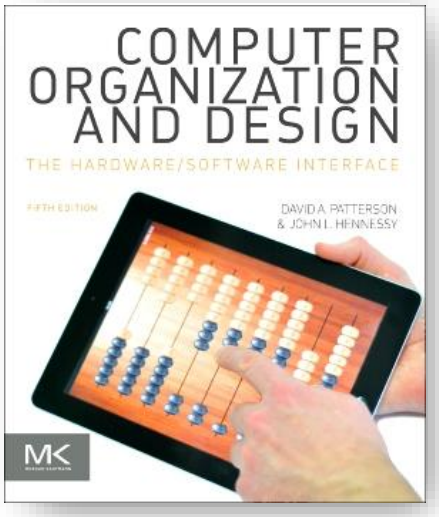


ESE 345 Computer Architecture

Design Process

The Design Process

"*To Design Is To Represent*"

Design activity yields description/representation of an object

-- Traditional craftsman does not distinguish between the conceptualization and the artifact

-- Separation comes about because of *complexity*

-- The concept is captured in one or more *representation languages*

-- This process IS design

Design Begins With Requirements

-- Functional Capabilities: what it will do

-- Performance Characteristics: Speed, Power, Area, Cost, . . .

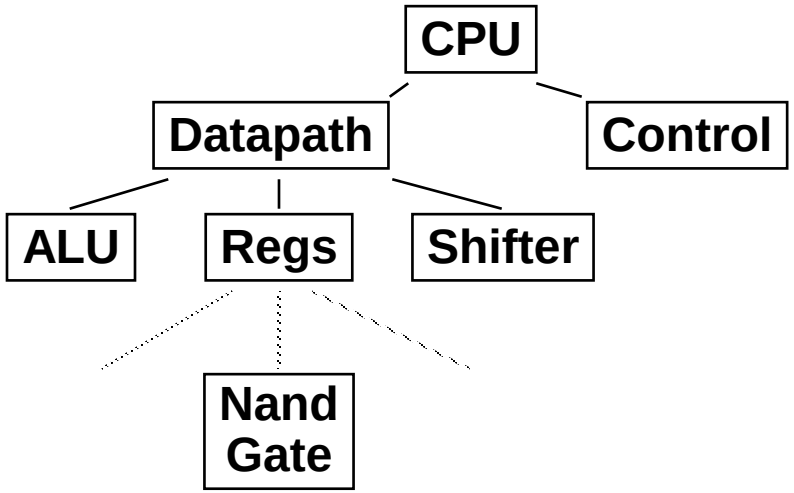
Design Process (cont.)

Design Finishes As Assembly

-- Design understood in terms of components and how they have been assembled

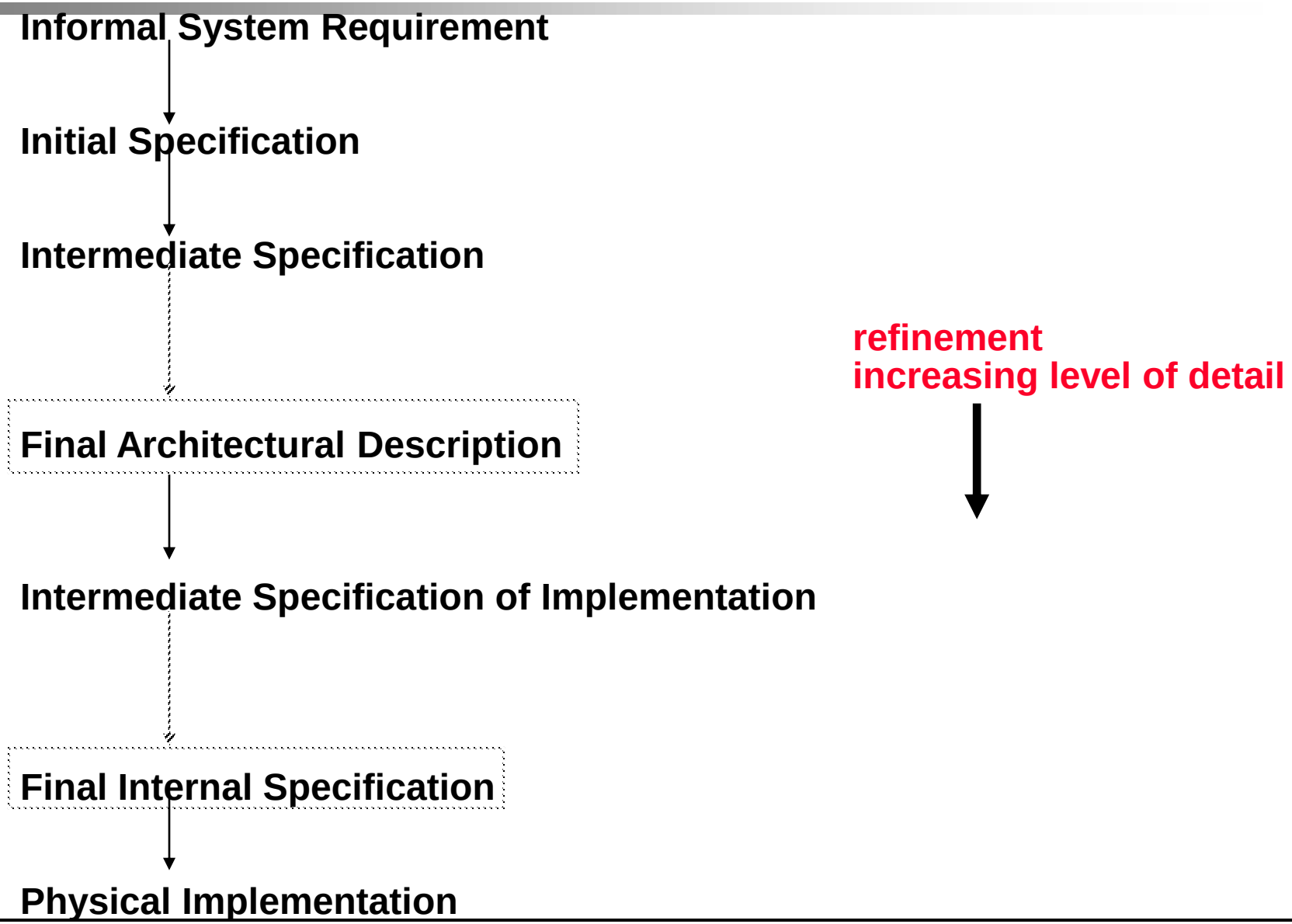
-- Top Down *decomposition* of complex functions (behaviors) into more primitive functions

-- bottom-up *composition* of primitive building blocks into more complex assemblies

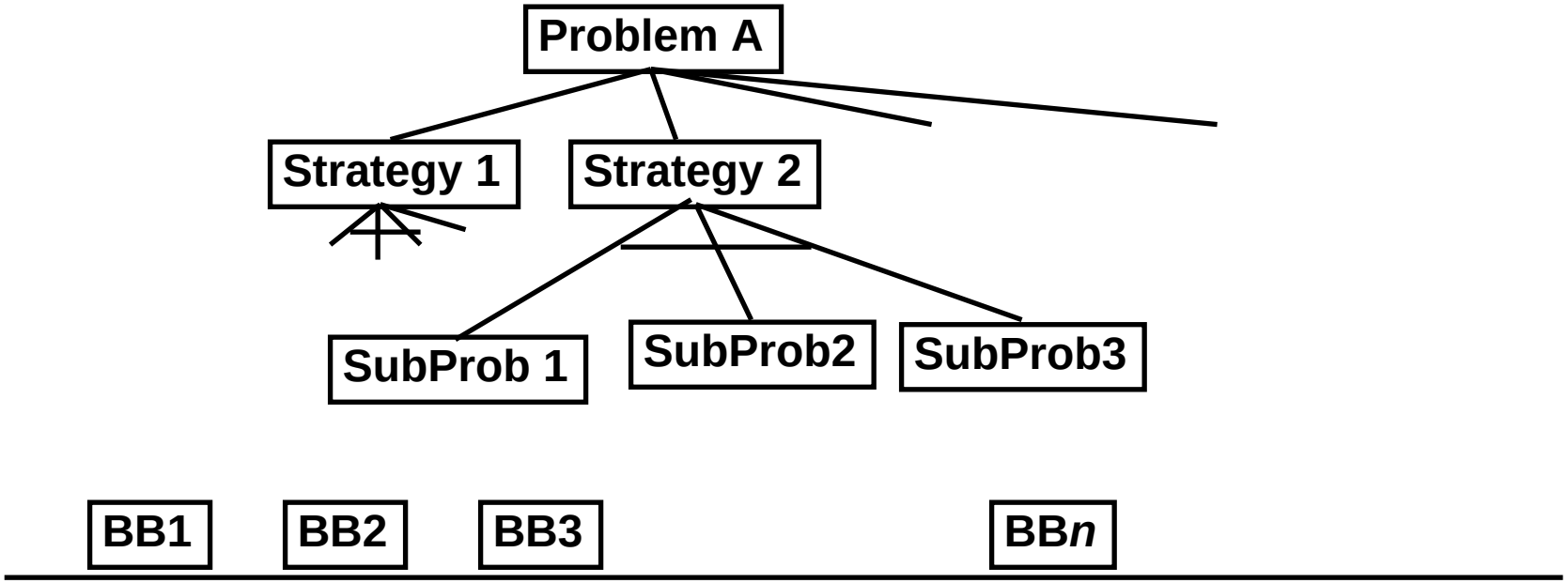


Design is a "creative process," not a simple method

Design Refinement



Design as Search



Design involves educated guesses and verification

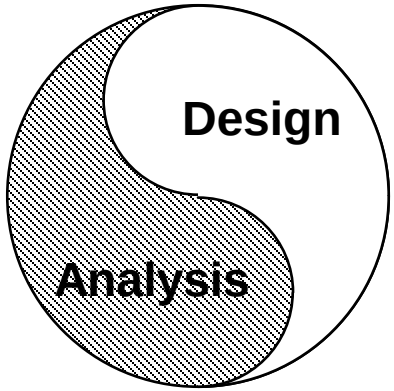
-- **Given the goals, how should these be prioritized?**

