Computer Architecture

EECS 361
Lecture 5: The Design Process & ALU Design

Quick Review of Last Lecture

MIPS ISA Design Objectives and Implications

- **Support general OS and C-style language needs**
- **Support general and embedded applications**
- **Use dynamic workload characteristics from general purpose program traces and SPECint to guide design decisions**

Traditional data types, common operations, typical addressing modes

- **Implement processsor core with a relatively small number of gates**
- **Emphasize performance via fast clock**

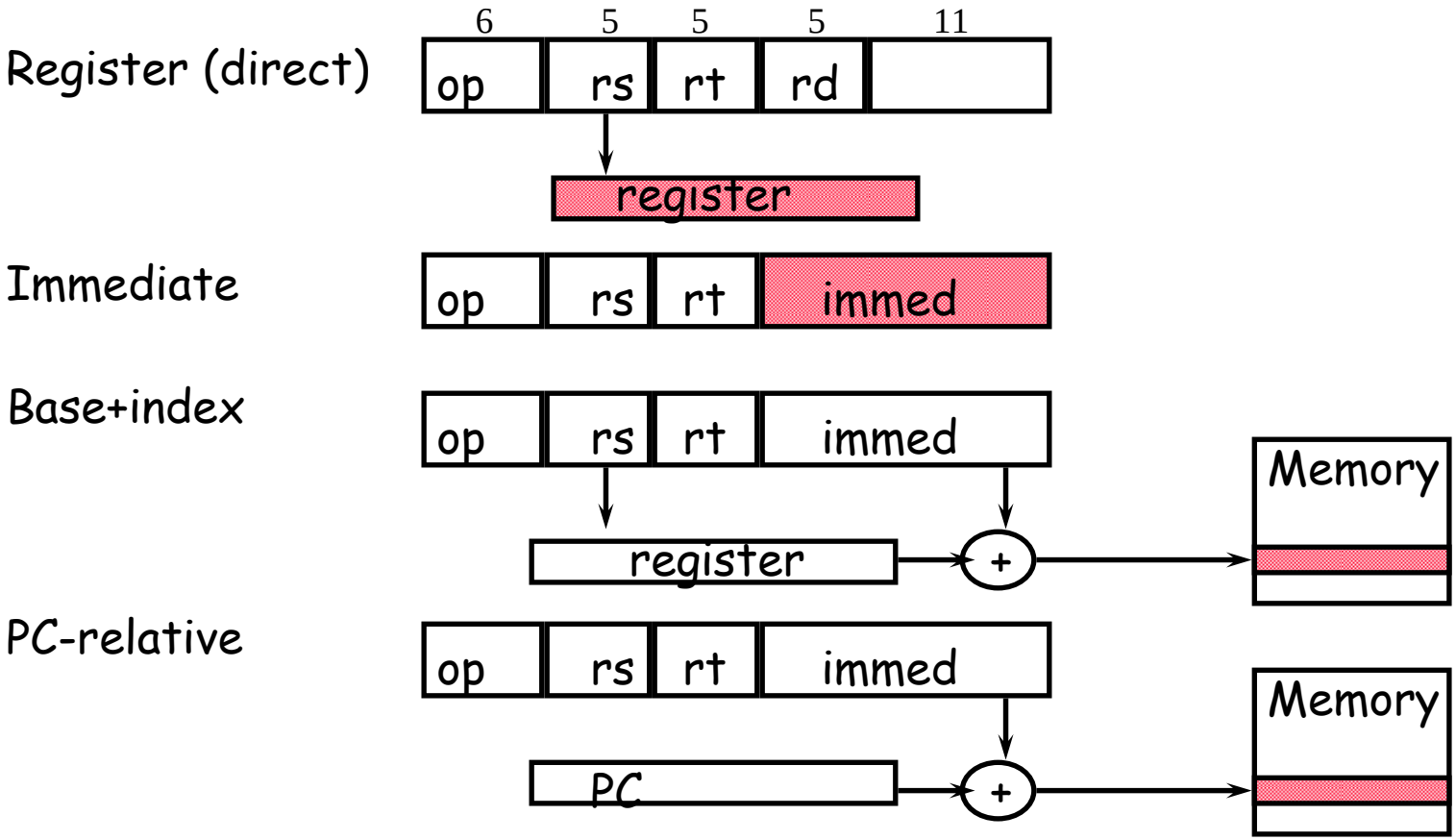
RISC-style: Register-Register / Load-Store

MIPS jump, branch, compare instructions

Instruction	*Example*	*Meaning*
branch on equal	beq $1,$2,100	if ($1 == $2) go to PC+4+100
	Equal test; PC relative branch	
branch on not eq.	bne $1,$2,100	if ($1!= $2) go to PC+4+100
	Not equal test; PC relative	
set on less than	slt $1,$2,$3	if ($2 < $3) $1=1; else $1=0
	Compare less than; 2's comp.	
set less than imm.	slti $1,$2,100	if ($2 < 100) $1=1; else $1=0
	Compare < constant; 2's comp.	
set less than uns.	sltu $1,$2,$3	if ($2 < $3) $1=1; else $1=0
	Compare less than; natural numbers	
set l. t. imm. uns.	sltiu $1,$2,100	if ($2 < 100) $1=1; else $1=0
	Compare < constant; natural numbers	
jump	j 10000	go to 10000
	Jump to target address	
jump register	jr $31	go to $31
	For switch, procedure return	
jump and link	jal 10000	$31 = PC + 4; go to 10000
	For procedure call	

Example: MIPS Instruction Formats and Addressing Modes

- All instructions 32 bits wide



MIPS Instruction Formats

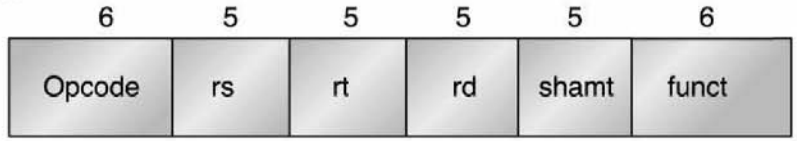
- Fixed instruction size: 4 bytes
- I-type:
  - rt <=> Memory [rs + IMM]
  - rt <= rs op IMM
  - if (rs == 0) PC += IMM
  - [r31 = PC+4] PC <= rs1
- R-type
  - rd <= rs op rt
- J-type
  - PC += Offset
  - r31 <= PC+4; PC += Offset

I-type instruction

Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt ← rs op immediate)

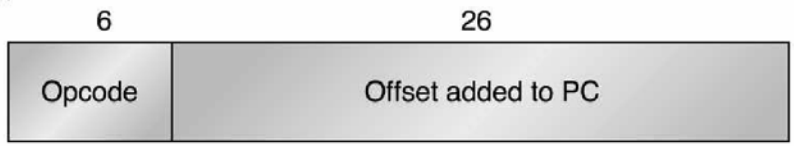
Conditional branch instructions (rs is register, rd unused)
Jump register, jump and link register
(rd = 0, rs = destination, immediate = 0)

R-type instruction



Register-register ALU operations: rd ← rs funct rt
Function encodes the data path operation: Add, Sub, . . .
Read/write special registers and moves

J-type instruction

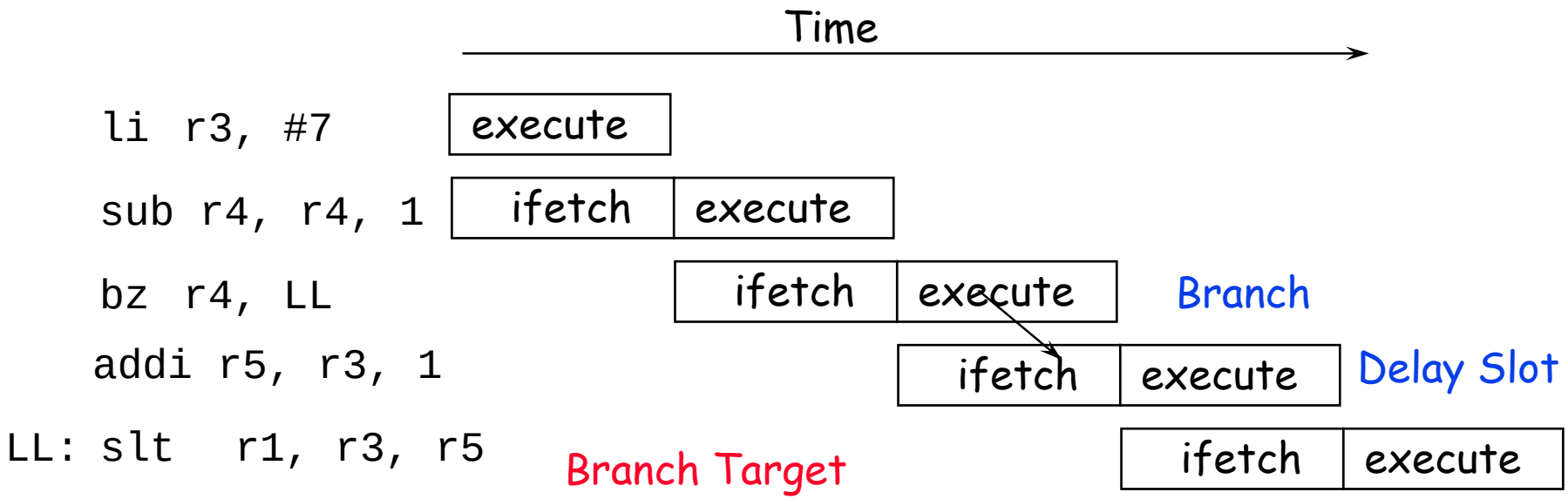


Jump and jump and link
Trap and return from exception

MIPS Operation Overview

- Arithmetic logical
- Add, AddU, AddI, ADDIU, Sub, SubU
- And, AndI, Or, OrI
- SLT, SLTI, SLTU, SLTIU
- SLL, SRL
- Memory Access
- LW, LB, LBU
- SW, SB

Branch & Pipelines

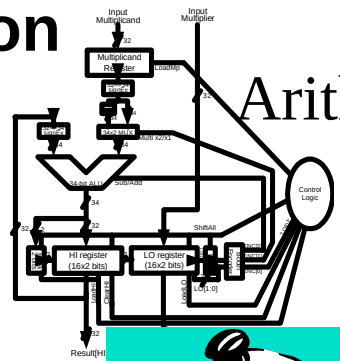


By the end of Branch instruction, the CPU knows whether or not the branch will take place.

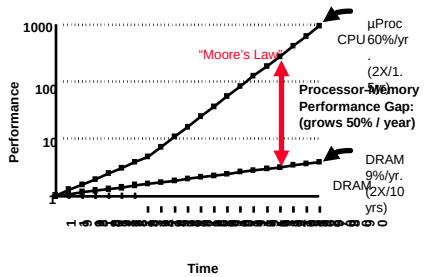
However, it will have fetched the next instruction by then, regardless of whether or not a branch will be taken.

Why not execute it?

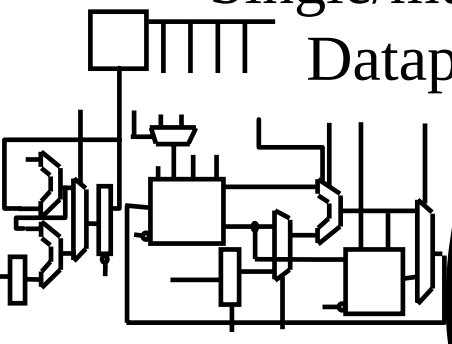
The next Destination



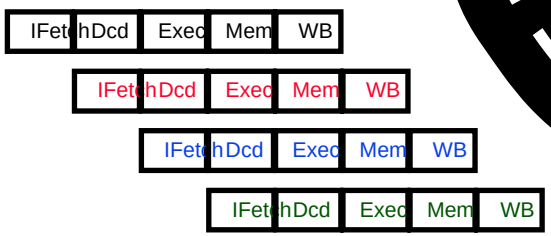
Arithmetic



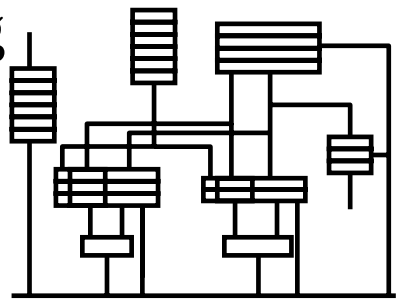
Single/multicycle Datapaths

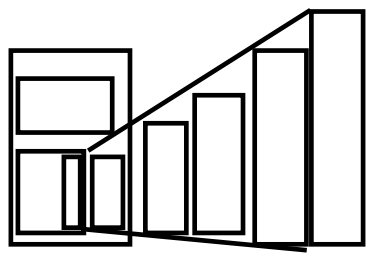


Begin ALU design using MIPS ISA.

