CS184a: Computer Architecture (Structure and Organization)

Day 23: March 7, 2005
Specialization

Previously

- How to support bit processing operations
- How to compose any task

Today

- What bit operations do I need to perform?
- Specialization
  - Binding Time
  - Specialization Time Models
  - Specialization Benefits
  - Expression

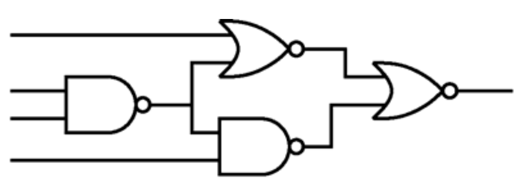
Quote

- The fastest instructions you can execute, are the ones you don't.

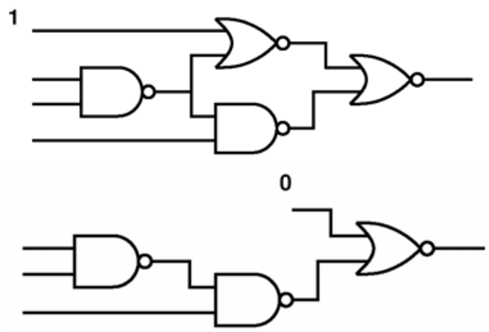
Idea

- **Goal:** Minimize computation must perform
- Instantaneous computing requirements less than general case
- Some data known or predictable
  - compute minimum computational residue
- As know more data → reduce computation
- Dual of **generalization** we saw for local control

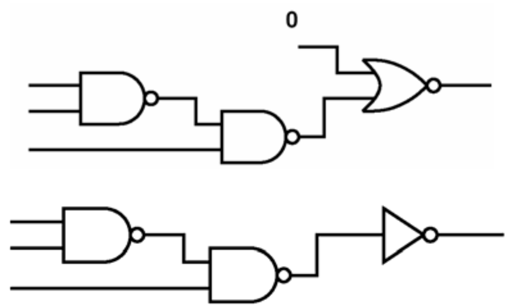
Know More → Less Compute



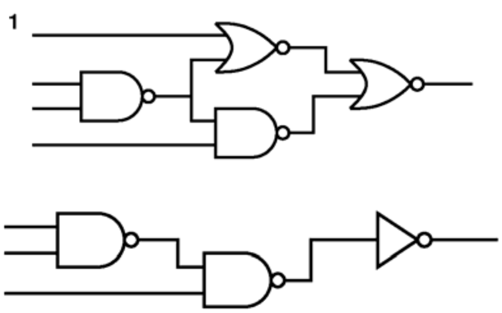
Know More → Less Compute



Know More → Less Compute



Know More → Less Compute



Typical Optimization

- Once know another piece of information about a computation
  - (data value, parameter, usage limit)
- Fold into computation
  - producing smaller computational residue

Opportunity Exists

- Spatial unfolding of computation
  - can afford more specificity of operation
  - *E.g.* last assignment (FIR,IIR)
- Fold (early) bound data into problem
- Common/exceptional cases

Opportunity

- Arises for programmables
  - can change their *instantaneous* implementation
  - don't have to cover all cases with a single configuration
  - can be heavily specialized
    - while still capable of solving entire problem
      - (all problems, all cases)

Opportunity

- With bit level control
  - larger space of optimization than word level
- While true for both spatial and temporal programmables
  - bigger effect/benefits for spatial

Multiply Example

Architecture	Feature Size (λ)	Area and Time	16×16 mpy $\frac{1}{\lambda^2 s}$	16×16 scale $\frac{1}{\lambda^2 s}$	8×8 mpy $\frac{1}{\lambda^2 s}$	8×8 scale $\frac{1}{\lambda^2 s}$
Custom 16×16	0.63μm	2.6Mλ^2, 40 ns	9.6	9.6	9.6	9.6
Custom 8×8	0.80μm	3.3Mλ^2, 4.3 ns			70	70
Gate-Array 16×16	0.75μm	26Mλ^2, 30ns	1.3	1.3	1.3	1.3
FPGA	0.60μm	1.25Mλ^2/CLB				
(XC4K)		316 CLBs, 26 ns	0.097			
		84 CLBs, 40 ns		0.24		
		220 CLBs, 12.1 ns			0.30	
		22 CLBs, 25 ns				1.5
16b DSP	0.65μm	350Mλ^2, 50 ns	0.057	0.057	0.057	0.057
RISC	0.75μm	125Mλ^2, 66 ns/cycle				
(no multiplier)		two 16b operands – 44 cycles	0.0028			
		16b constant – 7 cycles		0.017		
		one 8b operand – 24 cycles			0.0051	
		8b constant – 4 cycles				0.030