-- **Given alternative design pieces, which should be selected?**

-- **Given design space of components & assemblies, which part will yield the best solution?**

Feasible (good) choices vs. Optimal choices

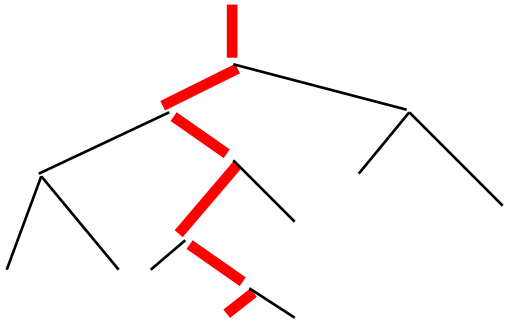
Measurement and Evaluation

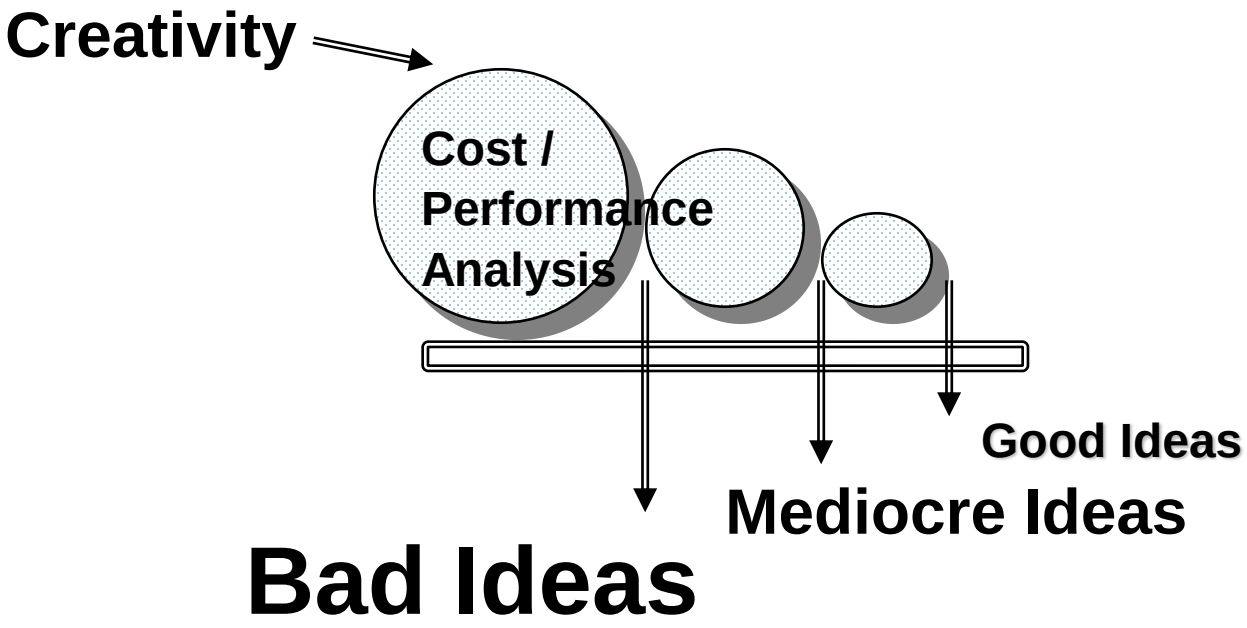


Architecture is an iterative process

-- searching the space of possible designs

-- at all levels of computer systems





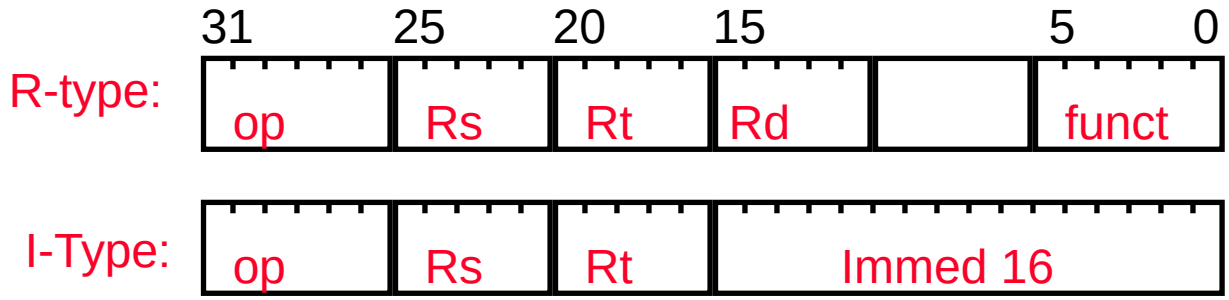
Problem: Design a “fast” ALU for the MIPS ISA

- Requirements?
- Must support the MIPS ISA
  Arithmetic / Logic operations
- Tradeoffs of cost and speed based on frequency of occurrence, hardware budget

MIPS ALU Requirements

- Add, AddU, Sub, SubU, AddI, AddIU
  - => 2's complement adder/sub with overflow detection
- And, Or, AndI, OrI, Xor, Xori, Nor
  - => Logical AND, logical OR, XOR, nor
- SLTI, SLTIU (set less than)
  - => 2's complement adder with inverter, check sign bit of result

MIPS arithmetic instruction format



Type	op	funct
ADDI	10	xx
ADDIU	11	xx
SLTI	12	xx
SLTIU	13	xx
ANDI	14	xx
ORI	15	xx
XORI	16	xx
LUI	17	xx

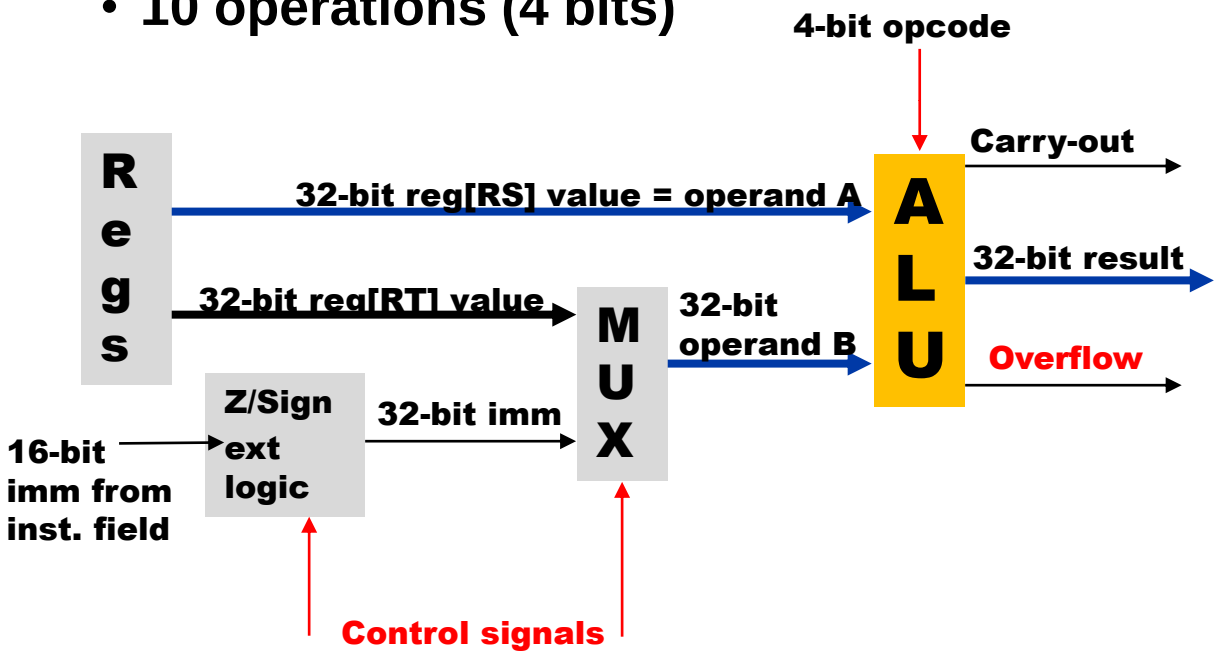
Type	op	funct
ADD	00	40
ADDU	00	41
SUB	00	42
SUBU	00	43
AND	00	44
OR	00	45
XOR	00	46
NOR	00	47