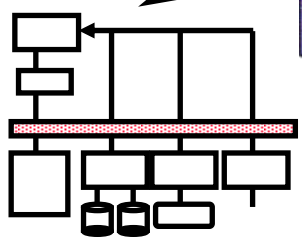


Pipelining

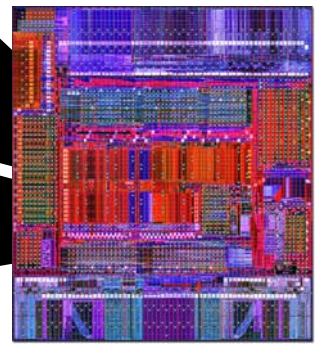




Memory Systems



I/O



Outline of Today's Lecture

- An Overview of the Design Process
- Illustration using ALU design
- Refinements

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

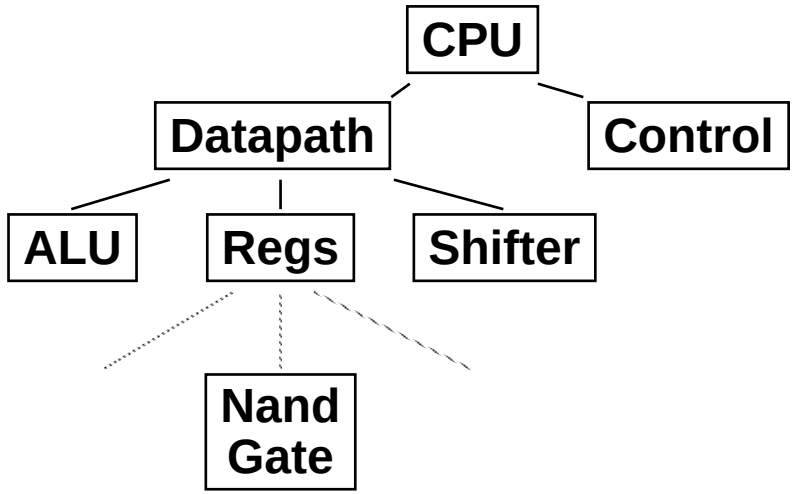
Design Begins With Requirements

-- Functional Capabilities: what it will do

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

Design Process

Design Finishes As Assembly



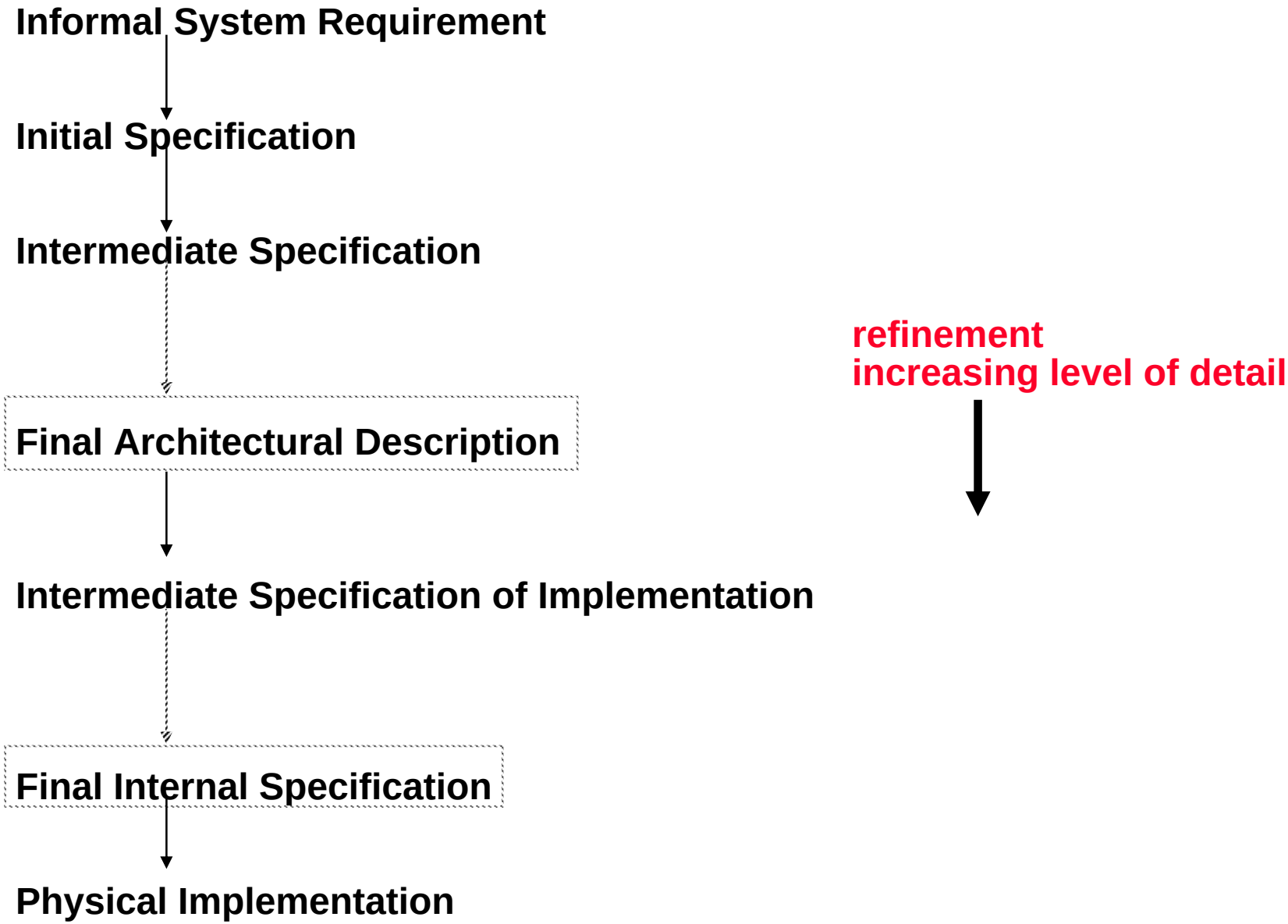
-- Design understood in terms of components and how they have been assembled

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

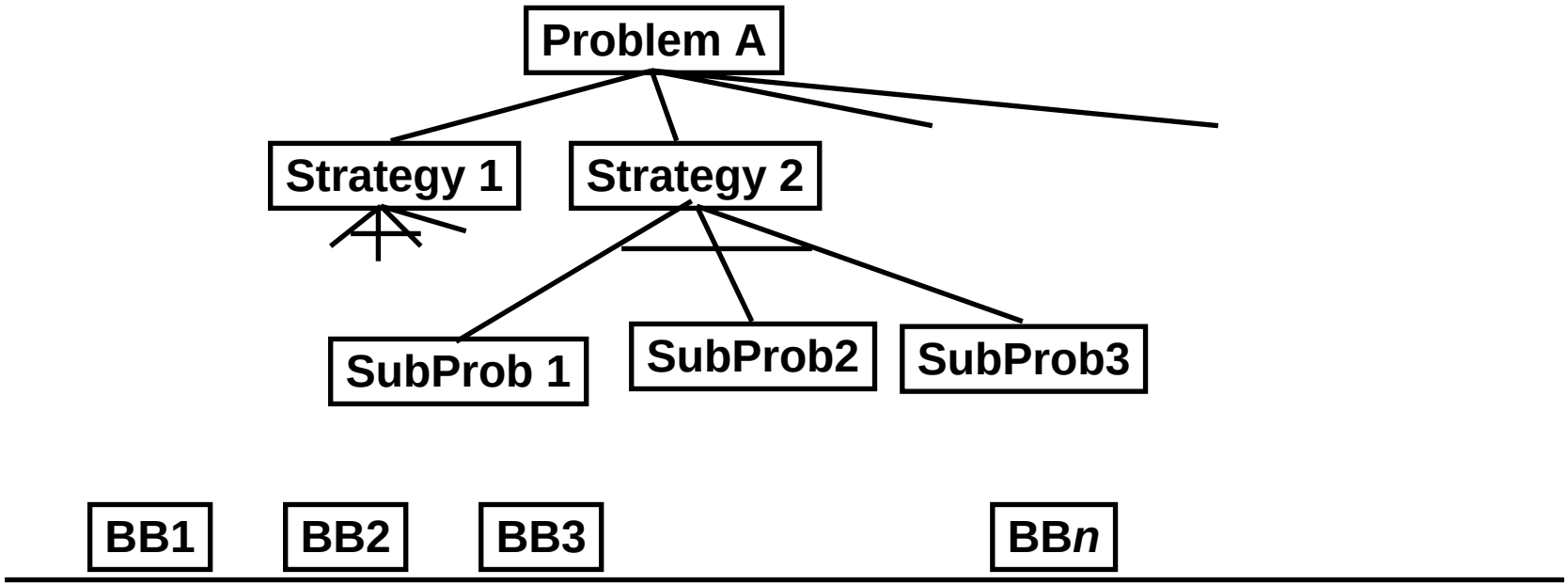
-- bottom-up *composition* of primitive building blocks into more complex assemblies

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

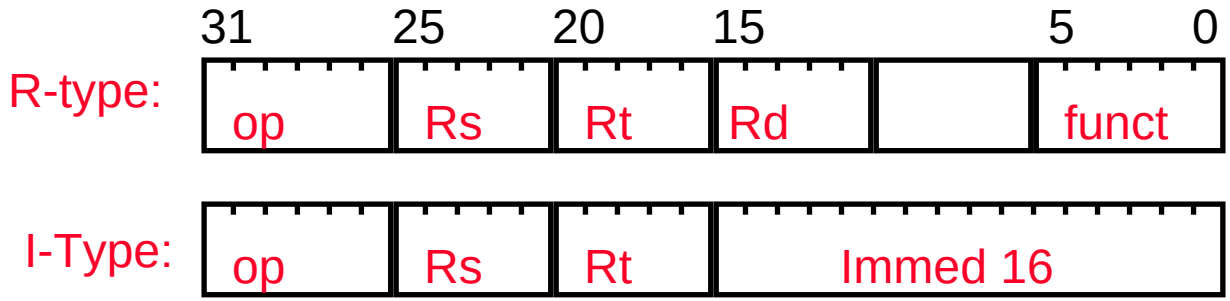
Problem: Design a “fast” ALU for the MIPS ISA

- Requirements?
- Must support the 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 arith generate overflow, no carry

Design Trick: divide & conquer

- 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

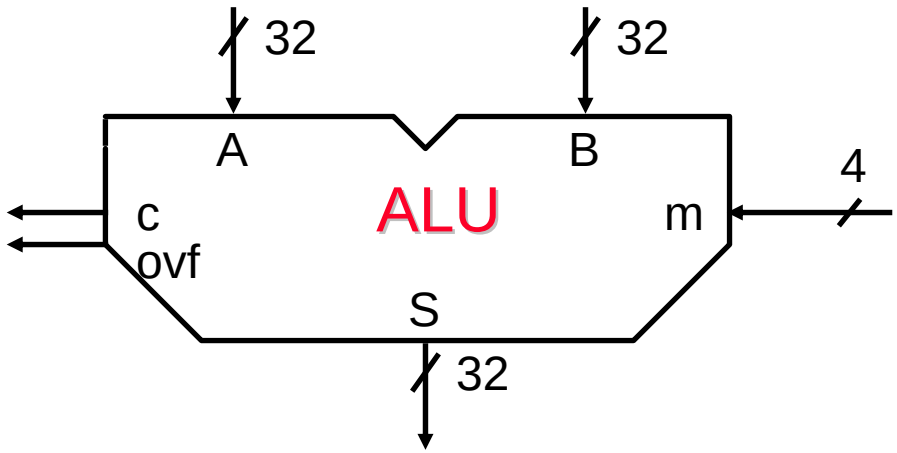
(1) Functional Specification

inputs: 2 x 32-bit operands A, B, 4-bit mode (sort of control)

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

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

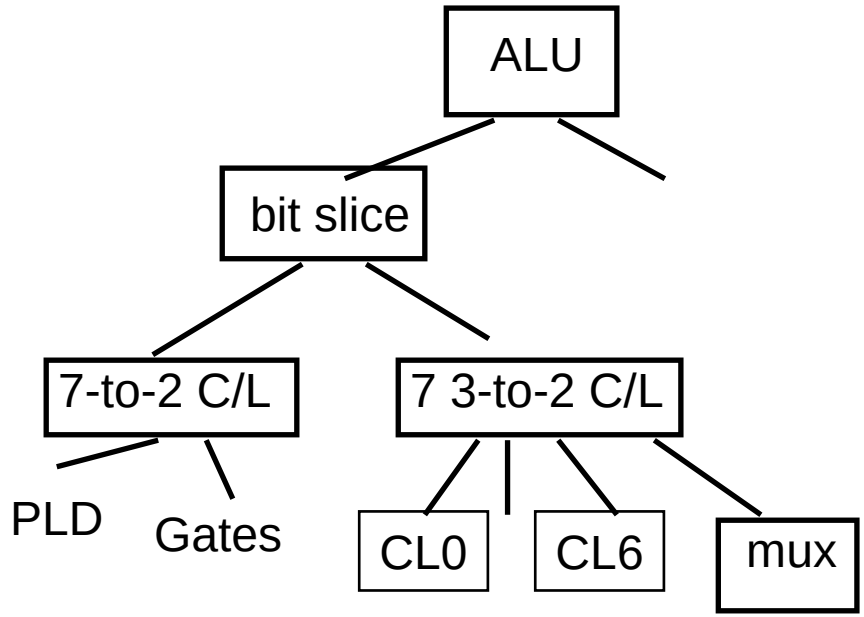
(2) Block Diagram (CAD-TOOL symbol, VHDL entity)



