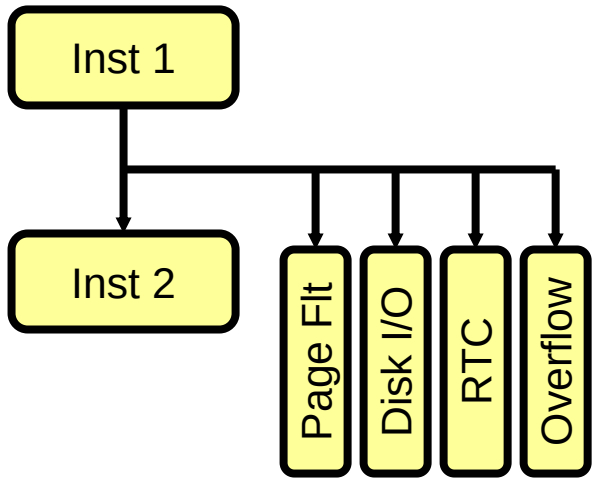
Lecture 7: Instruction Set Architectures - IV

- Last Time
  - Register organization
  - Memory issues (endian-ness, alignment, etc.)

- Today
  - Exceptions
  - General principles of ISA design
  - Role of compiler
  - Computer arithmetic

Control - Exceptions/Events

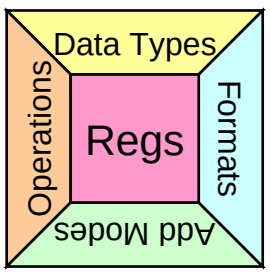
- Implied multi-way branch after every instruction
  - External events (interrupts)
    - completion of I/O operations
  - Internal events (faults or exceptions)
    - arithmetic overflow
    - page fault



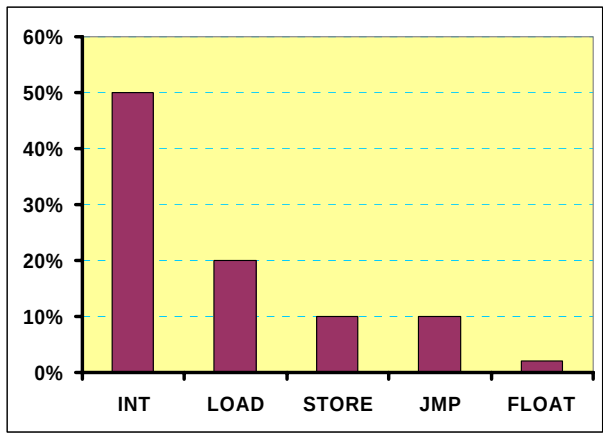
- What happens????
  - EPC ← PC of instruction that caused fault
  - PC ← f(Fault type)
    - new PC from HW table lookup
  - Return: PC ← FPC + 4

Principles of Instruction Set Design

- Keep it simple (KISS)
  - complexity
    - increases logic area
    - increases pipe stages
    - increases development time
  - evolution tends to make kludges
- Orthogonality (modularity)
  - simple rules, few exceptions
  - all ops on all registers

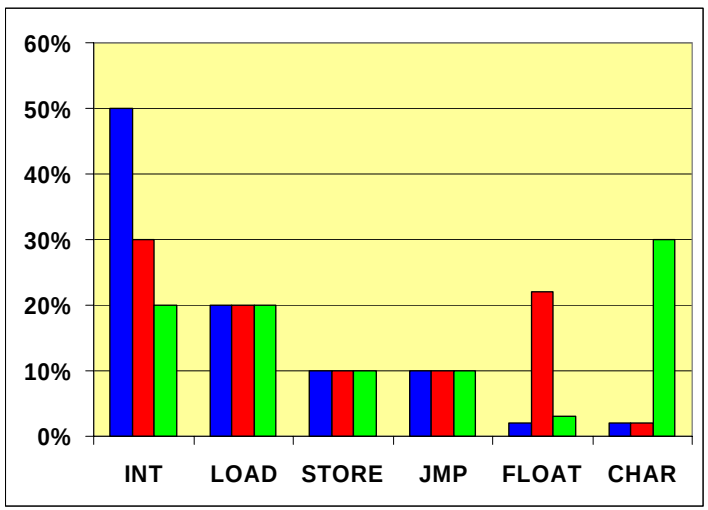


- Frequency
  - make the common case fast
    - some instructions (cases) are more important than others