Type	op	funct
	00	50
	00	51
SLT	00	52
SLTU	00	53

- Signed arithmetic generate overflow, no carry

Design Trick: divide & conquer

- Trick #1: Break the problem into simpler problems, solve them and glue together the solution
- Example: assume the immediates have been taken care of before the ALU
  - 10 operations (4 bits)



00	add
01	addU
02	sub
03	subU
04	and
05	or
06	xor
07	nor
12	slt
13	sltU

Refined Requirements

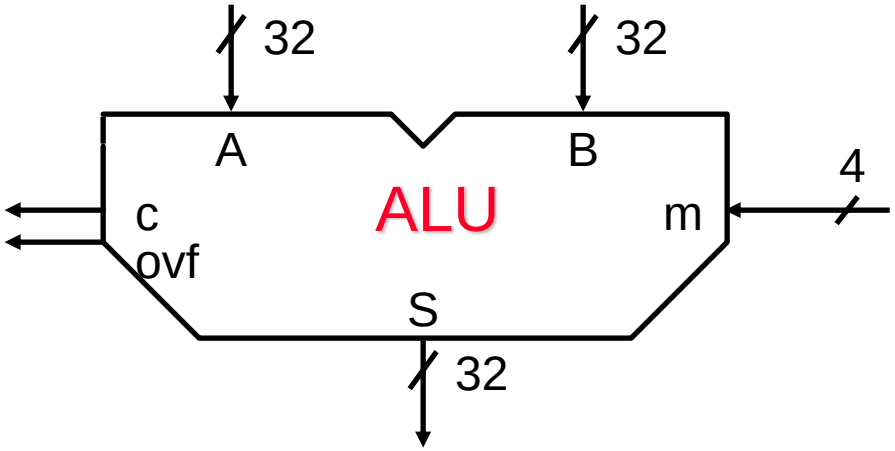
(1) Functional Specification

inputs: 2 x 32-bit operands A, B, 4-bit mode

outputs: 32-bit result S, 1-bit carry, 1 bit overflow

operations: add, addu, sub, subu, and, or, xor, nor, slt, sltU

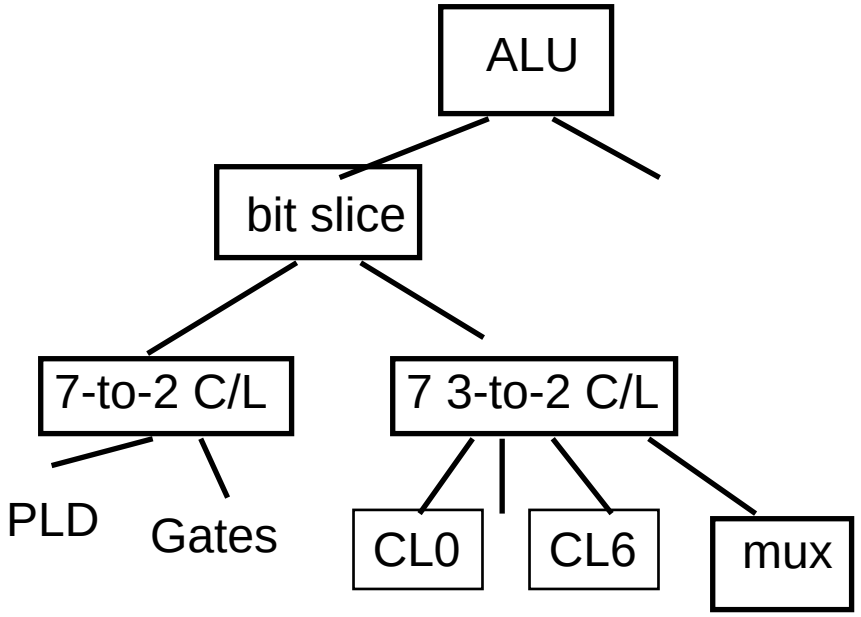
(2) Block Diagram (schematic symbol, Verilog description)



Behavioral Representation: Verilog

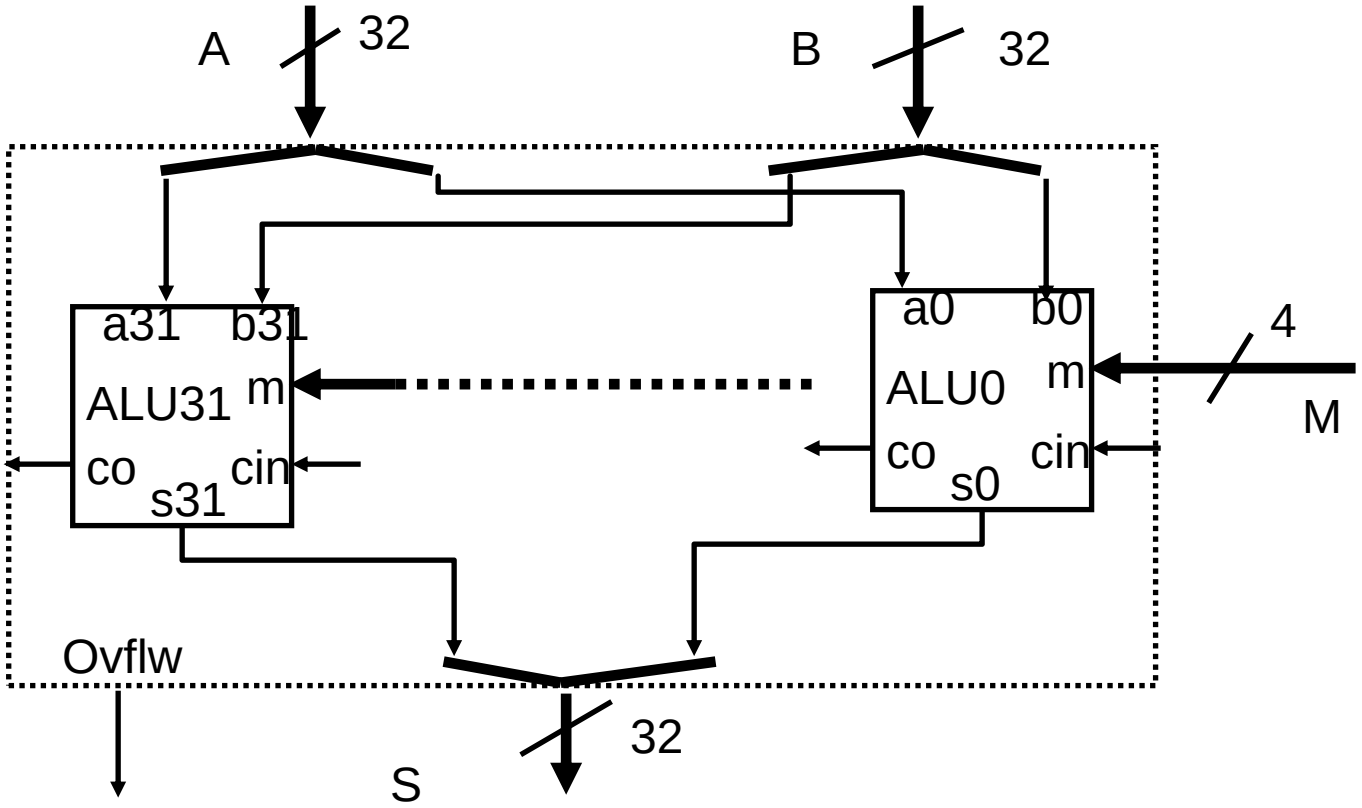
```
module ALU(A, B, m, S, c, ovf);
input [0:31] A, B;
input [0:3] m;
output [0:31] S;
output c, ovf;

reg [0:31] S;
reg c, ovf;

always @(A, B, m) begin
   case (m)
       0: S = A + B;

   . . .

end
endmodule
```

Design Decisions



- Simple bit-slice
  - big combinational problem
  - many little combinational problems
  - partition into 2-step problem
- Bit slice with carry look-ahead
- . . .