Behavioral Representation: VHDL

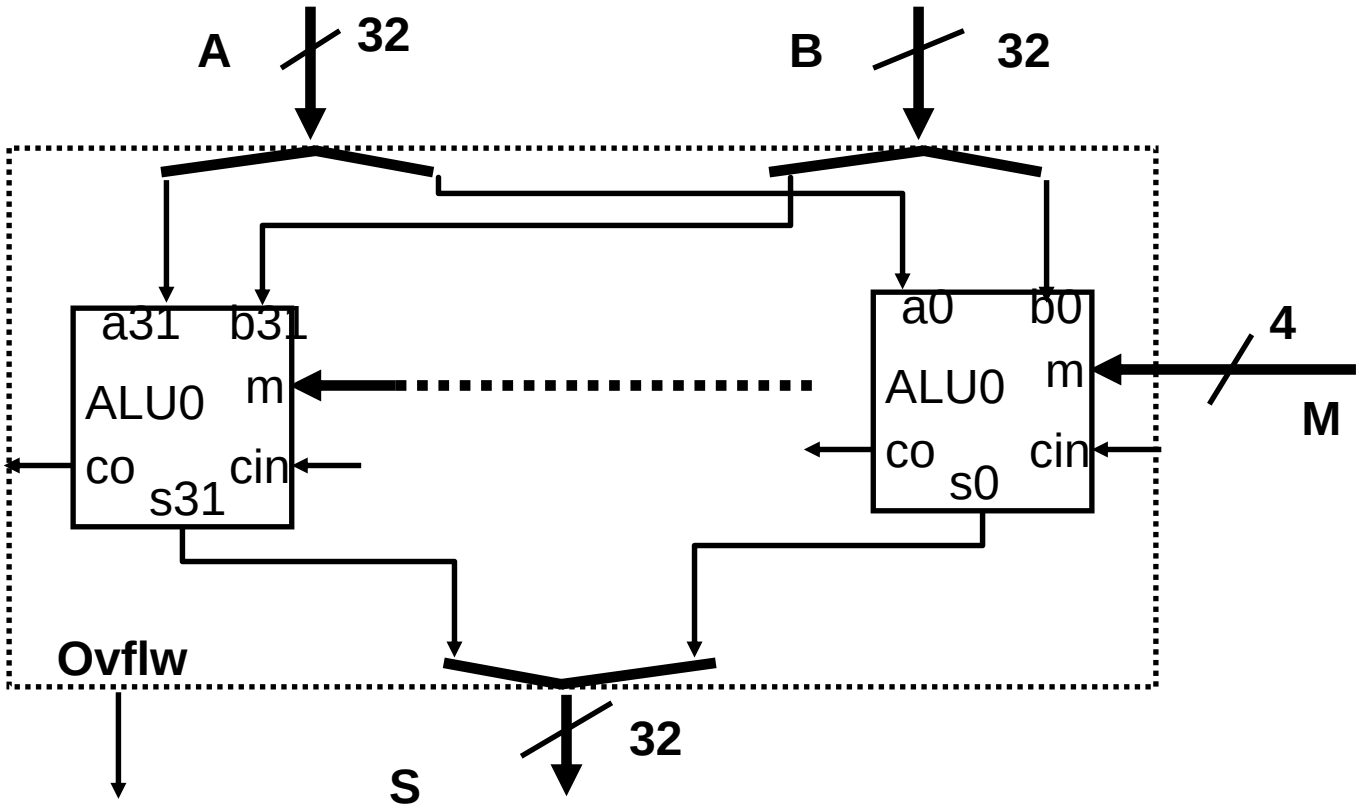
```
Entity ALU is
    generic (c_delay: integer := 20 ns;
             S_delay: integer := 20 ns);

    port ( signal A, B:  in  vlbit_vector (0 to 31);
           signal    m:  in  vlbit_vector (0 to 3);
           signal    S: out  vlbit_vector (0 to 31);
           signal    c: out  vlbit;
           signal  ovf: out  vlbit)
end ALU;


...

      S <= A + B;
```

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



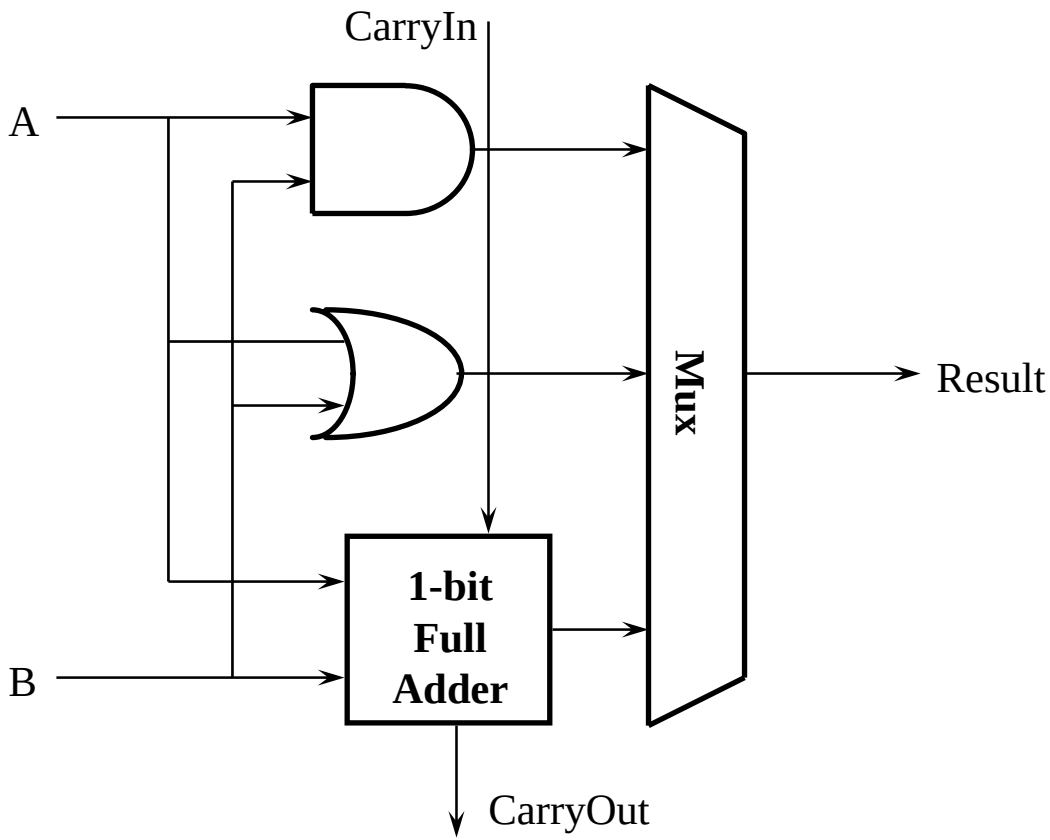
7-to-2 Combinational Logic

° **start turning the crank . . .**

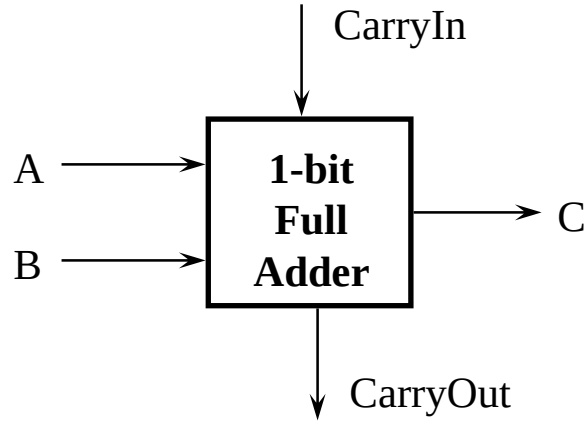
	Function	Inputs							Outputs		K-Map
		M0	M1	M2	M3	A	B	Cin	S	Cout	
0	add	0	0	0	0	0	0	0	0	0	
127											

A One Bit ALU

- **This 1-bit ALU will perform AND, OR, and ADD**



A One-bit Full Adder



- This is also called a (3, 2) adder
- Half Adder: No CarryIn nor CarryOut
- Truth Table:

Inputs			Outputs		
A	B	CarryIn	CarryOut	Sum	Comments
0	0	0	0	0	0 + 0 + 0 = 00
0	0	1	0	1	0 + 0 + 1 = 01
0	1	0	0	1	0 + 1 + 0 = 01
0	1	1	1	0	0 + 1 + 1 = 10
1	0	0	0	1	1 + 0 + 0 = 01
1	0	1	1	0	1 + 0 + 1 = 10
1	1	0	1	0	1 + 1 + 0 = 10
1	1	1	1	1	1 + 1 + 1 = 11

Logic Equation for CarryOut

Inputs			Outputs		
A	B	CarryIn	CarryOut	Sum	Comments
0	0	0	0	0	0 + 0 + 0 = 00
0	0	1	0	1	0 + 0 + 1 = 01
0	1	0	0	1	0 + 1 + 0 = 01
0	1	1	1	0	0 + 1 + 1 = 10
1	0	0	0	1	1 + 0 + 0 = 01
1	0	1	1	0	1 + 0 + 1 = 10
1	1	0	1	0	1 + 1 + 0 = 10
1	1	1	1	1	1 + 1 + 1 = 11

- **CarryOut = (!A & B & CarryIn) | (A & !B & CarryIn) | (A & B & !CarryIn) | (A & B & CarryIn)**
- **CarryOut = B & CarryIn | A & CarryIn | A & B**

Logic Equation for Sum

Inputs			Outputs		
A	B	CarryIn	CarryOut	Sum	Comments
0	0	0	0	0	0 + 0 + 0 = 00
0	0	1	0	1	0 + 0 + 1 = 01
0	1	0	0	1	0 + 1 + 0 = 01
0	1	1	1	0	0 + 1 + 1 = 10
1	0	0	0	1	1 + 0 + 0 = 01
1	0	1	1	0	1 + 0 + 1 = 10
1	1	0	1	0	1 + 1 + 0 = 10
1	1	1	1	1	1 + 1 + 1 = 11

° **Sum = (!A & !B & CarryIn) | (!A & B & !CarryIn) | (A & !B & !CarryIn) | (A & B & CarryIn)**

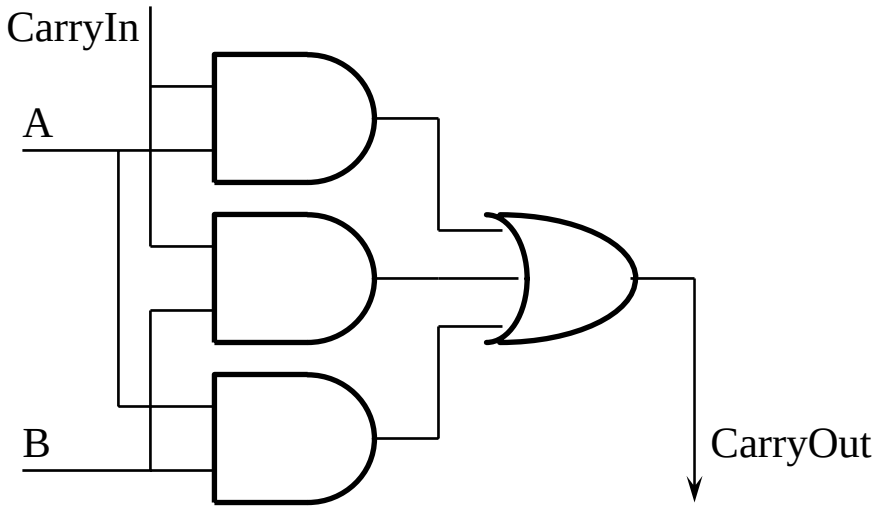
Logic Equation for Sum (continue)

- **Sum = (!A & !B & CarryIn) | (!A & B & !CarryIn) | (A & !B & !CarryIn) | (A & B & CarryIn)**
- **Sum = A XOR B XOR CarryIn**
- **Truth Table for XOR:**

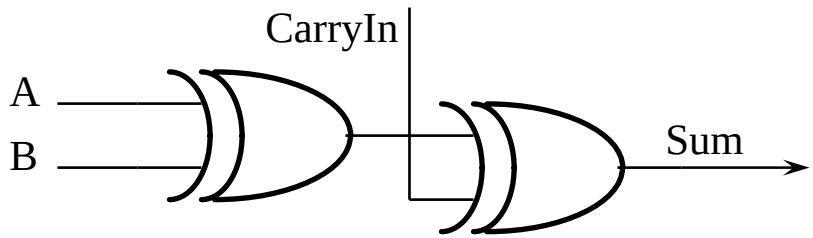
X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

