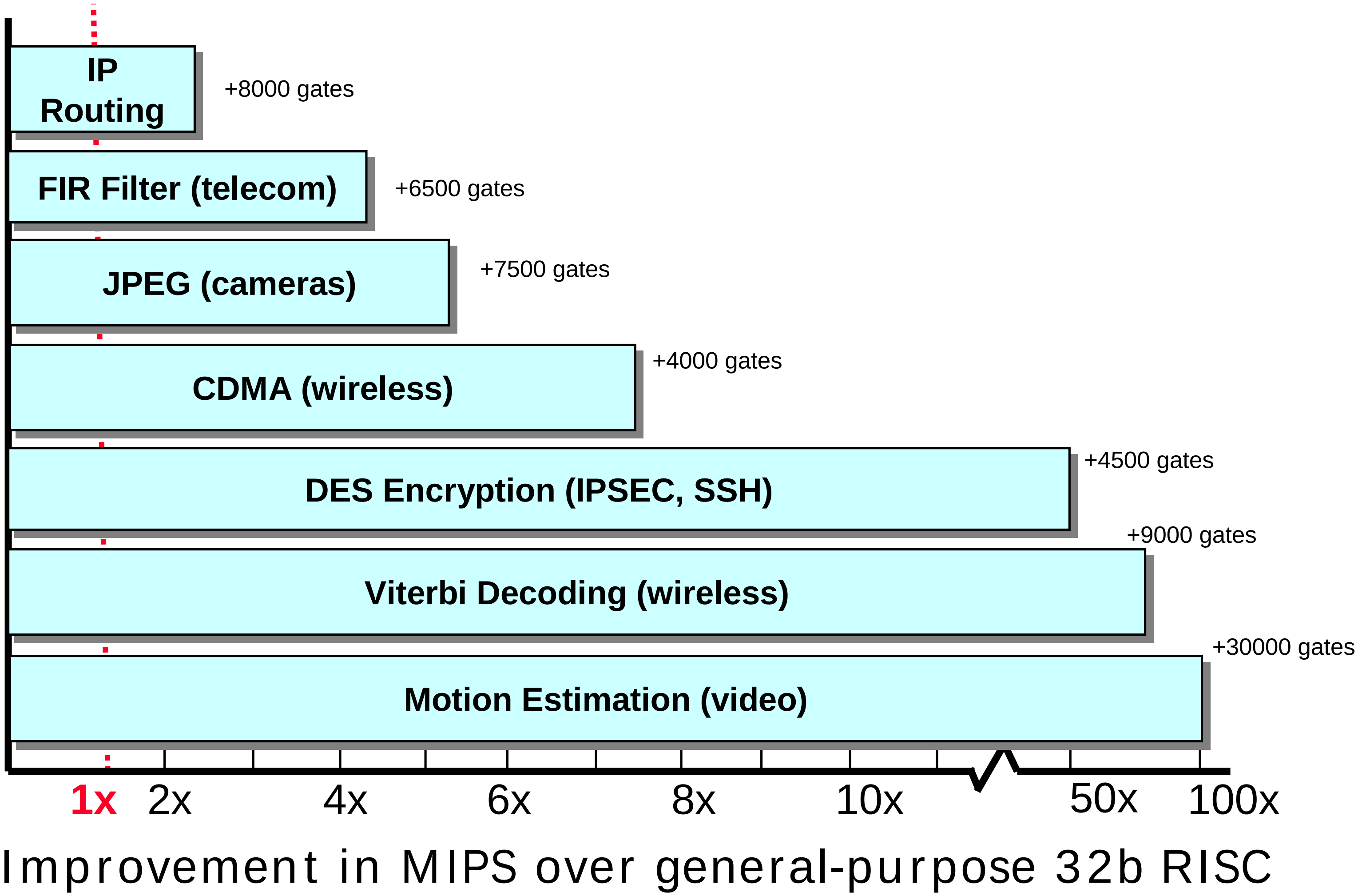
Triple DES Example

- Application:
  - Secure Shell Tools (SSH)
  - Internet Protocol for Security (IPSEC)