Refined Diagram: bit-slice ALU



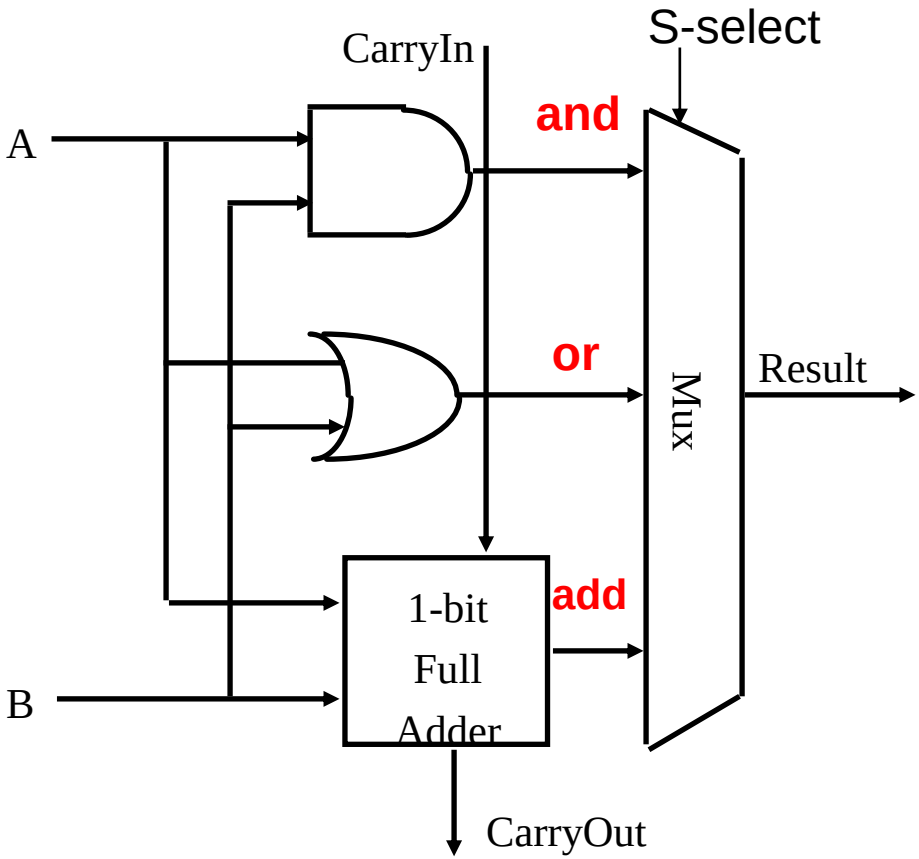
7-to-2 Combinational Logic

- start turning the crank . . .

	Function	Inputs							Outputs		K-Map
		M0	M1	M2	M3	A	B	Cin	S	Cout	
0	add	0	0	0	0	0	0	0	0	0	
127											

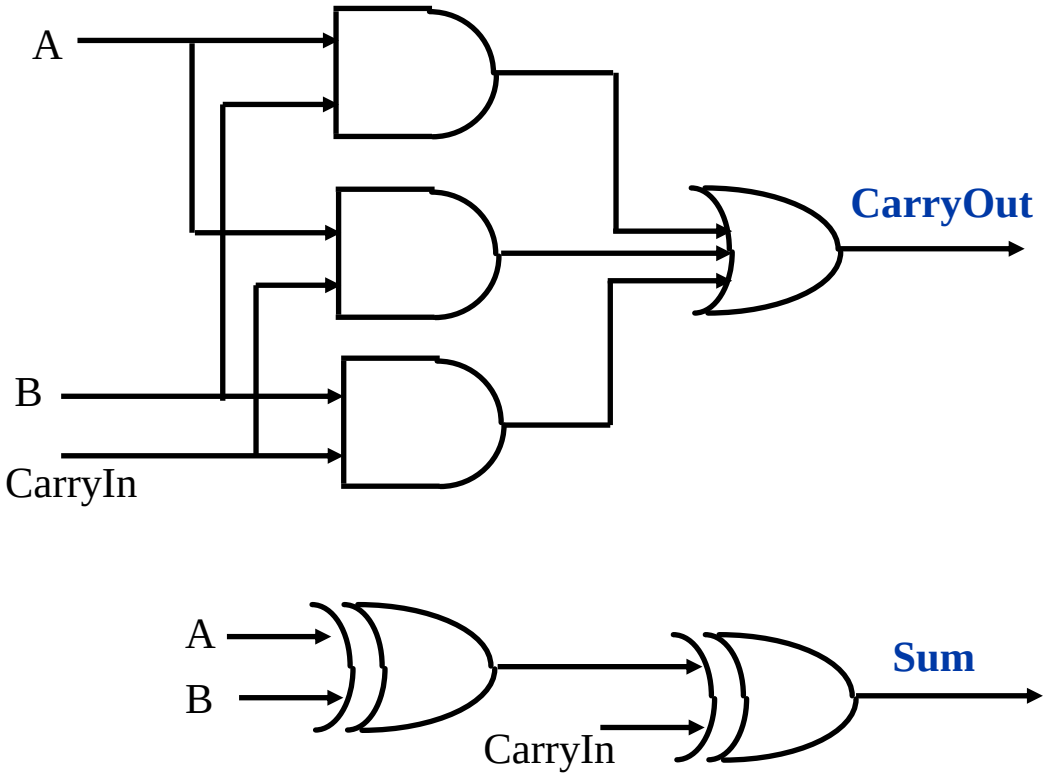
Seven plus a MUX ?

- Design trick 2: take pieces you know (or can imagine) and try to put them together
- Design trick 3: solve part of the problem and extend



1-bit Full Adder

- We'll assume the same gate delay for the AND, OR, and XOR gates
- 1-bit full adder latency = FA critical path (CP) time = 2 gate delays



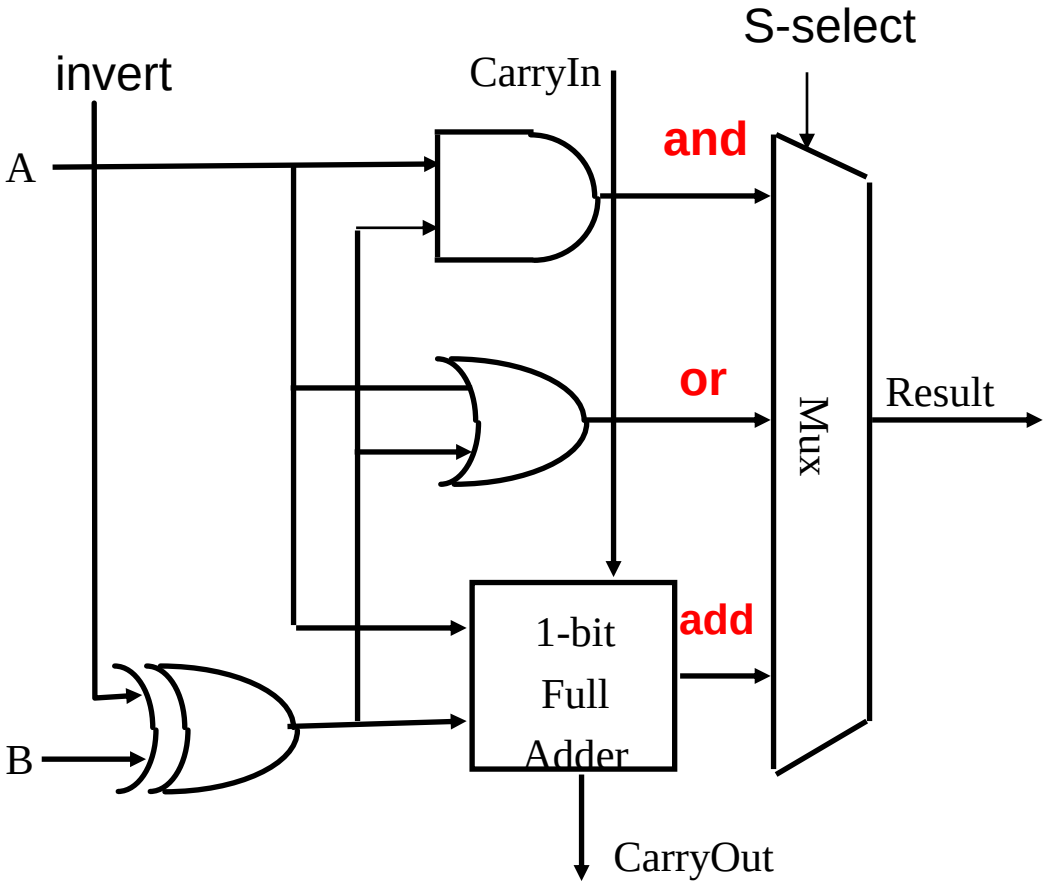
Full Adder

(3->2 element)

A	B	C in	C out	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Additional operations