Multiply Show

- Specialization in datapath width
- Specialization in data

Benefits

Empirical Examples

Benefit Examples

- UART
- Pattern match
- Less than
- Multiply revisited
  - more than just constant propagation
- ATR

UART

- I8251 Intel (PC) standard UART
- Many operating modes
  - bits
  - parity
  - sync/async
- Run in same mode for length of connection

UART FSMs

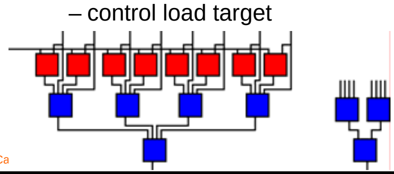
FSM	Fully Generic				Specialized	
	Speed Mapped		Area Mapped			
	CLBs	path	CLBs	path	CLBs	path
I8251 processor i/o	11	3.5	11	3.5		
	fast (any configuration)				6.5	2
	small (any configuration)				5.5	2.5
I8251 transmitter	57.5	4.5	57.5	4.5		
	Asynchronous, parity				24	4
	Asynchronous, no parity				27	4.5
	2 Sync chars, parity				31	4.5
	1 Synch char, noparity				31	4
I8251 receiver	52.5	5.5	52.5	5.5		
	Asynchronous, parity				32.5	4.5
	Asynchronous, no parity				36	4.5
	External Sync, parity				29.5	4.5
	Internal, 2 Sync chars, parity				27	3.5
	Internal, 1 Sync chars, parity				28	4.5
	Internal, 1 Sync chars, no parity				31.5	4.5

UART Composite

design	Fully Generic				Specialized	
	Speed Mapped		Area Mapped			
	CLBs	path	CLBs	path	CLBs	path
I8251 core	358.5	8.5	348.5	10.5		
	Async, 64 clks/bit, 8e2				216.5	7
	Async, 16 clks/bit, 8n1				201	6
	Async, 1 clks/bit, 5n1				141.5	4.5
	Sync, internal, 2 sync, 8o				165	4.5
	Sync, external, 5n				136	5.5

Pattern Match

- Savings:
  - 2N bit input computation → N
  - if N variable, maybe trim unneeded
  - state elements store target
  - control load target



Pattern Match

(size)	CLBs	path	CLBs	path	AT Ratio	
$a = b$	b variable		b constant			w/state
(8)	2.5 (+4)	2	1.5	2	0.60	0.23
(16)	5.5 (+8)	3	2.5	2	0.30	0.12
(32)	10.5 (+16)	3	5.5	3	0.52	0.21
(64)	21.5 (+32)	4	10.5	3	0.37	0.15

Less Than (Bounds check?)

- Area depend on target value
- But all targets less than generic comparison

Function	Speed Mapped		Area Mapped		Speed Mapped		Area Mapped	
(size)	CLBs	path	CLBs	path	CLBs	path	CLBs	path
$a \leq b$	b variable				b constant			
(8)	4	8	4	8	≤ 2	≤ 2	≤ 1.5	≤ 3
(16)	18.5	14	16.5	16	≤ 6.5	≤ 3	≤ 3	≤ 5
(32)	35	19	36	24	≤ 13.5	≤ 4	≤ 6	≤ 11
(64)	77.5	20	74.5	28	≤ 30	≤ 5	≤ 14	≤ 16

Multiply (revisited)

- Specialization can be more than constant propagation
- Naïve,
  - save product term generation
  - complexity number of 1's in constant input
- Can do better exploiting algebraic properties

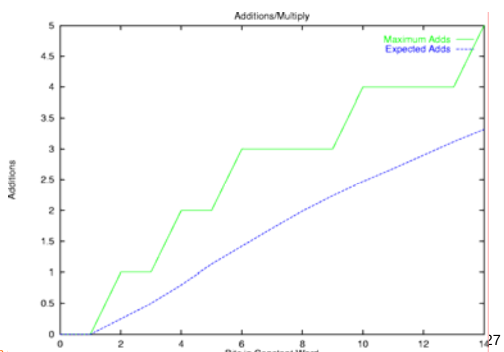
Multiply

- Never really need more than $\lfloor N/2 \rfloor$ one bits in constant
- If more than N/2 ones:
  - invert c $(2^{N+1}-1-c)$
  - (less than N/2 ones)
  - multiply by x $(2^{N+1}-1-c)x$
  - add x $(2^{N+1}-c)x$
  - subtract from $(2^{N+1})x$ $= cx$