Logic Diagrams for CarryOut and Sum

- **CarryOut = B & CarryIn | A & CarryIn | A & B**

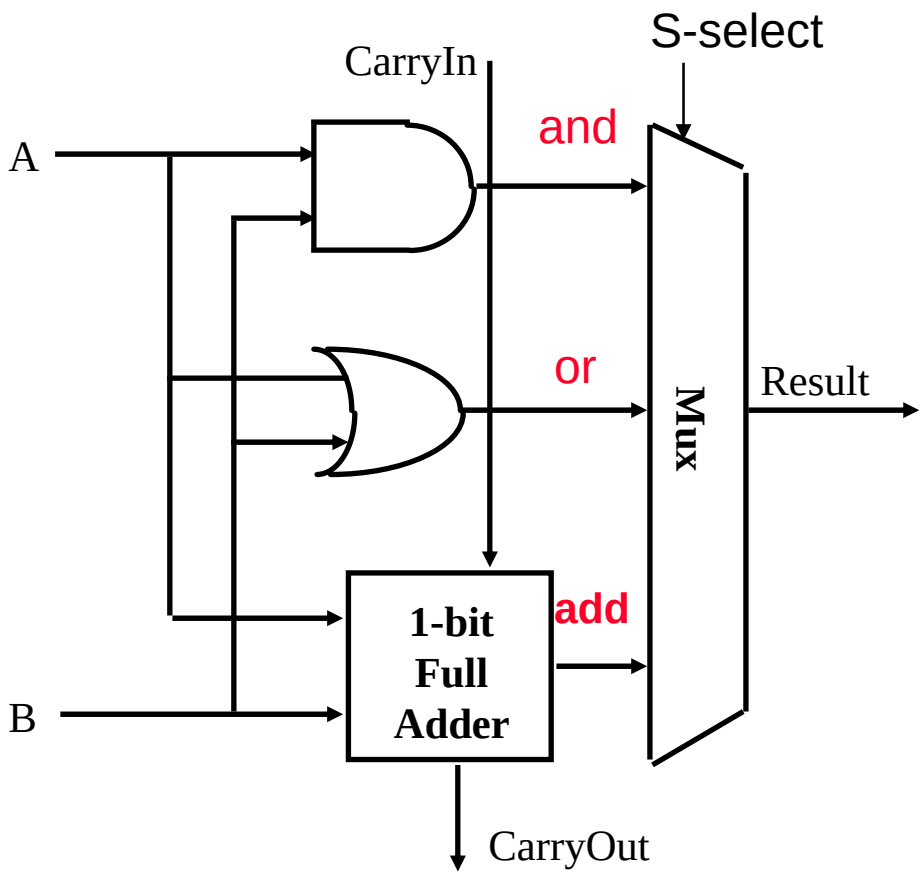


- **Sum = A XOR B XOR CarryIn**



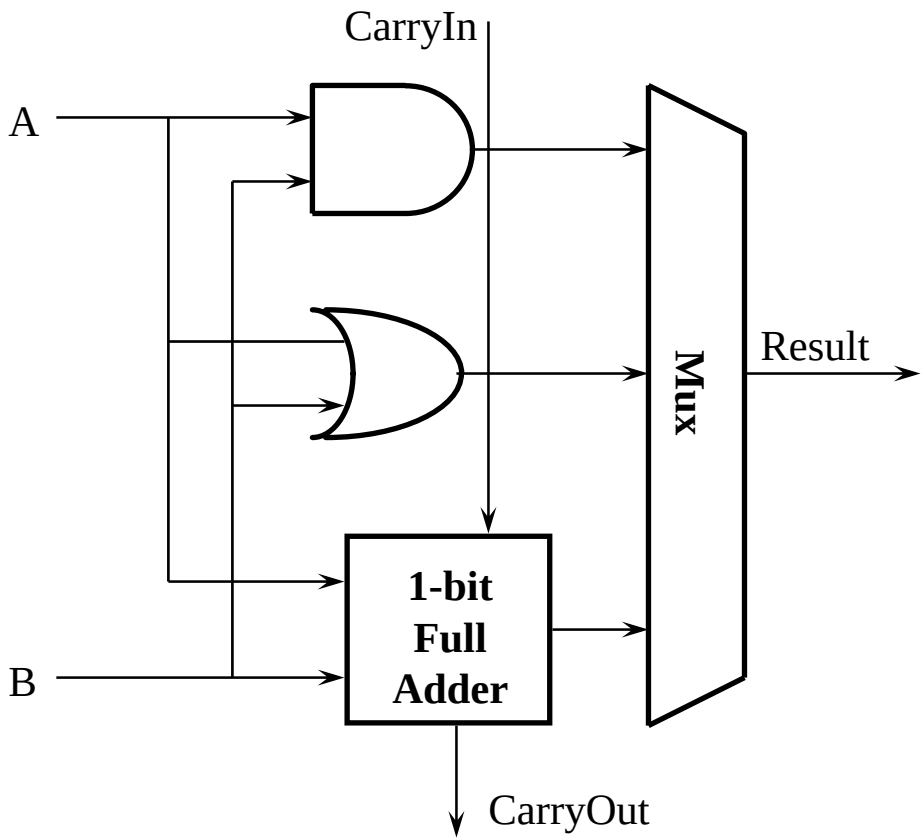
Seven plus a MUX ?

- **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**

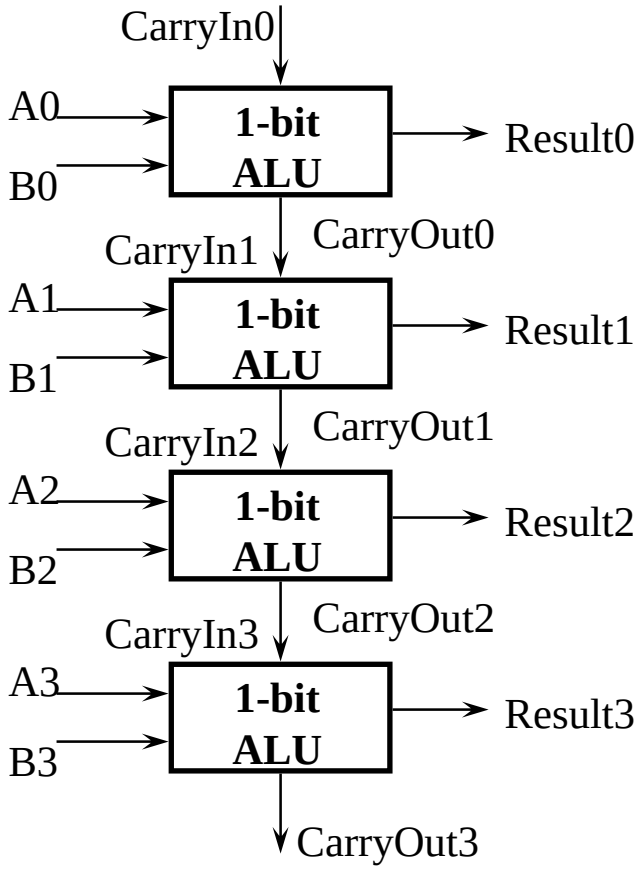


A 4-bit ALU

- 1-bit ALU

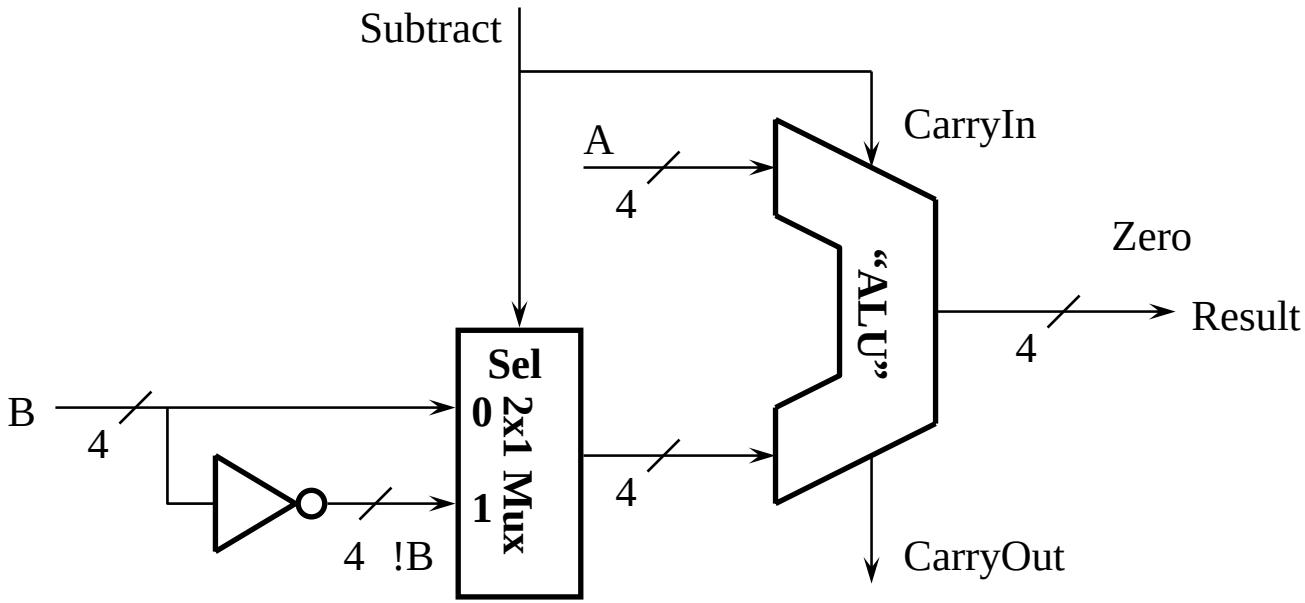


4-bit ALU



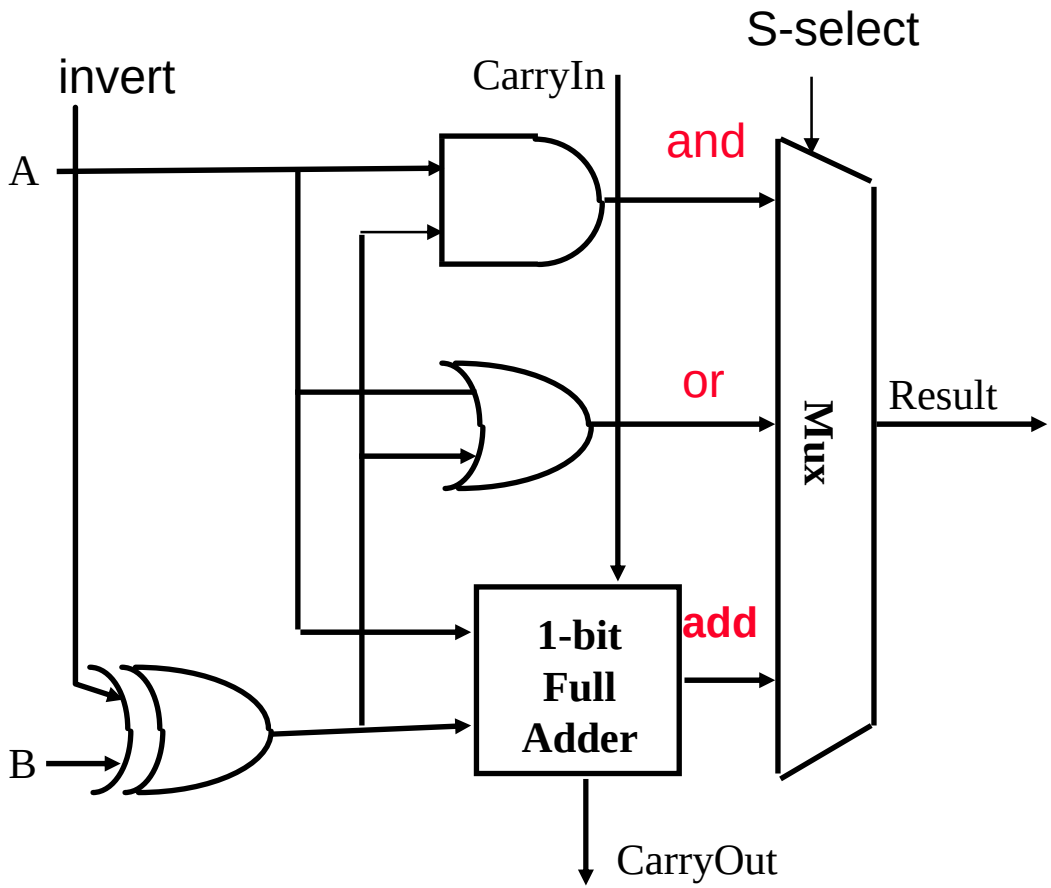
How About Subtraction?

- **Keep in mind the followings:**
  - **(A - B) is the that as: A + (-B)**
  - **2's Complement: Take the inverse of every bit and add 1**
- **Bit-wise inverse of B is !B:**
  - **A + !B + 1 = A + (!B + 1) = A + (-B) = A - B**



Additional operations

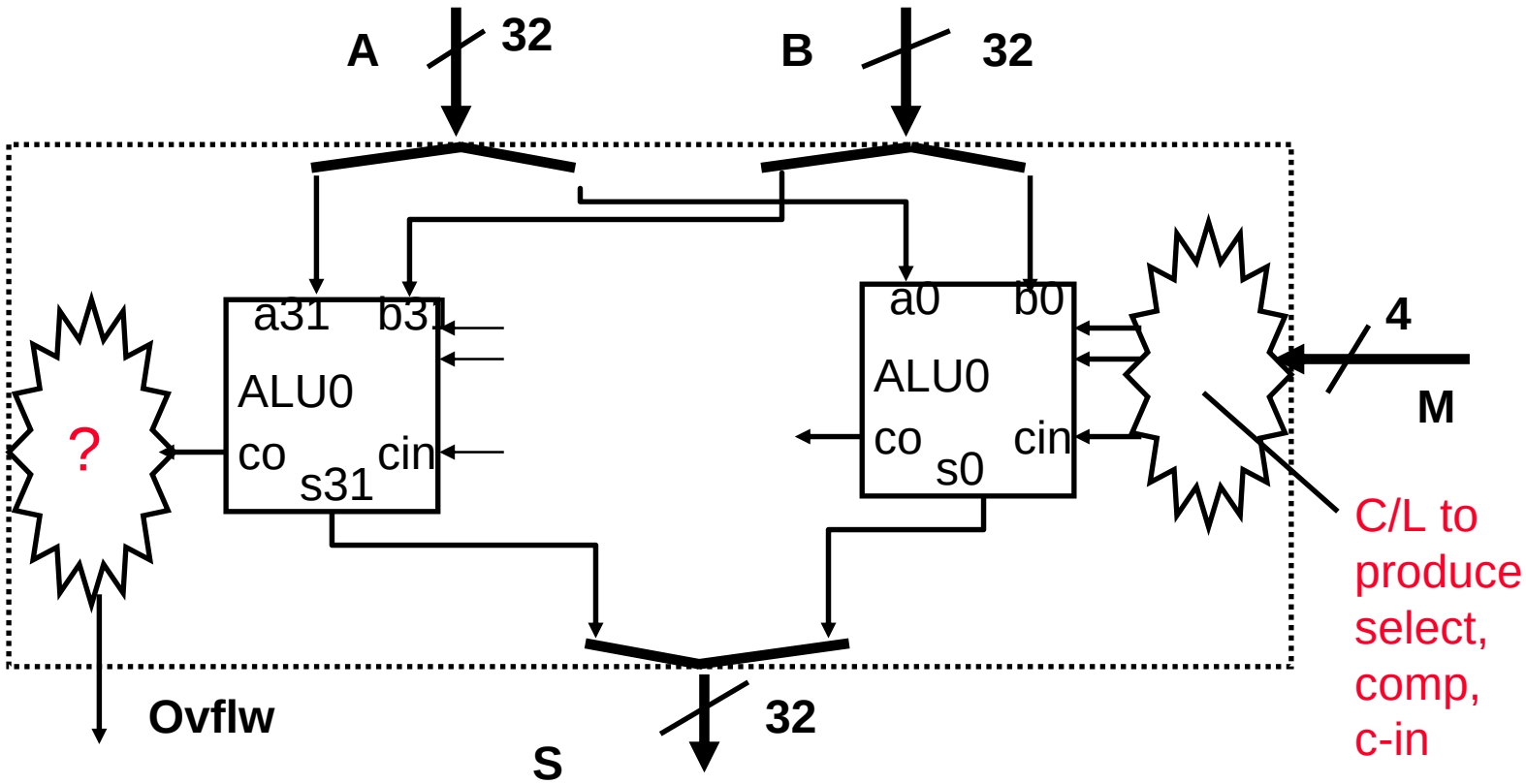
- **A - B = A + (– B)**
  - **form two complement by invert and add one**