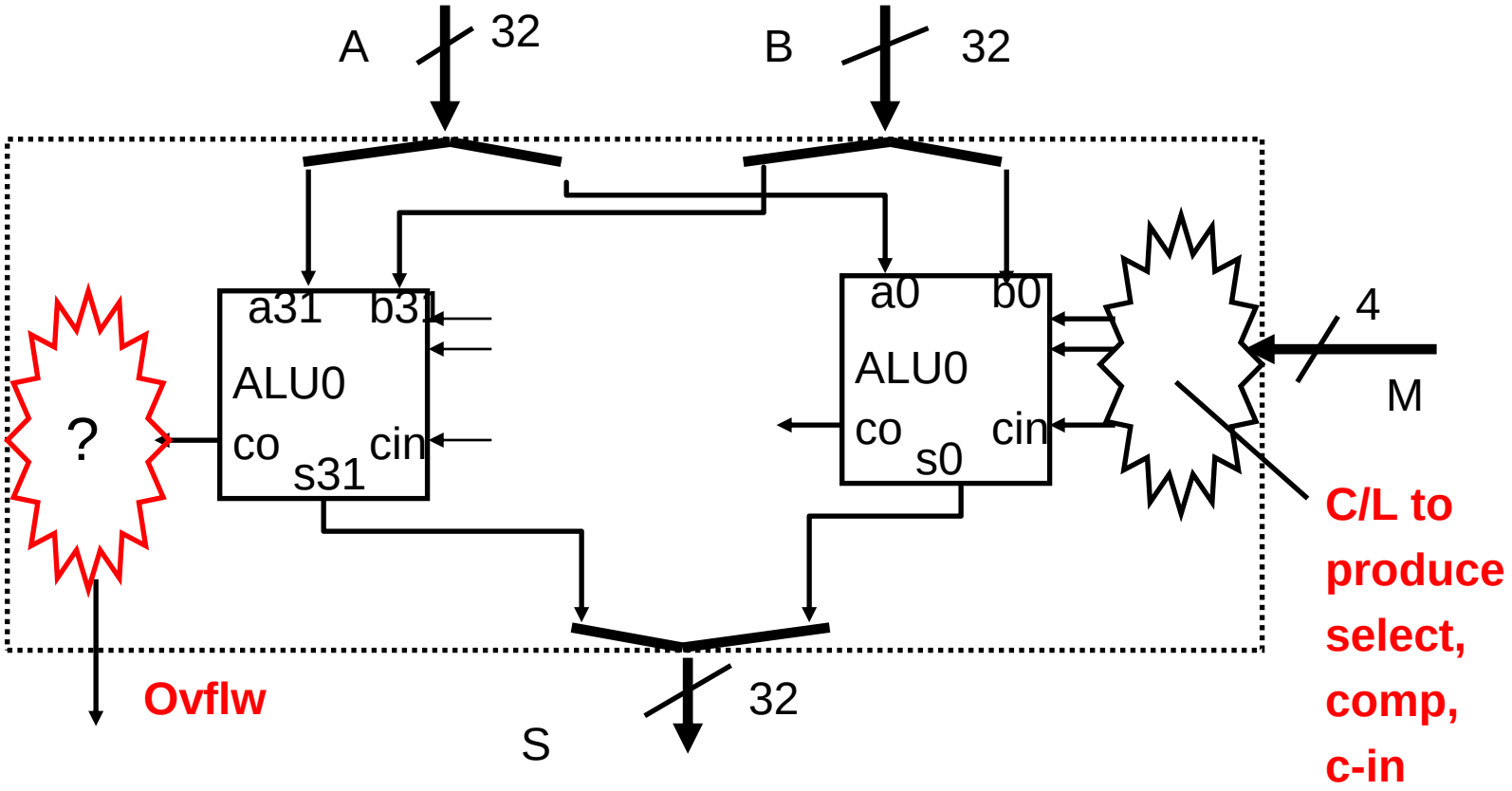
- **A - B = A + (– B) = A + $\overline{B}$ + 1**
  - form two complement by invert and add one



Set-less-than? – left as an exercise

Revised Diagram

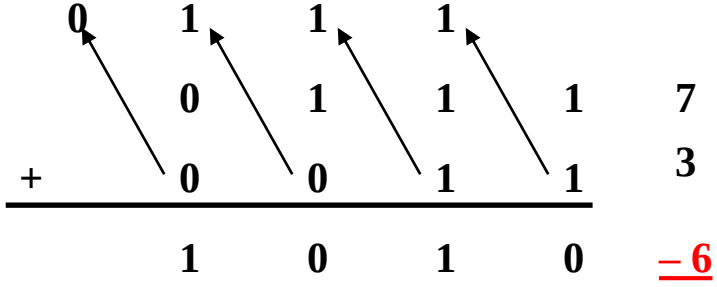
- LSB and MSB need to do a little extra



Overflow

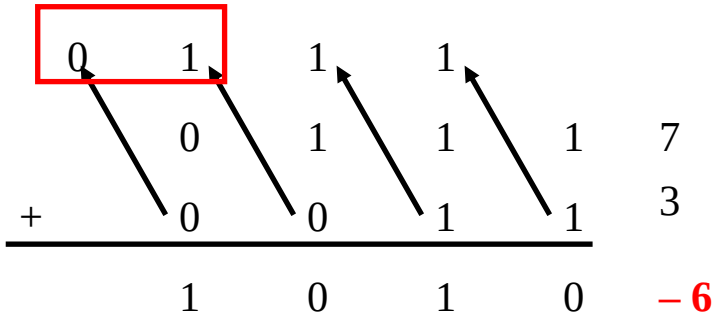
Decimal	Binary	Decimal	2's Complement
0	0000	0	0000
1	0001	-1	1111
2	0010	-2	1110
3	0011	-3	1101
4	0100	-4	1100
5	0101	-5	1011
6	0110	-6	1010
7	0111	-7	1001
		-8	1000

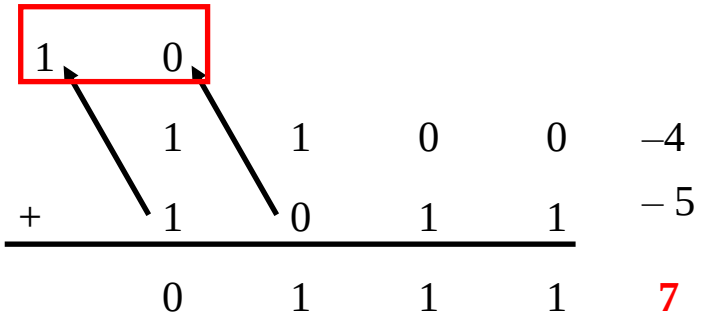
- Examples: 7 + 3 = 10 but ...
- - 4 - 5 = - 9 but ...



Overflow Detection

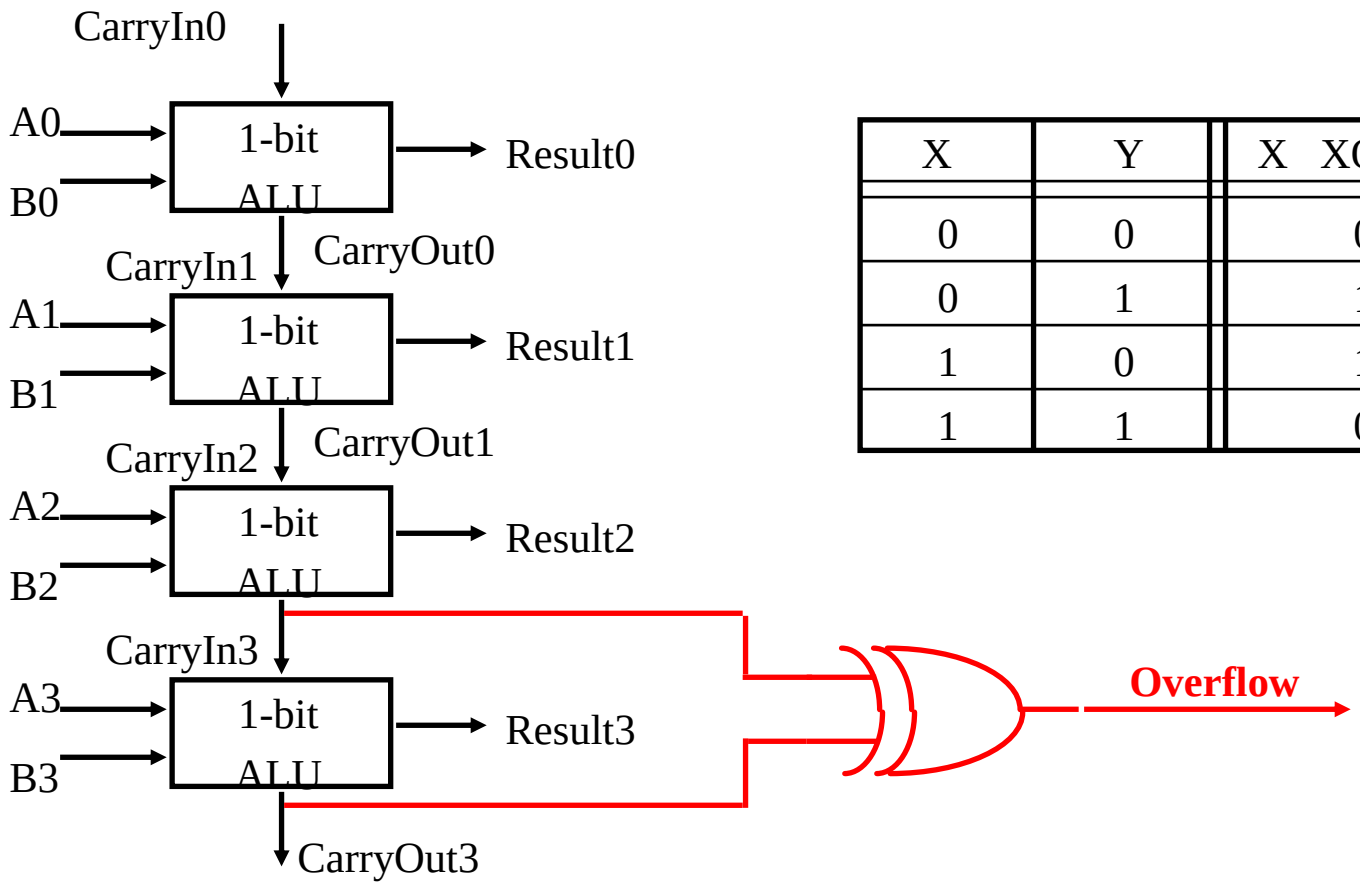
- **Overflow: the result is too large (or too small) to represent properly**
  - **Example: - 8 ≤ 4-bit binary number ≤ 7**
- **When adding operands with different signs, overflow cannot occur!**
- **Overflow occurs when adding:**
  - **2 positive numbers and the sum is negative**
  - **2 negative numbers and the sum is positive**
- **On your own: Prove you can detect overflow by:**
  - **Carry into MSB ≠ Carry out of MSB**