Multiply

- At most $\lfloor N/2 \rfloor+2$ adds for any constant
- Exploiting common subexpressions can do better:
  - e.g.
    - c=10101010
    - t1=x+x<<2
    - t2=t1<<5+t1<<1

Multiply

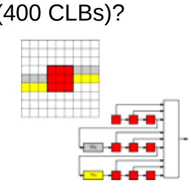


Example: ATR

- Automatic Target Recognition
  - need to score image for a number of different patterns
    - different views of tanks, missles, etc.
  - reduce target image to a binary template with don't cares
  - need to track many (e.g. 70-100) templates for each image region
  - templates themselves are sparse
    - small fraction of care pixels

Example: ATR

- 16x16x2=512 flops to hold single target pattern
- 16x16=256 LUTs to compute match
- 256 score bits→8b score ~ 500 adder bits in tree
- more for retiming
- ~800 LUTs here
- Maybe fit 1 generic template in XC4010 (400 CLBs)?



Example: UCLA ATR

- UCLA
  - specialize to template
  - ignore don't care pixels
  - only build adder tree to care pixels
  - exploit common subexpressions
  - get 10 templates in a XC4010

[Villasenor et. al./FCCM'96]

Example: FIR Filtering

$Y_i = w_1 x_i + w_2 x_{i+1} + \ldots$

Application metric:
TAPs = filter taps
multiply accumulate

Architecture	Feature Size (λ)	$\frac{TAPs}{\lambda^2 s}$
32b RISC	0.75μm	0.020
16b DSP	0.65μm	0.057
32b RISC/DSP	0.25μm	0.021
64b RISC	0.18μm	0.064
FPGA (XC4K)	0.60μm	1.9
(Altera 8K)	0.30μm	3.6
Full Custom	0.75μm	3.6
	0.60μm	3.5
	0.75μm	2.4
(fixed coefficient)	0.60μm	56
(*n.b.* 16b samples)		

Usage Classes

Specialization Usage Classes

- Known binding time
- Dynamic binding, persistent use
  - apparent
  - empirical
- Common case

Known Binding Time

- Sum=0
- For I=0→N
  Sum+=V[I]
- For I=0→N
  VN[I]=V[I]/Sum

- Scale(max,min,V)
  for I=0→V.length
    tmp=(V[I]-min)
    Vres[I]=tmp/(max-min)

Dynamic Binding Time

- cexp=0;
- For I=0→V.length
  - if (V[I].exp!=cexp)
    cexp=V[I].exp;
  - Vres[I]=
    V[I].mant<<cexp

- Thread 1:
  - a=src.read()
  - if (a.newavg())
    avg=a.avg()
- Thread 2:
  - v=data.read()
  - out.write(v/avg)

Empirical Binding

- Have to check if value changed
  - Checking value O(N) area [pattern match]
  - Interesting because computations
    - can be $O(2^N)$ [Day 9]
    - often greater area than pattern match
  - Also Rent's Rule:
    - Computation > linear in IO
    - $IO = C n^P \rightarrow n \propto IO^{(1/p)}$

Common/Exceptional Case

- For I=0→N
  - Sum+=V[I]
  - delta=V[I]-V[I-1]
  - SumSq+=V[I]*V[I]
  - ….
  - if (overflow)
    - ….
- For IB=0→N/B
  - For II= 0→B
    - I=II+IB
    - Sum+=V[I]
    - delta=V[I]-V[I-1]
    - SumSq+=V[I]*V[I]
    - ….
  - if (overflow)
    - ….

Binding Times

- Pre-fabrication
- Application/algorithm selection
- Compilation
- Installation
- Program startup (load time)
- Instantiation (`new` ...)
- Epochs
- Procedure
- Loop

Exploitation Patterns

- Full Specialization (Partial Evaluation)
  - May have to run (synth?) p&r at runtime
- Worst-case footprint
  - *e.g.* multiplier worst-case, avg., this case
- Constructive Instance Generator
- Range specialization (wide-word datapath)
  - data width
- Template
  - *e.g.* pattern match – only fillin LUT prog.