Set-less-than? – left as an exercise

Revised Diagram

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



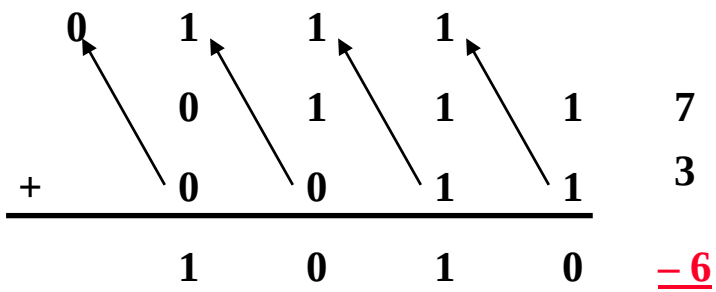
Overflow

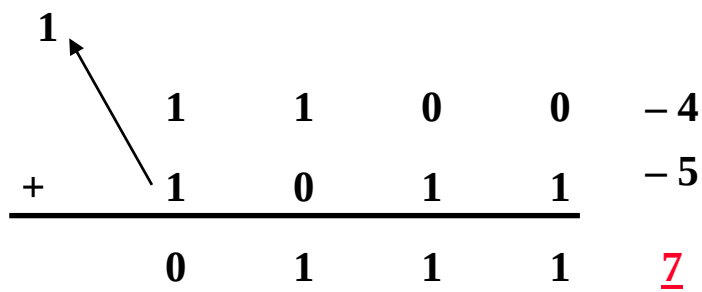
Decimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Decimal	2's Complement
0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

° Examples: 7 + 3 = 10 but ...

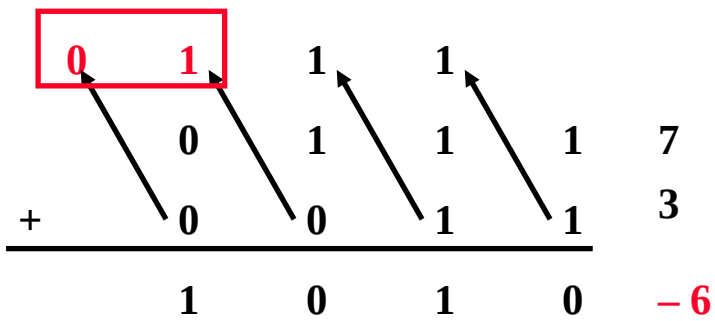
° - 4 - 5 = - 9 but ...

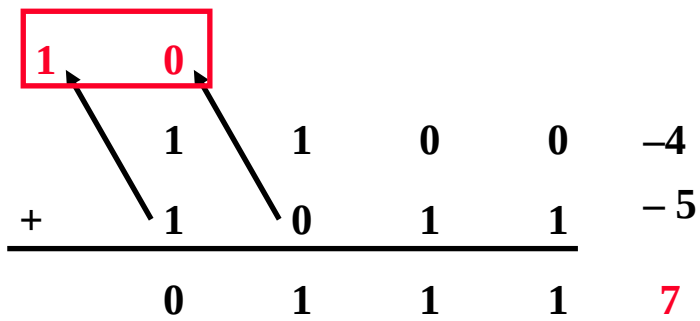




Overflow Detection

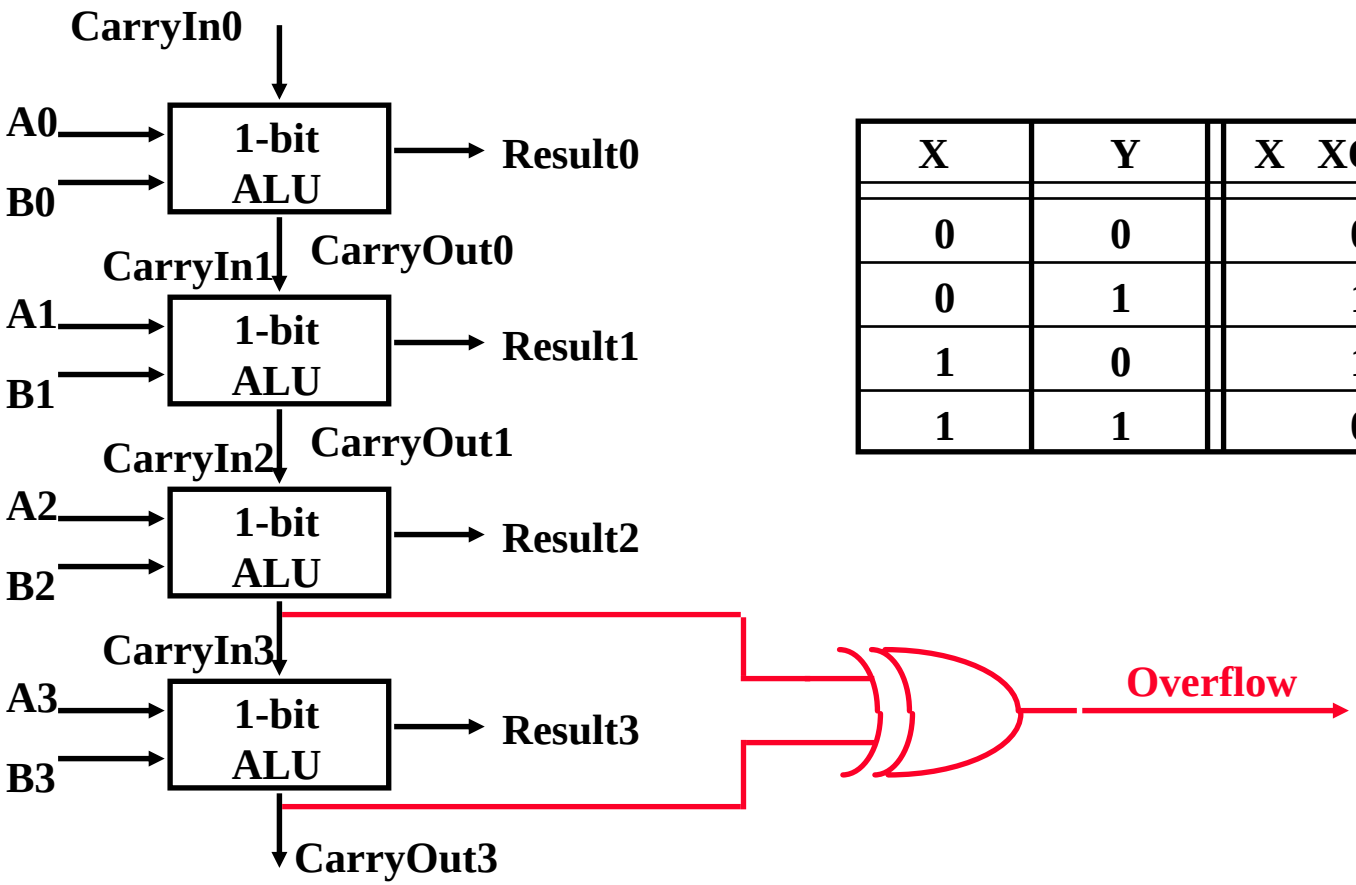
- Overflow: the result is too large (or too small) to represent properly
  - Example: - 8 < = 4-bit binary number <= 7
- When adding operands with different signs, overflow cannot occur!
- Overflow occurs when adding:
  - 2 positive numbers and the sum is negative
  - 2 negative numbers and the sum is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





Overflow Detection Logic

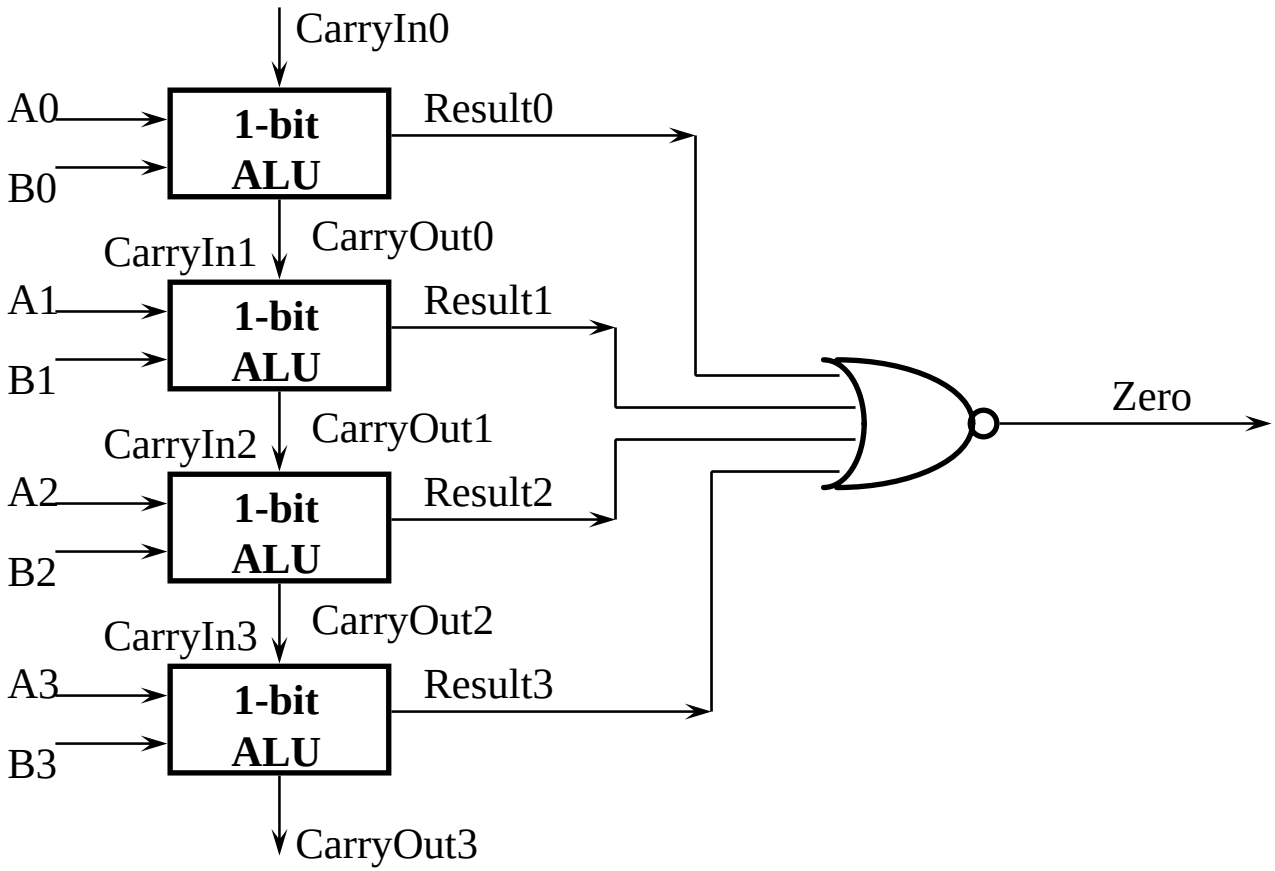
- Carry into MSB ° Carry out of MSB
  - For a N-bit ALU: Overflow = CarryIn[N - 1] XOR CarryOut[N - 1]