Overflow Detection Logic

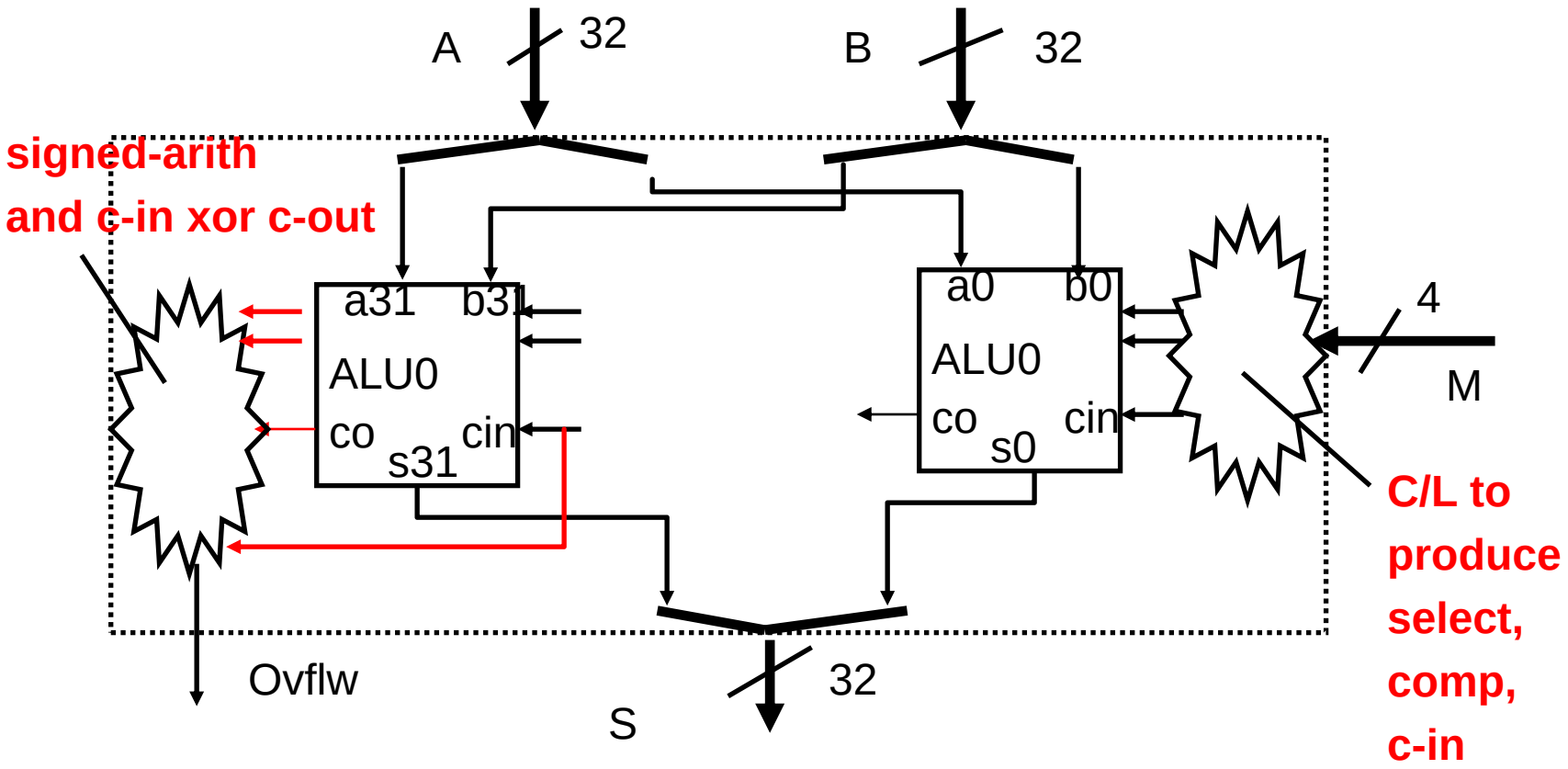
- **Carry into MSB ≠ Carry out of MSB**
  - **For a N-bit ALU: Overflow = CarryIn[N - 1] XOR CarryOut[N - 1]**



X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

More Revised Diagram

- **LSB and MSB need to do a little extra**

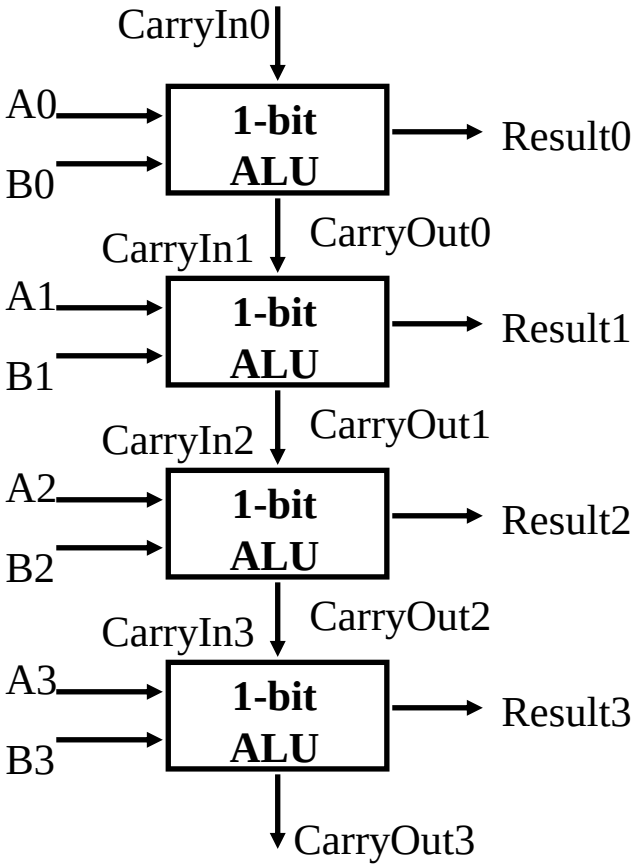


Which is good design advice?