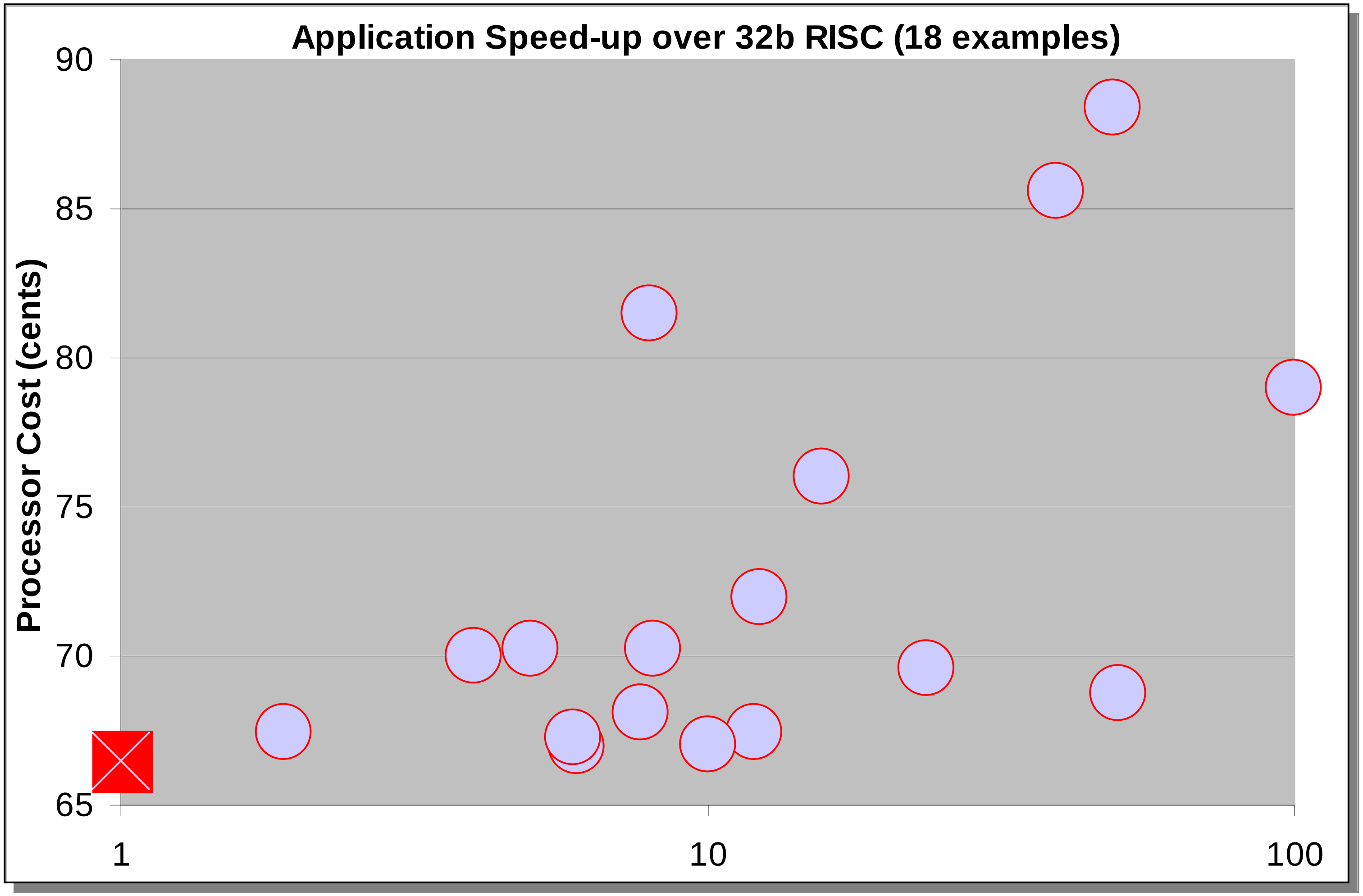
- Add 4 TIE instructions:
  - 80 lines of TIE description
  - No cycle time impact
  - ~1700 additional gates
  - Code-size reduced