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

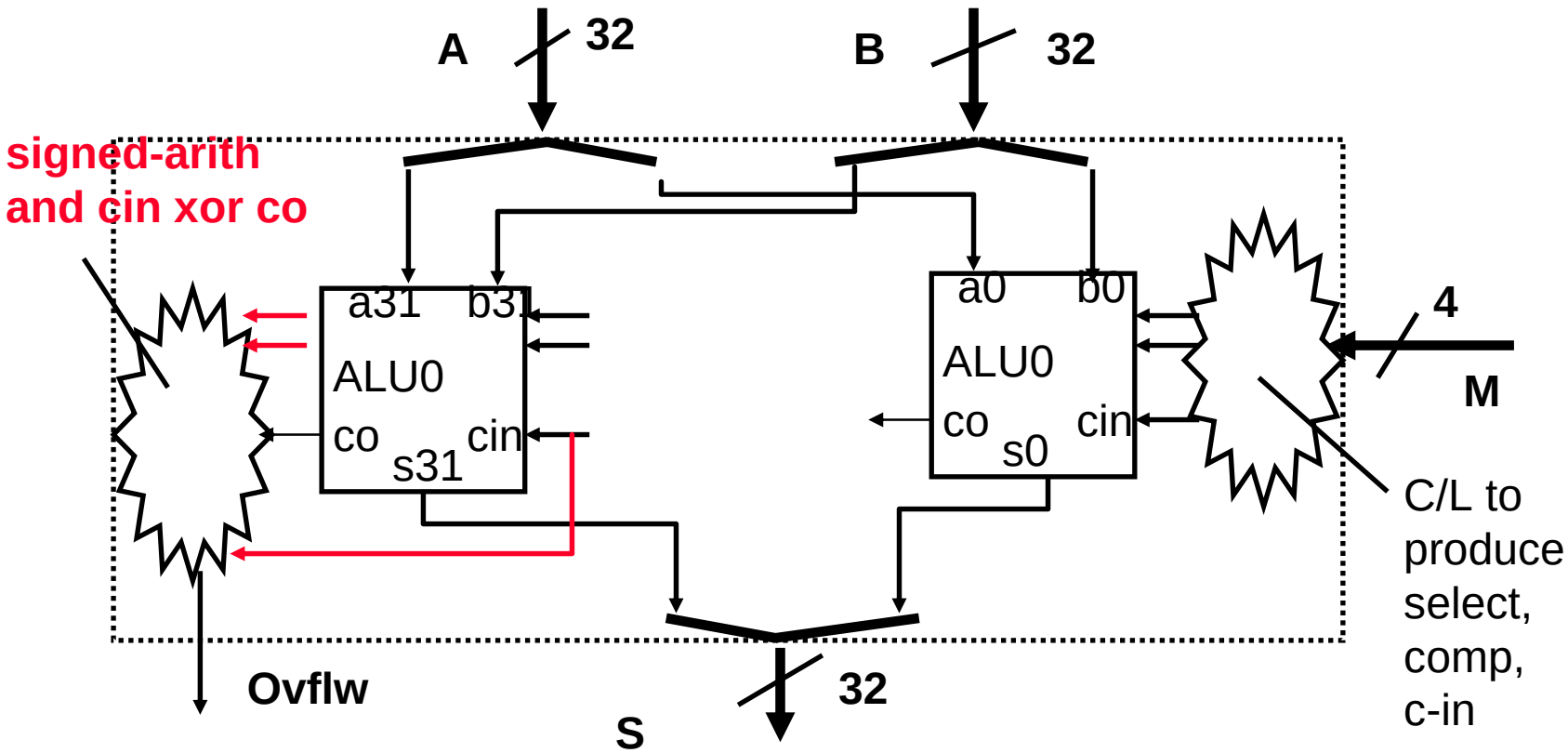
Zero Detection Logic

- **Zero Detection Logic is just a one BIG NOR gate**
  - **Any non-zero input to the NOR gate will cause its output to be zero**



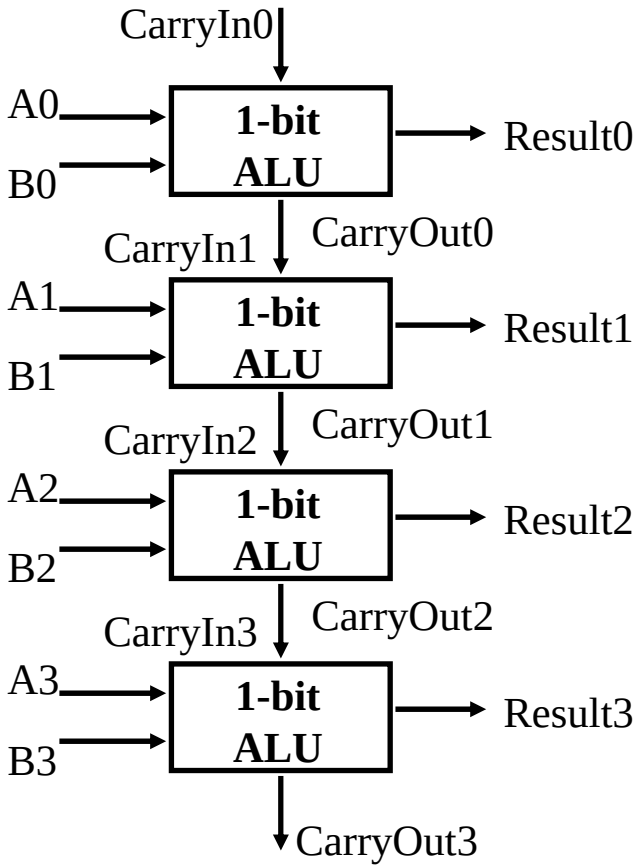
More Revised Diagram

- LSB and MSB need to do a little extra



But What about Performance?

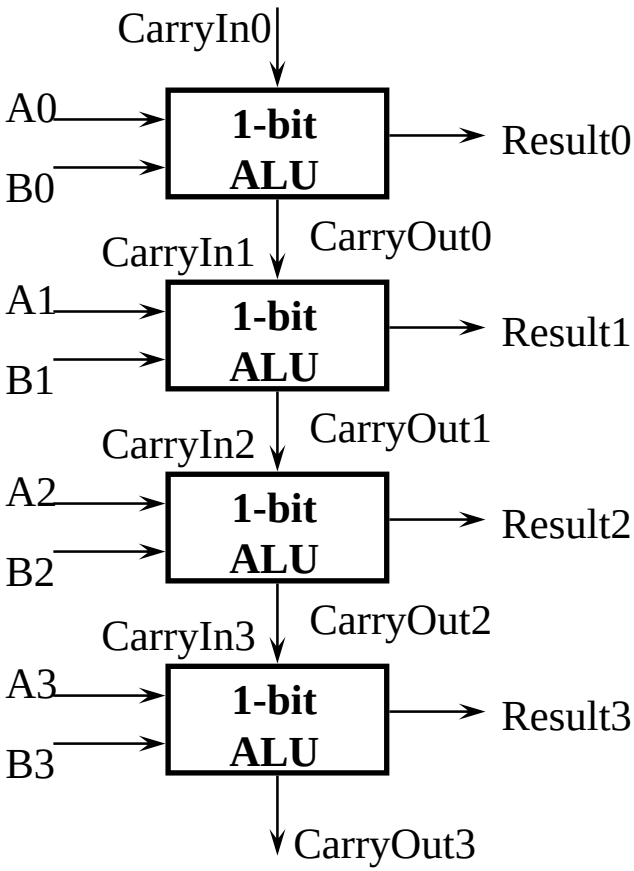
° **Critical Path of n-bit Rippled-carry adder is n*CP**

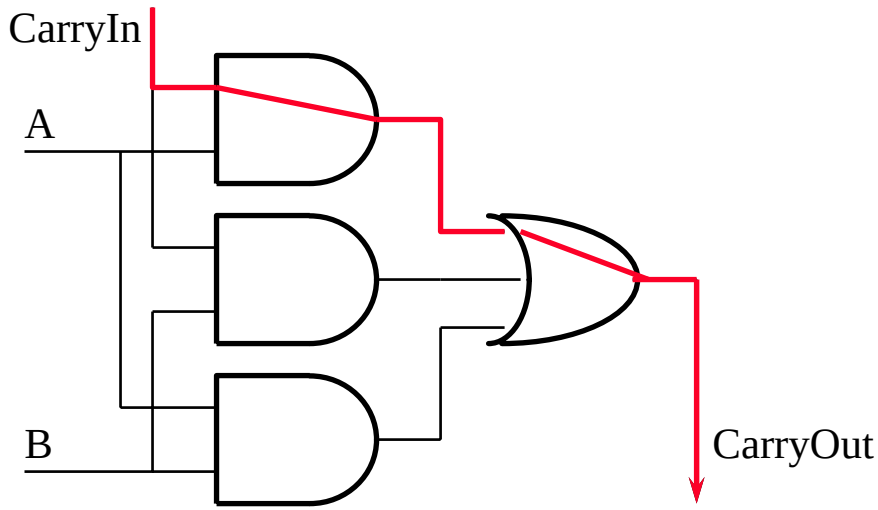


Design Trick: throw hardware at it

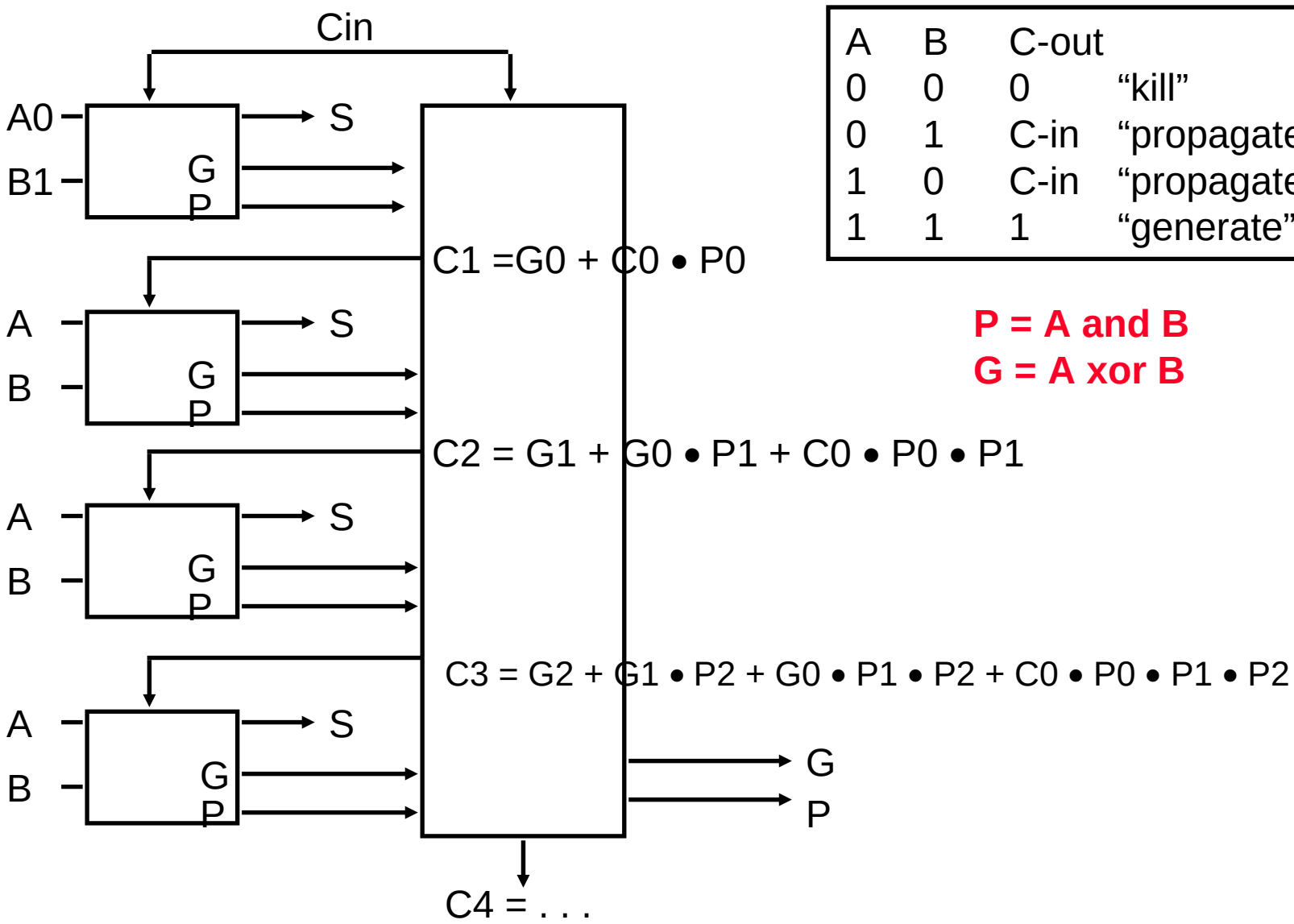
The Disadvantage of Ripple Carry

- The adder we just built is called a “Ripple Carry Adder”
  - The carry bit may have to propagate from LSB to MSB
  - Worst case delay for a N-bit adder: 2N-gate delay





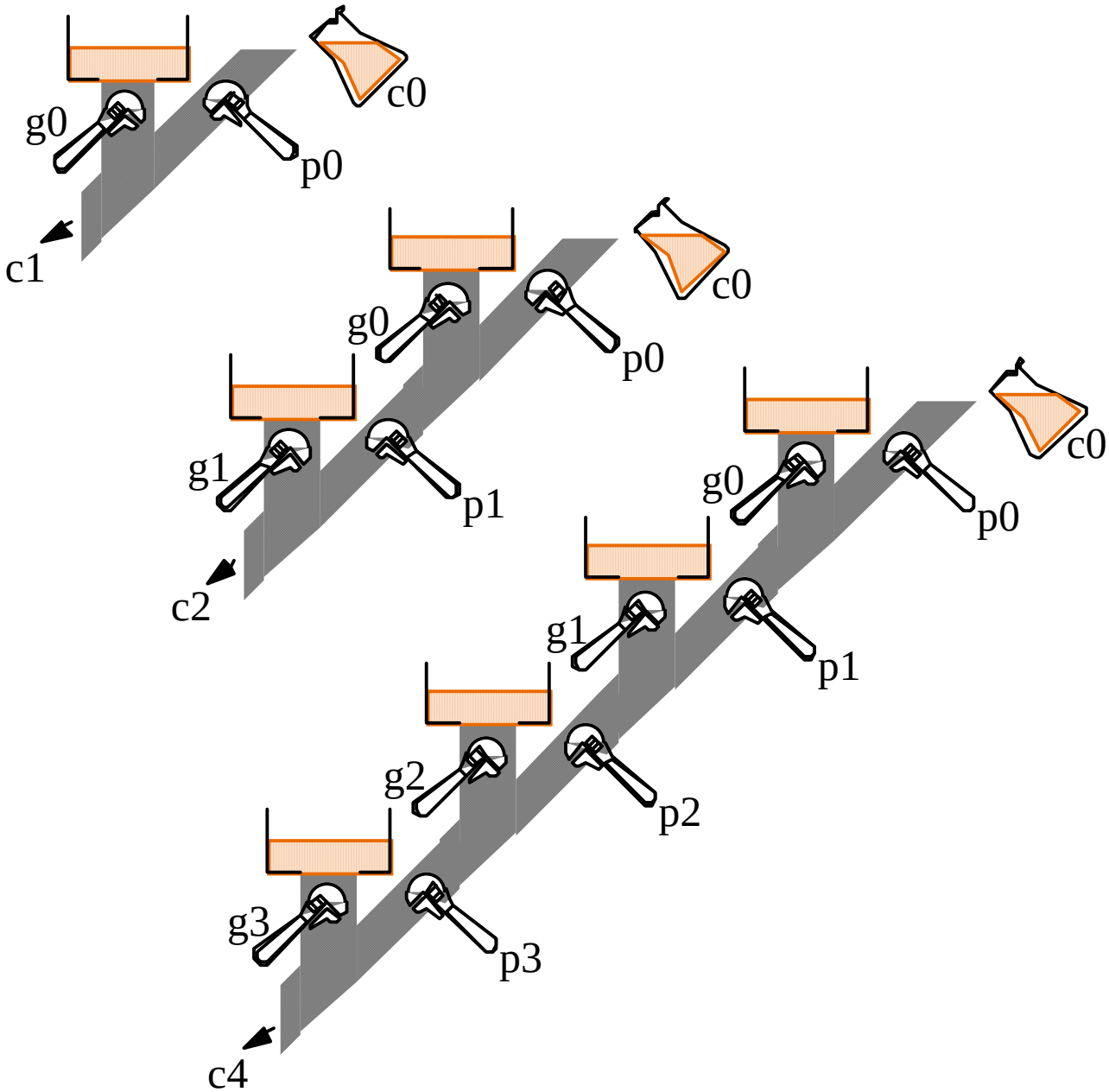
Carry Look Ahead (Design trick: peek)