1. **Wait until you know everything before you start (“Be prepared”)**
2. **The best design is a one-pass, top down process (“Plan Ahead”)**
3. **Start simple, measure, then optimize (“Less is more”)**
4. **Don’t be biased by the components you already know (“Start with a clean slate”)**

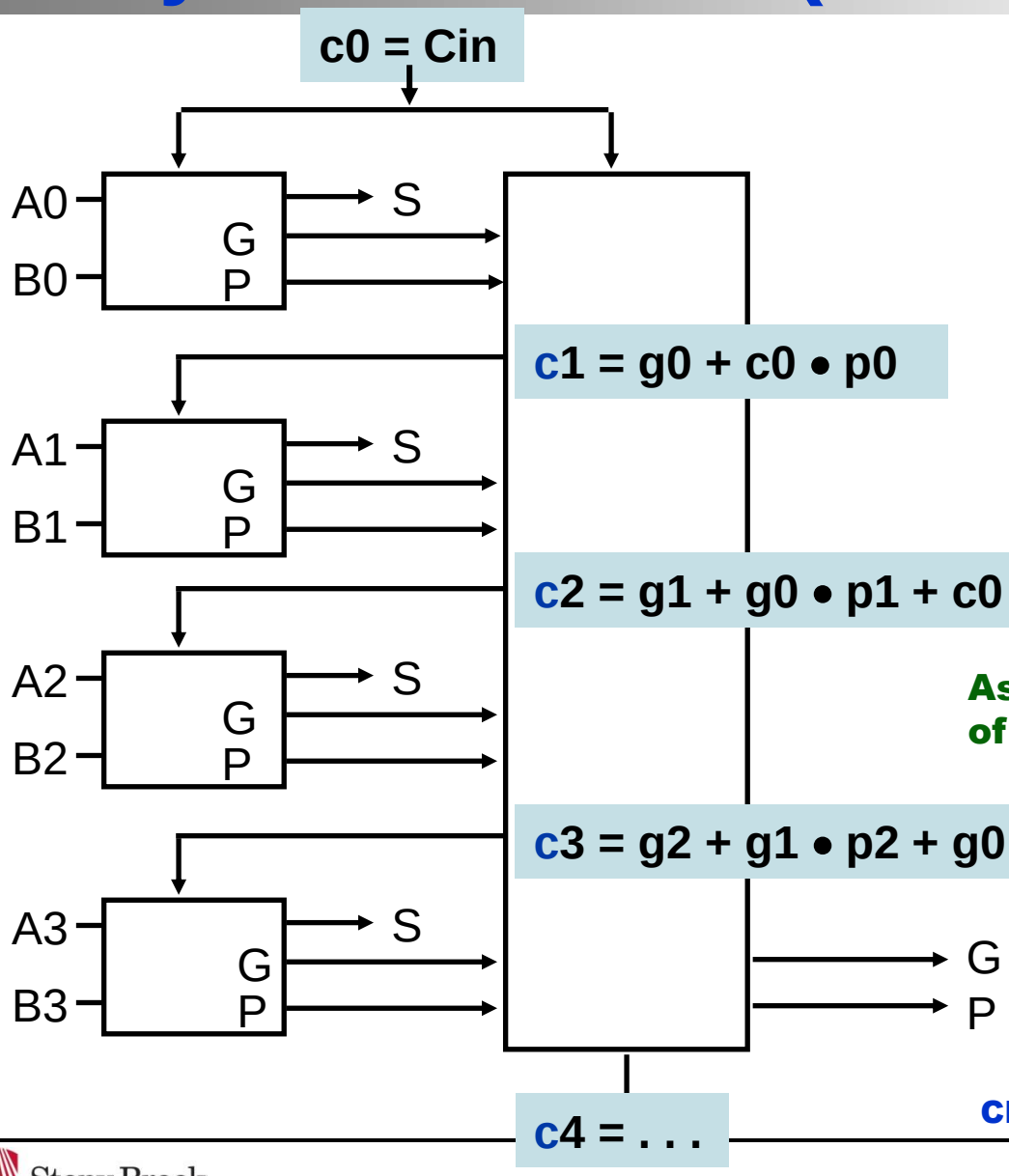
But What about Performance?

- **Critical Path of n-bit Ripple-carry adder (RCA) is n*Critical Path (CP) time of a 1-bit adder**
  - **a 4-bit RCA CP latency (time to produce a result) = 4 x 2 = 8 gate delays**



Design Trick:
Throw hardware at it

Carry Look Ahead (CLA) Adder



A	B	C-out	
0	0	0	"kill"
0	1	C-in	"propagate"
1	0	C-in	"propagate"
1	1	1	"generate"

g(i) = a(i) and b(i)
p(i) = a(i) xor b(i)

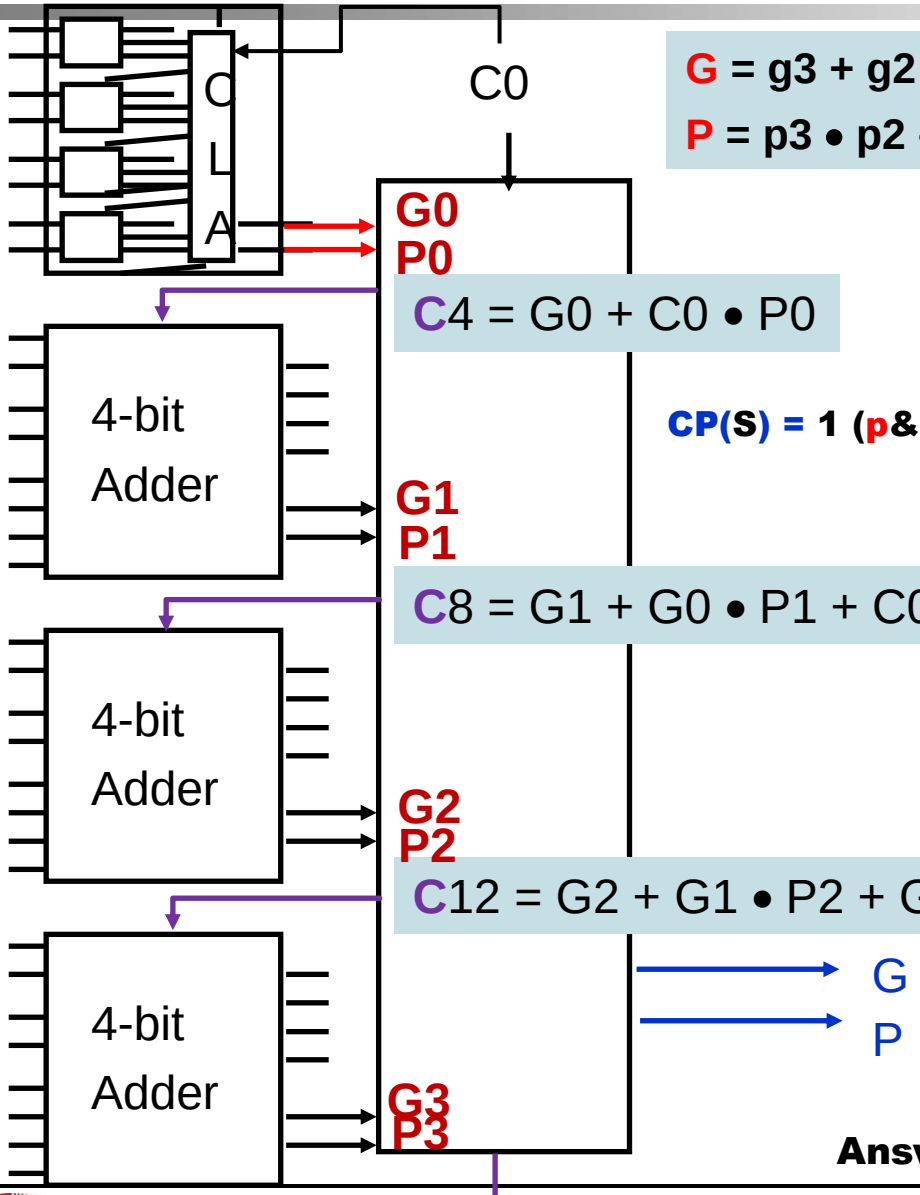
Assume that the maximum number of OR/AND inputs/terms = 5

What about a 4-bit CLA CP time to produce sum S and Cout(c4) in gate delays (GDs)?

CP(c4) = 1 (p&g) + 2 (c) = 3 GDs

CP(S) = 1 (p&g) + 2 (c) + 1 (xor in FA) = 4 GDs

Cascaded Carry Look-ahead (16-bit): Abstraction



G = g3 + g2 • p3 + g1 • p3 • p2 + g0 • p3 • p2 • p1

P = p3 • p2 • p1 • p0

What about a 16-bit CLA CP time to produce sum S and Cout in gate delays (GDs)?

CP(C16) = 1 (p&g) + 2 (G) + 2 (C) = 5 GDs

CP(S) = 1 (p&g) + 2 (G) + 2 (C) + 2 (c) + 1 (xor in FA) = 8 GDs

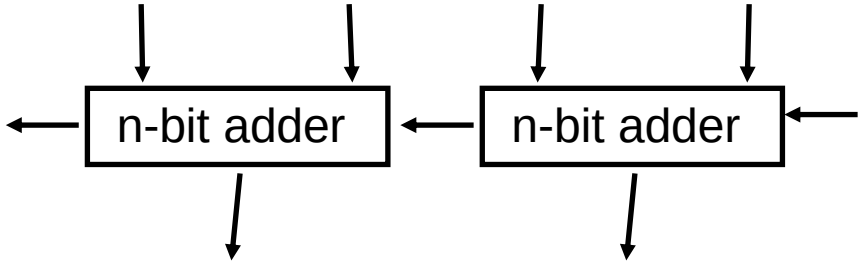
The textbook answer (5 GDs) to the question on CP(S) (B-46) is wrong because it does not take into account the contribution of the last two items in the CP(S) equation!

What about the performance of a 16-bit CLA vs. a 16-bit RCA?