Result: Flexibility + Efficiency

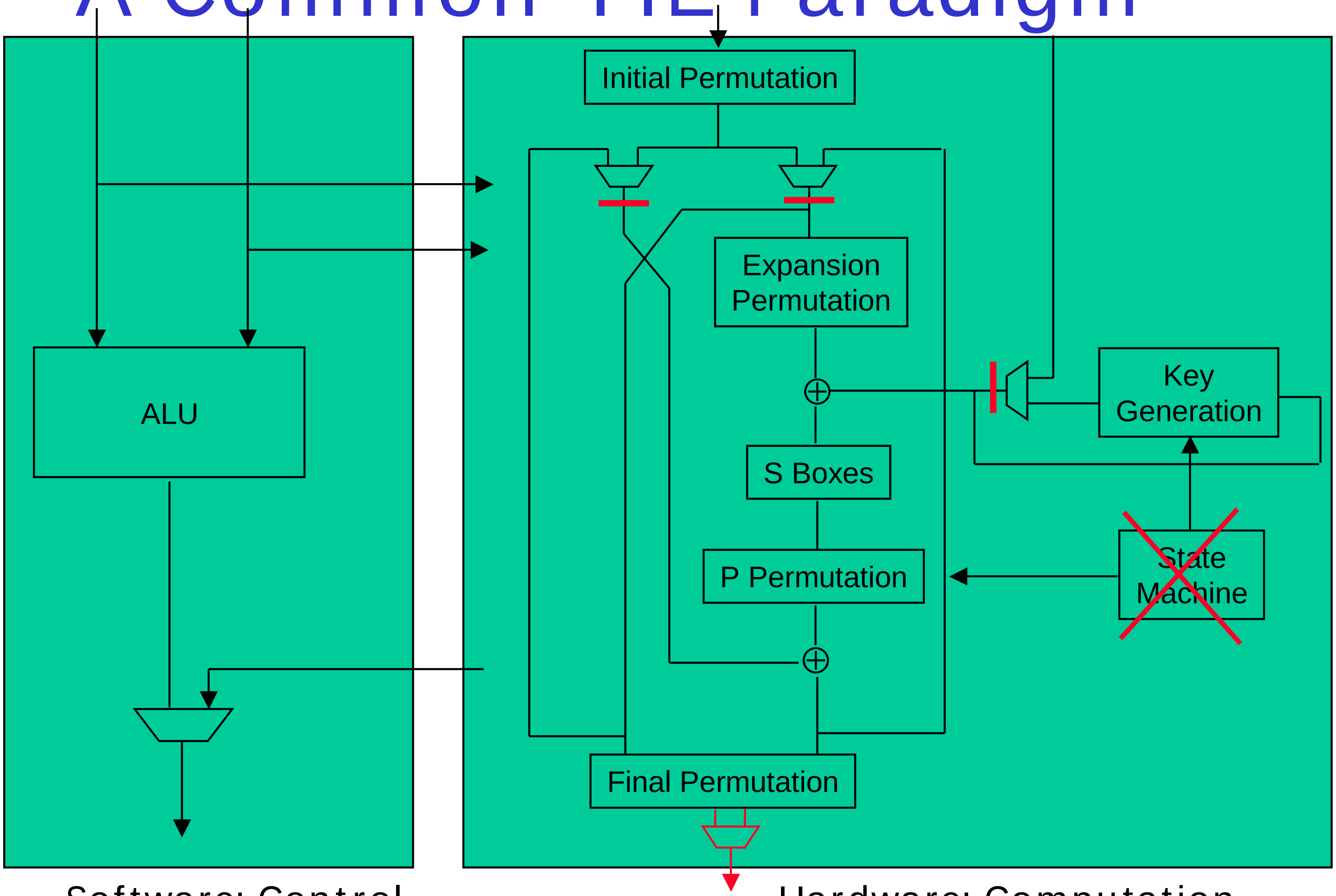


Cost <$1 , 5 -100x speed-up



- Cost = marginal cost for core+memory in 0.25µ foundry in volume
- Data from communication and consumer applications: FIR filter, Viterbi, DES, JPEG, Motion Estimation, W-CDMA, Packet Flow, RGB2CYMK, RGB2CYMK, RGB2YIQ, Grayscale Filter, Auto-Correlation,

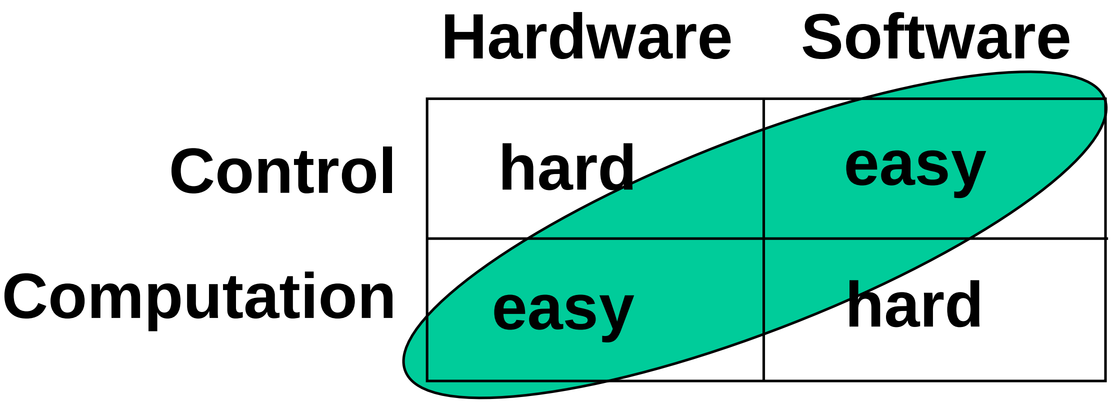
A Common TIE Paradigm



Software: Control

Hardware: Computation

Summary continued



	Hardware	Software
Control	hard	easy
Computation	easy	hard

Application-specific instructions

Conclusion

- Presentation
    - About Tensilica
    - Application-Specific Processors
    - Xtensa ISA
    - TIE
- Is there anything else you would like me to cover?