A	B	C-out	
0	0	0	“kill”
0	1	C-in	“propagate”
1	0	C-in	“propagate”
1	1	1	“generate”

P = A and B
G = A xor B

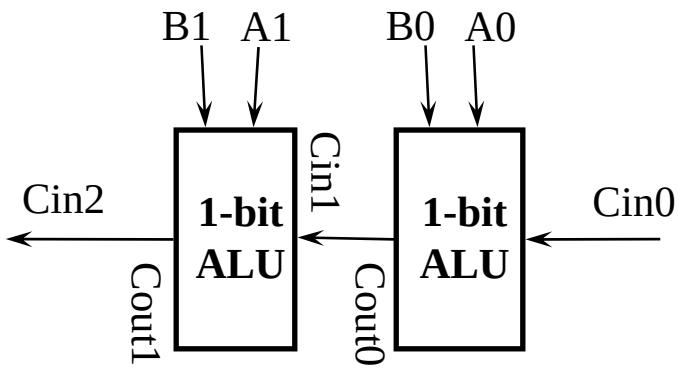
Plumbing as Carry Lookahead Analogy



The Idea Behind Carry Lookahead (Continue)

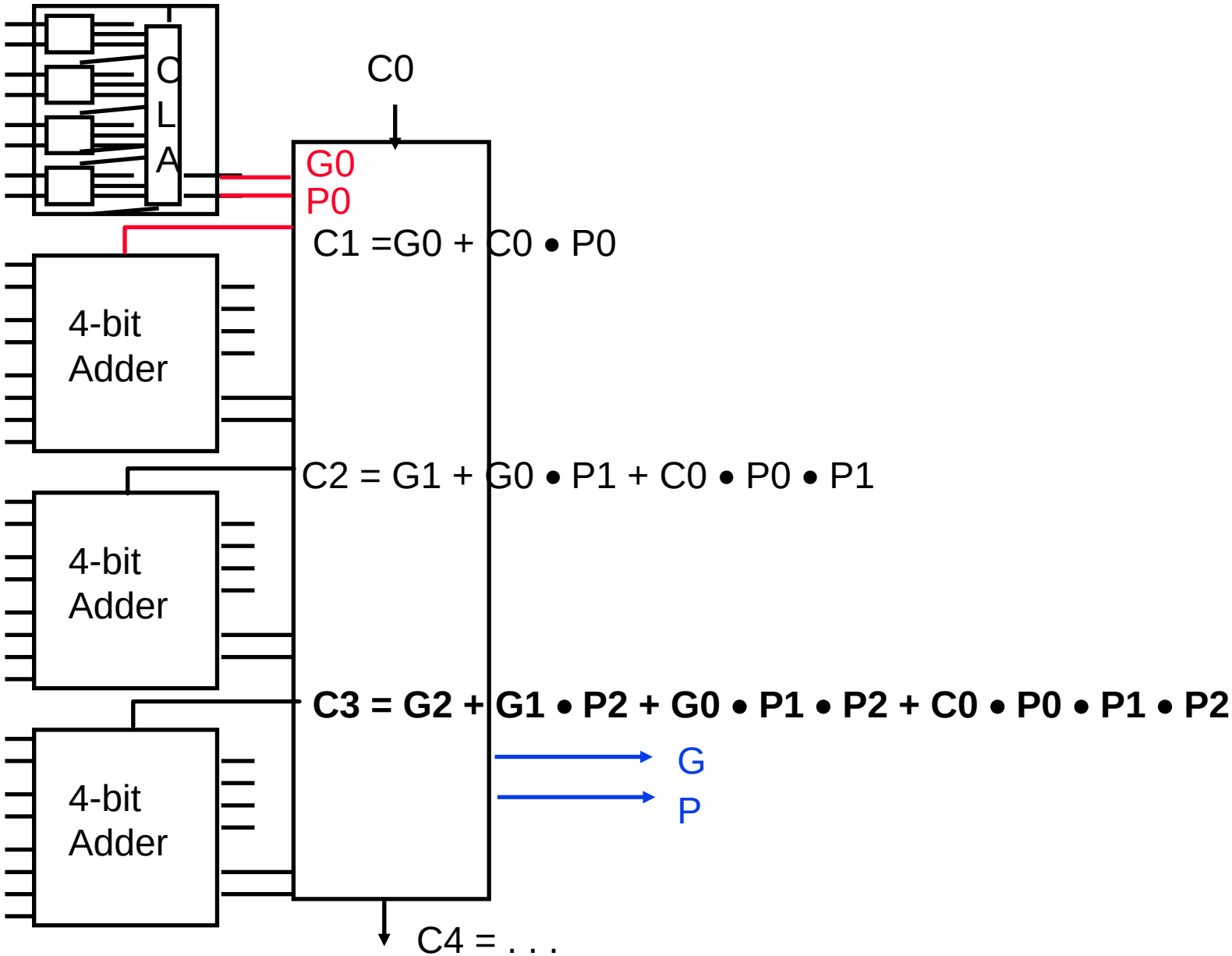
- Using the two new terms we just defined:
  - Generate Carry at Bit i gi = Ai & Bi
  - Propagate Carry via Bit i pi = Ai or Bi
- We can rewrite:
  - Cin1 = g0 | (p0 & Cin0)
  - Cin2 = g1 | (p1 & g0) | (p1 & p0 & Cin0)
  - Cin3 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & Cin0)
- Carry going into bit 3 is 1 if
  - We generate a carry at bit 2 (g2)
  - Or we generate a carry at bit 1 (g1) and bit 2 allows it to propagate (p2 & g1)
  - Or we generate a carry at bit 0 (g0) and bit 1 as well as bit 2 allows it to propagate (p2 & p1 & g0)
  - Or we have a carry input at bit 0 (Cin0) and bit 0, 1, and 2 all allow it to propagate (p2 & p1 & p0 & Cin0)

The Idea Behind Carry Lookahead

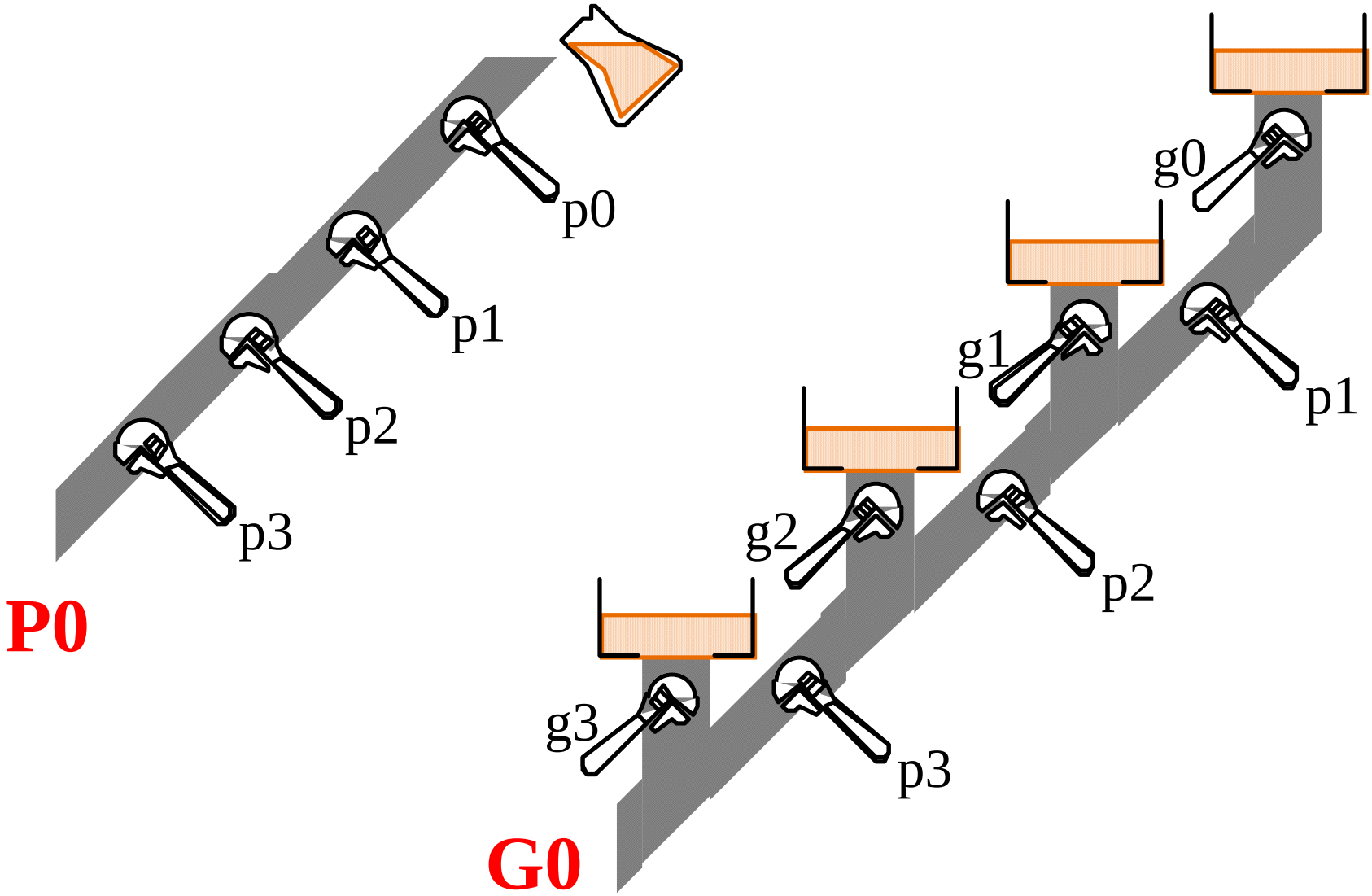


- Recall: CarryOut = (B & CarryIn) | (A & CarryIn) | (A & B)
  - Cin2 = Cout1 = (B1 & Cin1) | (A1 & Cin1) | (A1 & B1)
  - Cin1 = Cout0 = (B0 & Cin0) | (A0 & Cin0) | (A0 & B0)
- Substituting Cin1 into Cin2:
  - Cin2 = (A1 & A0 & B0) | (A1 & A0 & Cin0) | (A1 & B0 & Cin0) | (B1 & A0 & B0) | (B1 & A0 & Cin0) | (B1 & A0 & Cin0) | (A1 & B1)
- Now define two new terms:
  - Generate Carry at Bit i gi = Ai & Bi
  - Propagate Carry via Bit i pi = Ai or Bi
  - READ and LEARN Details

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

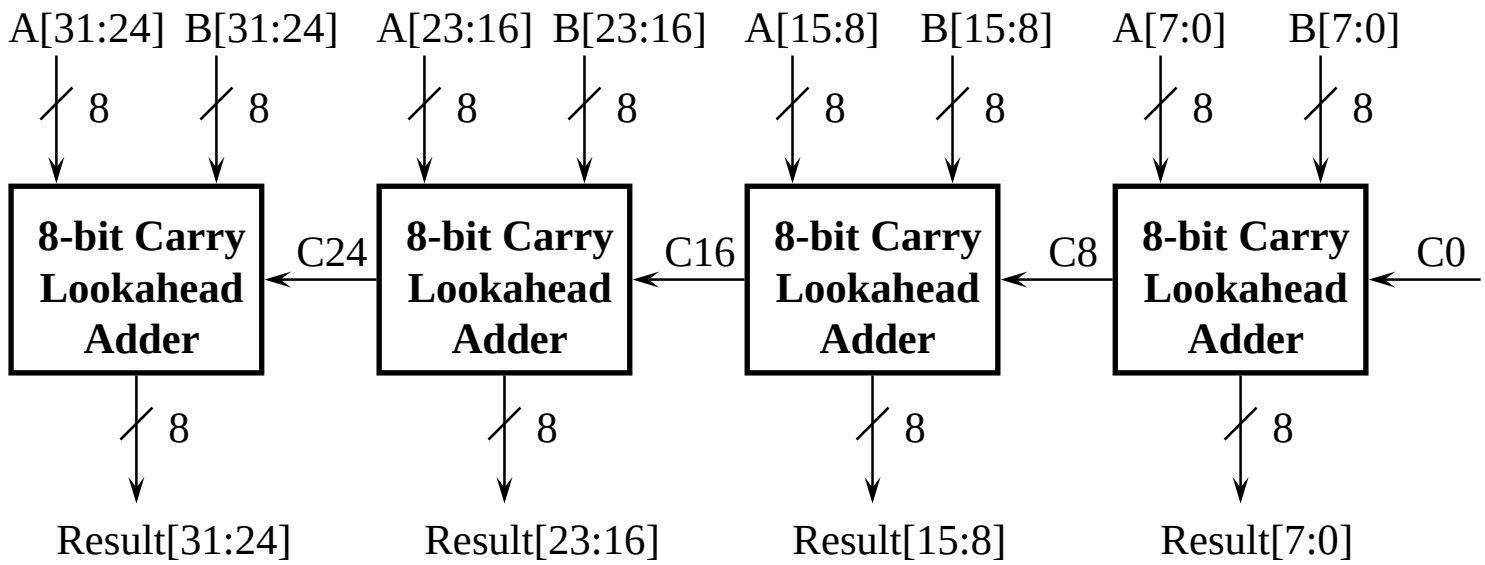


2nd level Carry, Propagate as Plumbing



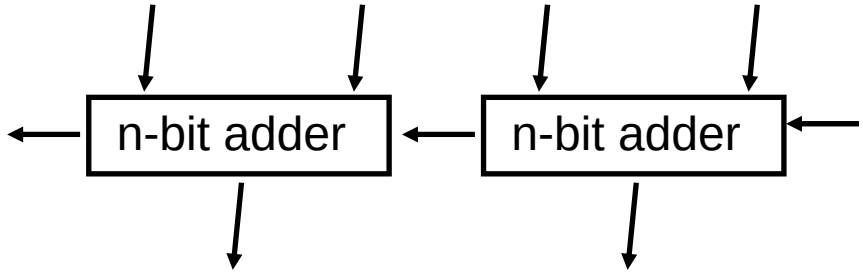
A Partial Carry Lookahead Adder

- It is very expensive to build a “full” carry lookahead adder
  - Just imagine the length of the equation for Cin31
- Common practices:
  - Connects several N-bit Lookahead Adders to form a big adder
  - Example: connects four 8-bit carry lookahead adders to form a 32-bit partial carry lookahead adder