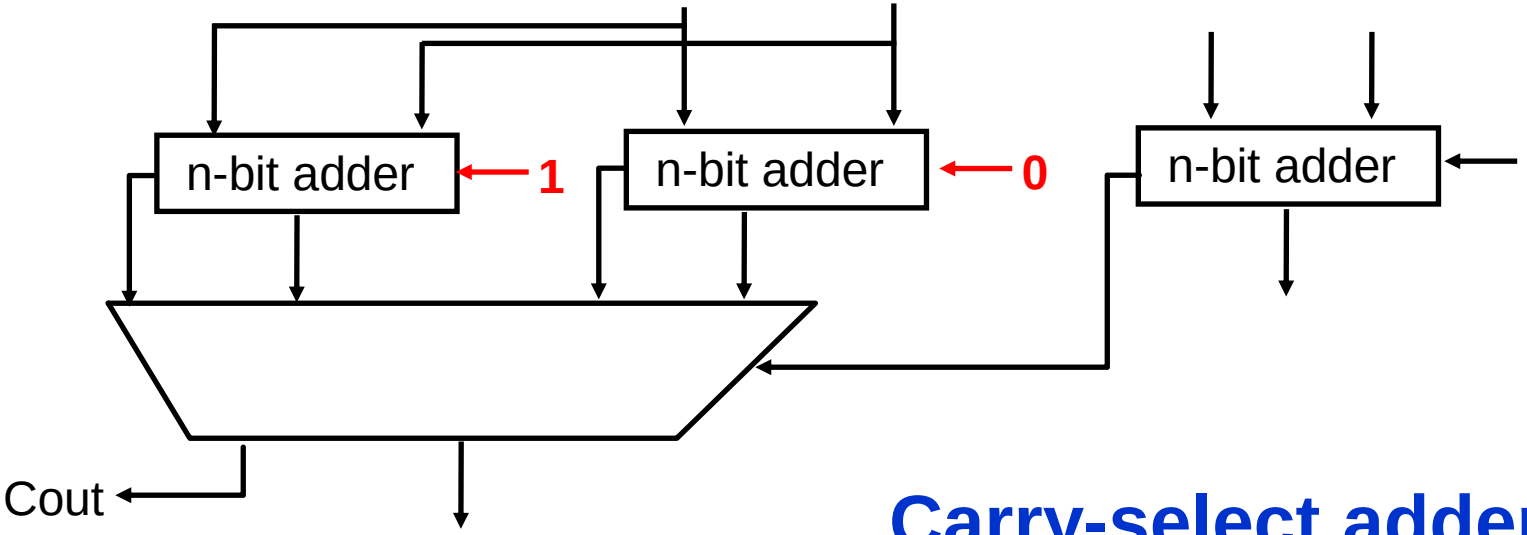
Answer: 32 GDs/8 GDs = > CLA is 4 times faster

Design Trick: Guess (or "Precompute")

CP(2n) = 2*CP(n)

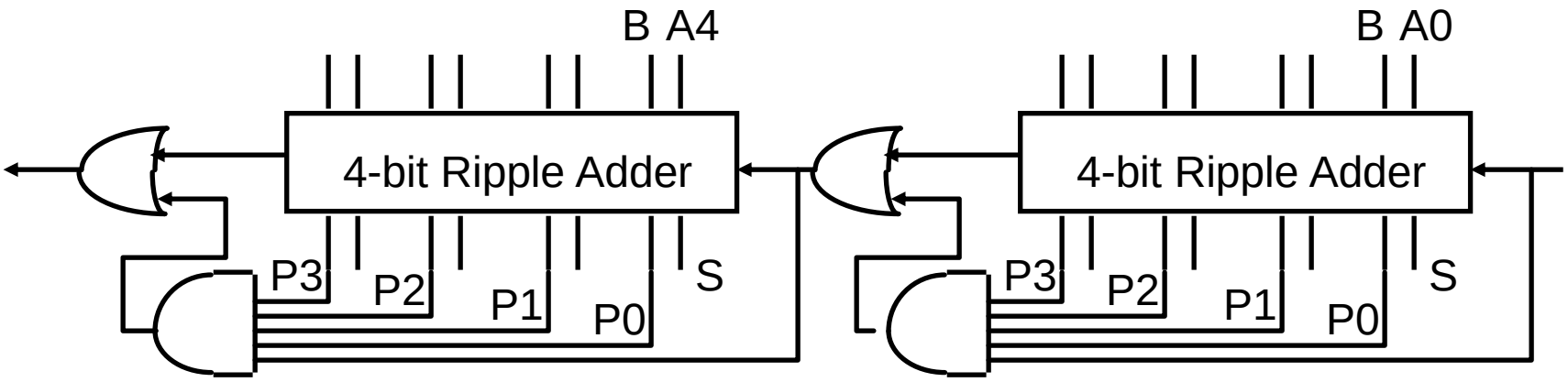


CP(2n) = CP(n) + CP(mux)



Carry-select adder

Carry Skip Adder: reduce worst case delay



Just speed up the slowest case for each block

Exercise: optimal design uses variable block sizes

Additional MIPS ALU requirements

- Mult, MultU, Div, DivU => Need 32-bit multiply and divide, signed and unsigned
- Sll, Srl, Sra => Need left shift, right shift, right shift arithmetic by 0 to 31 bits

Elements of the Design Process

- **Divide and Conquer (e.g., ALU)**
  - Formulate a solution in terms of simpler components.
  - Design each of the components (subproblems)
- **Generate and Test (e.g., ALU)**
  - Given a collection of building blocks, look for ways of putting them together that meets requirement
- **Successive Refinement (e.g., carry lookahead)**
  - Solve "most" of the problem (i.e., ignore some constraints or special cases), examine and correct shortcomings.
- **Formulate High-Level Alternatives (e.g., carry select)**
  - Articulate many strategies to "keep in mind" while pursuing any one approach.
- **Work on the Things you Know How to Do**
  - The unknown will become “obvious” as you make progress.

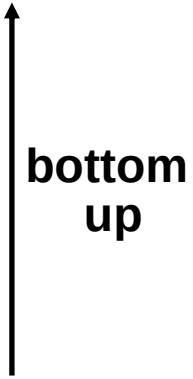
Summary of the Design Process

Hierarchical Design to manage complexity

Top Down vs. Bottom Up vs. Successive Refinement

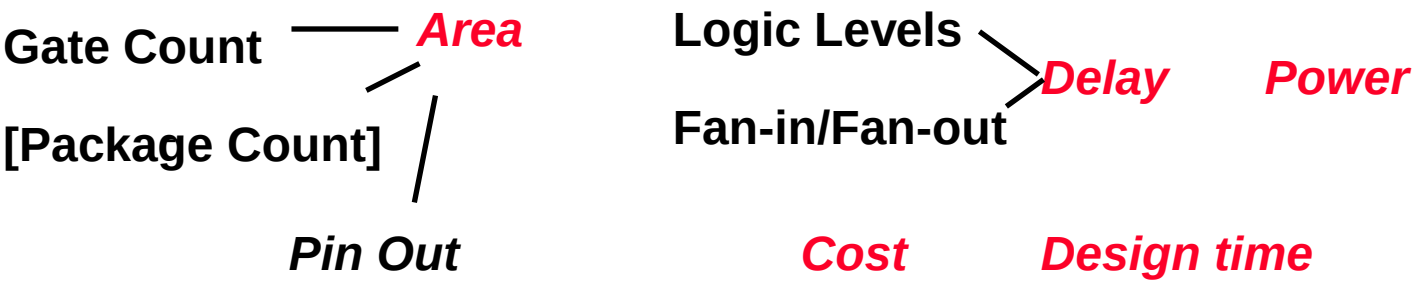
Importance of Design Representations:

- **Block Diagrams**
- **Decomposition into Bit Slices**
- **Truth Tables, K-Maps**
- **Circuit Diagrams**



- **Other Descriptions: state diagrams, timing diagrams, reg xfer, . . .**

Optimization Criteria:



Match for Design Principle?

I. Composition

II. Divide and Conquer

III. Start simple, then optimize critical paths

A. Design 1-bit ALU slice before 32-bit ALU

B. Replace ripple carry with carry lookahead

C. Use Mux to join AND, OR gates with Adder

Best match	I.	II.	III.
1.	A	B	C
2.	A	C	B
3.	B	A	C
4.	B	C	A
5.	C	A	B
6.	C	B	A

Summary

- An Overview of the Design Process
  - Design is an iterative process, multiple approaches to get started
  - Do NOT wait until you know everything before you start
- Example: Instruction Set drives the ALU design
  - Divide and Conquer
  - Take pieces you know and put them together
  - Start with a partial solution and extend
- Optimization: Start simple and analyze critical path
  - For adder: the carry is the slowest element
  - Logarithmic trees to flatten linear computation
  - Precompute: Double hardware and postpone slow decision

Acknowledgements

- These slides contain material developed and copyright by:
  - Morgan Kauffmann (Elsevier, Inc.)
  - David Patterson (UCB)
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)