Opportunity Example

Bit Constancy Lattice

- binding time for bits of variables (storage-based)

CBD	…… Constant between definitions
SCBD	…… + signed
CSSI	…… Constant in some scope invocations
SCSSI	…… + signed
CESI	…… Constant in each scope invocation
SCESI	…… + signed
CASI	…… Constant across scope invocations
SCASI	…… + signed
CAPI	…… Constant across program invocations
const	…… declared const

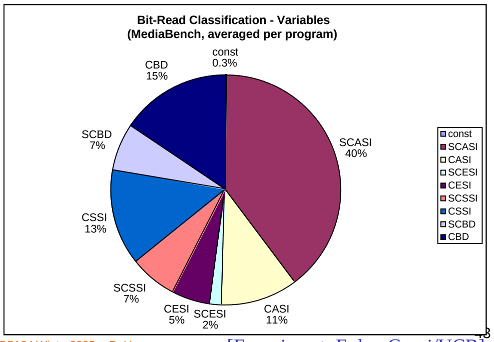
[Experiment: Eylon Caspi/UCB]

Experiments

- Applications:
  - UCLA MediaBench:
    adpcm, epic, g721, gsm, jpeg, mesa, mpeg2
    (not shown today - ghostscript, pegwit, pgp, rasta)
  - gzip, versatility, SPECint95 (parts)
- Compiler optimize→ instrument for profiling → run
- analyze variable usage, ignore heap
  - heap-reads typically 0-10% of all bit-reads
  - 90-10 rule (variables) - ~90% of bit reads in 1-20% or bits

[Experiment: Eylon Caspi/UCB]

Empirical Bit-Reads Classification



[Experiment: Eylon Caspi/UCB]

Bit-Reads Classification

- regular across programs
  - SCASI, CASI, CBD stddev ~11%
- nearly no activity in variables declared const
- ~65% in constant + signed bits
  - trivially exploited

[Experiment: Eylon Caspi/UCB]

Constant Bit-Ranges

- 32b data paths are too wide
- 55% of all bit-reads are to sign-bits
- most CASI reads clustered in bit-ranges (10% of 11%)
- CASI+SCASI reads (50%) are positioned:
  - 2% low-order 8% whole-word constant 39% high-order 1% elsewhere

[Experiment: Eylon Caspi/UCB]

Issue Roundup

Expression Patterns

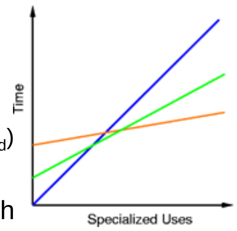
- Generators
- Instantiation/Immutable computations
  - (disallow mutation once created)
- Special methods (only allow mutation with)
- Data Flow (binding time apparent)
- Control Flow
  - (explicitly separate common/uncommon case)
- Empirical discovery

Benefits

- Much of the benefits come from reduced area
  - reduced area
    - room for more spatial operation
    - maybe less interconnect delay
- Fully exploiting, full specialization
  - don't know how big a block is until see values
  - dynamic resource scheduling

Optimization Prospects

- Area-Time Tradeoff
  - $T_{spcl} = T_{sc} + T_{load}$
  - $AT_{gen} = A_{gen} \times T_{gen}$
  - $AT_{spcl} = A_{spcl} \times (T_{sc} + T_{load})$
- If compute long enough
  - $T_{sc} >> T_{load} \rightarrow$ amortize out load



Storage

- Will have to store configurations somewhere
- LUT ~ $1M\lambda^2$
- Configuration 64+ bits
  - SRAM: $80K\lambda^2$ (12-13 for parity)
  - Dense DRAM: $6.4K\lambda^2$ (160 for parity)

Saving Instruction Storage

- Cache common, rest on alternate media
  - *e.g.* disk
- Compressed Descriptions
- Algorithmically composed descriptions
  - good for regular datapaths
  - think Kolmogorov complexity
- Compute values, fill in template
- Run-time configuration generation

Open

- How much opportunity exists in a given program?
- Can we measure entropy of programs?
  - How constant/predictable is the data compute on?
  - Maximum potential benefit if exploit?
  - Measure efficiency of architecture/implementation like measure efficiency of compressor?

Big Ideas

- Programmable advantage
  - Minimize work by specializing to instantaneous computing requirements
- Savings depends on functional complexity
  - but can be substantial for large blocks
  - close gap with custom?

Big Ideas

- Several models of structure
  - slow changing/early bound data, common case
- Several models of exploitation
  - template, range, bounds, full special