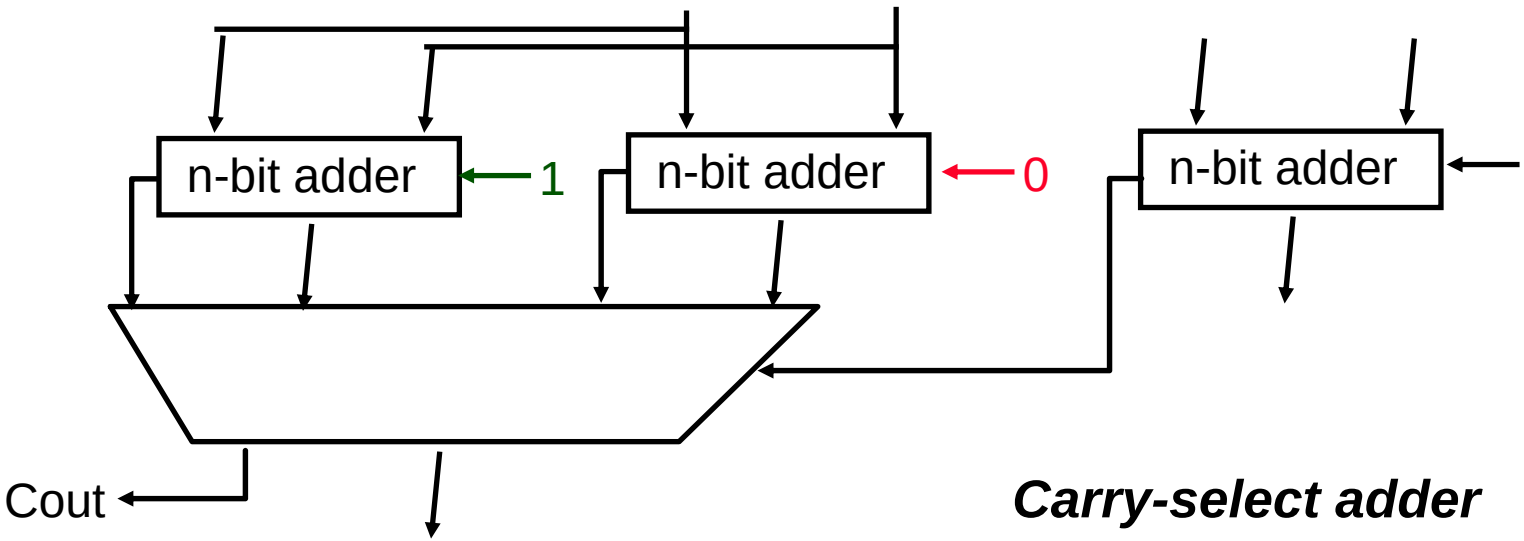


Design Trick: Guess

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



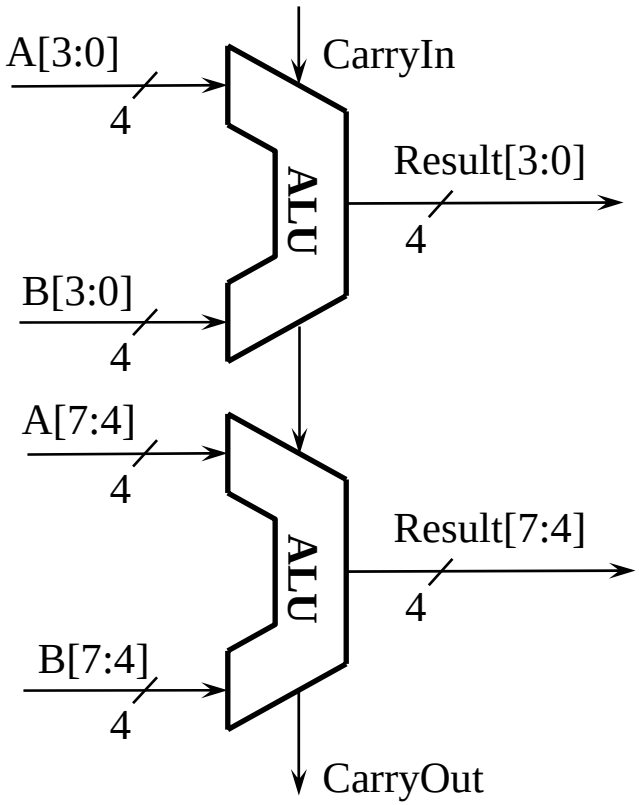
CP(2n) = CP(n) + CP(mux)



Carry-select adder

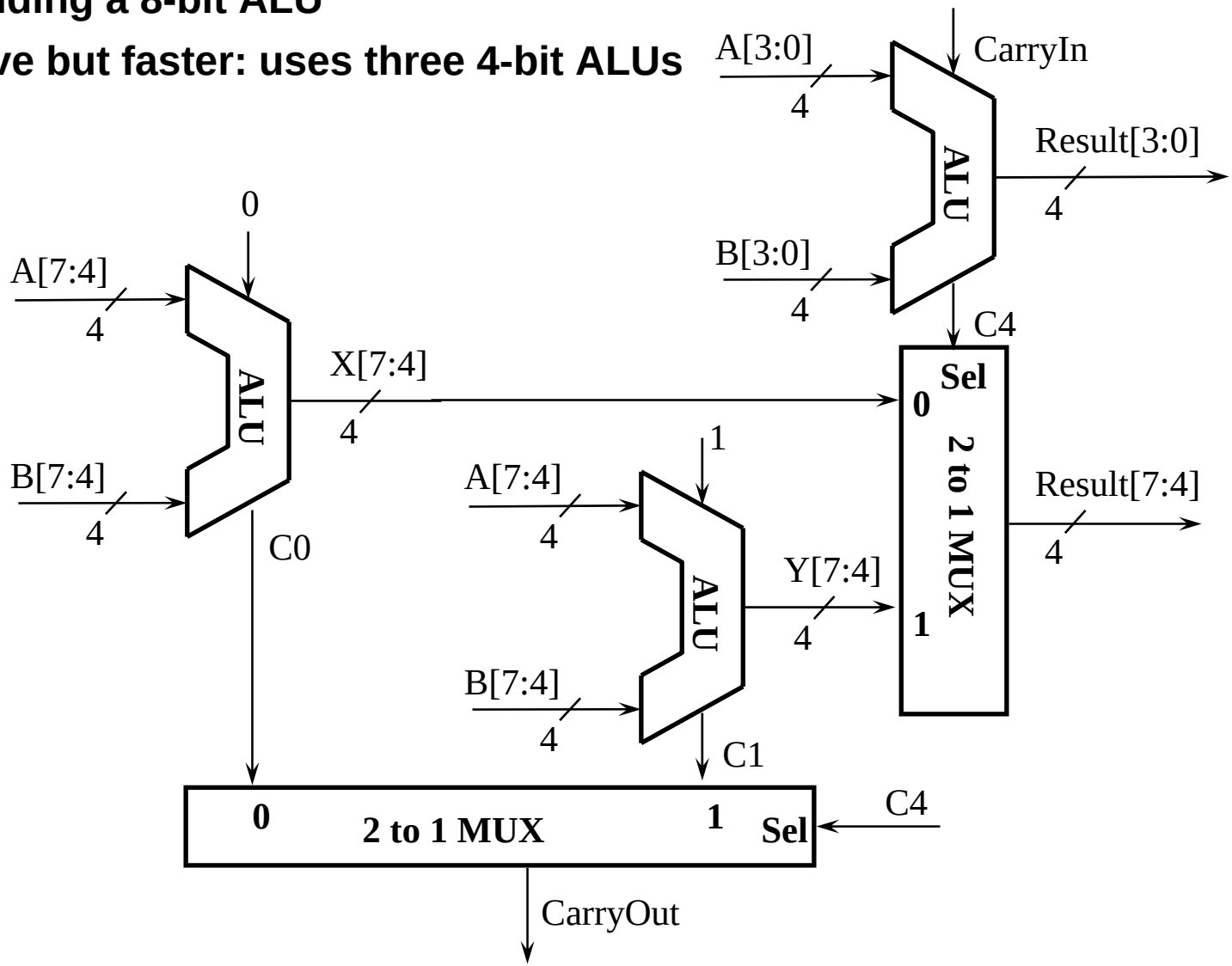
Carry Select

- **Consider building a 8-bit ALU**
  - **Simple: connects two 4-bit ALUs in series**

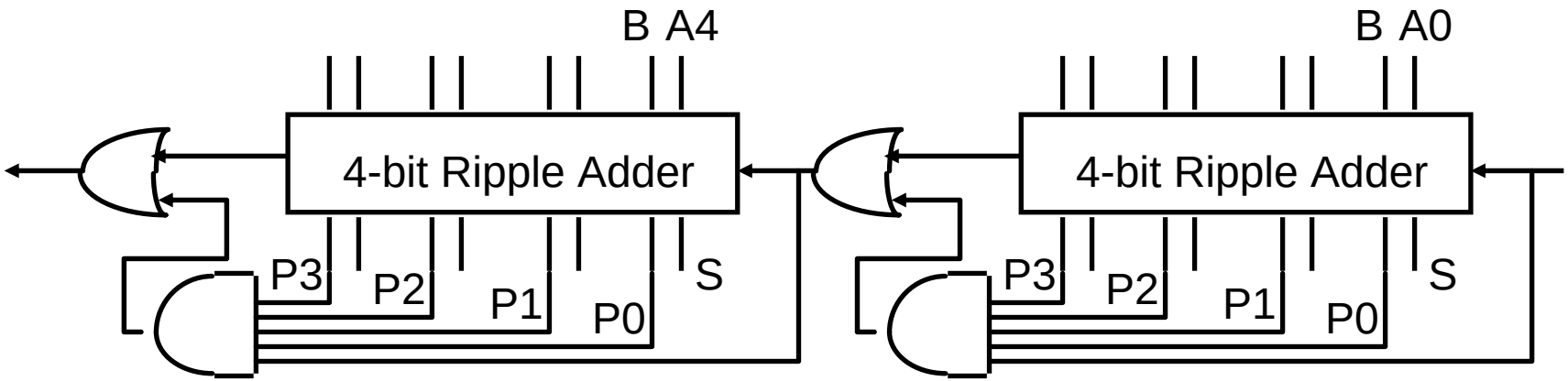


Carry Select (Continue)

- Consider building a 8-bit ALU
  - Expensive but faster: uses three 4-bit ALUs

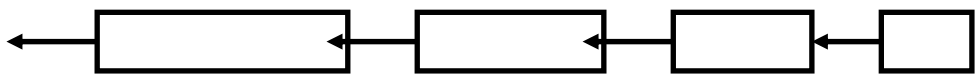


Carry Skip Adder: reduce worst case delay



Just speed up the slowest case for each block

Exercise: optimal design uses variable block sizes



Additional MIPS ALU requirements

- **Mult, MultU, Div, DivU (next lecture)**
  => Need 32-bit multiply and divide, signed and unsigned

- **Sll, Srl, Sra (next lecture)**
  => Need left shift, right shift, right shift arithmetic by 0 to 31 bits

- **Nor (leave as exercise to reader)**
  => logical NOR or use 2 steps: (A OR B) XOR 1111....1111

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.

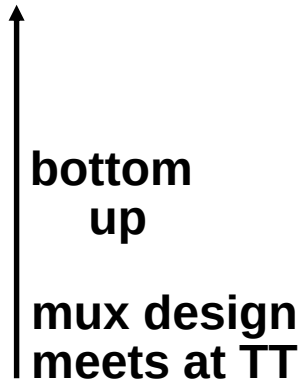
Summary of the Design Process

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:

- **Gate Count** —— ***Area***
- **[Package Count]** —— ***Pin Out***
- **Logic Levels** —— ***Delay*** ***Power***
- **Fan-in/Fan-out** —— ***Delay*** ***Power***

Cost ***Design time***