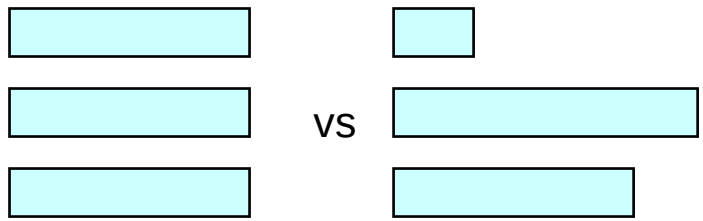


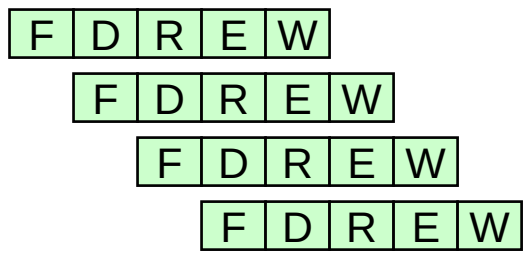
Principles of Instruction Set Design (part 2)

- Generality
  - not all problems need the same features/instructions
  - principle of *least surprise*
  - performance should be easy to predict



- Locality and concurrency
  - design ISA to permit efficient implementation
    - today
    - 10 years from now

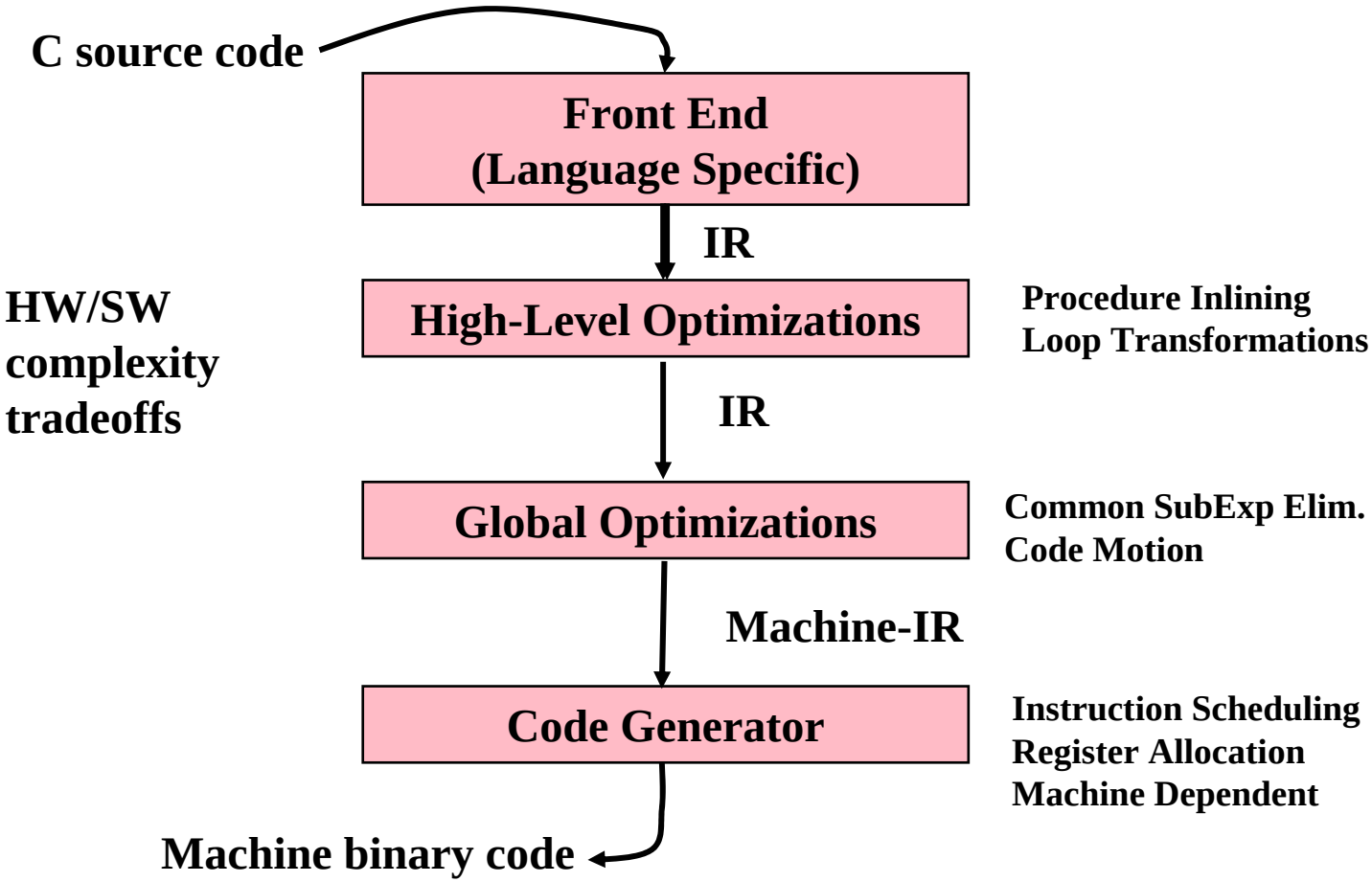




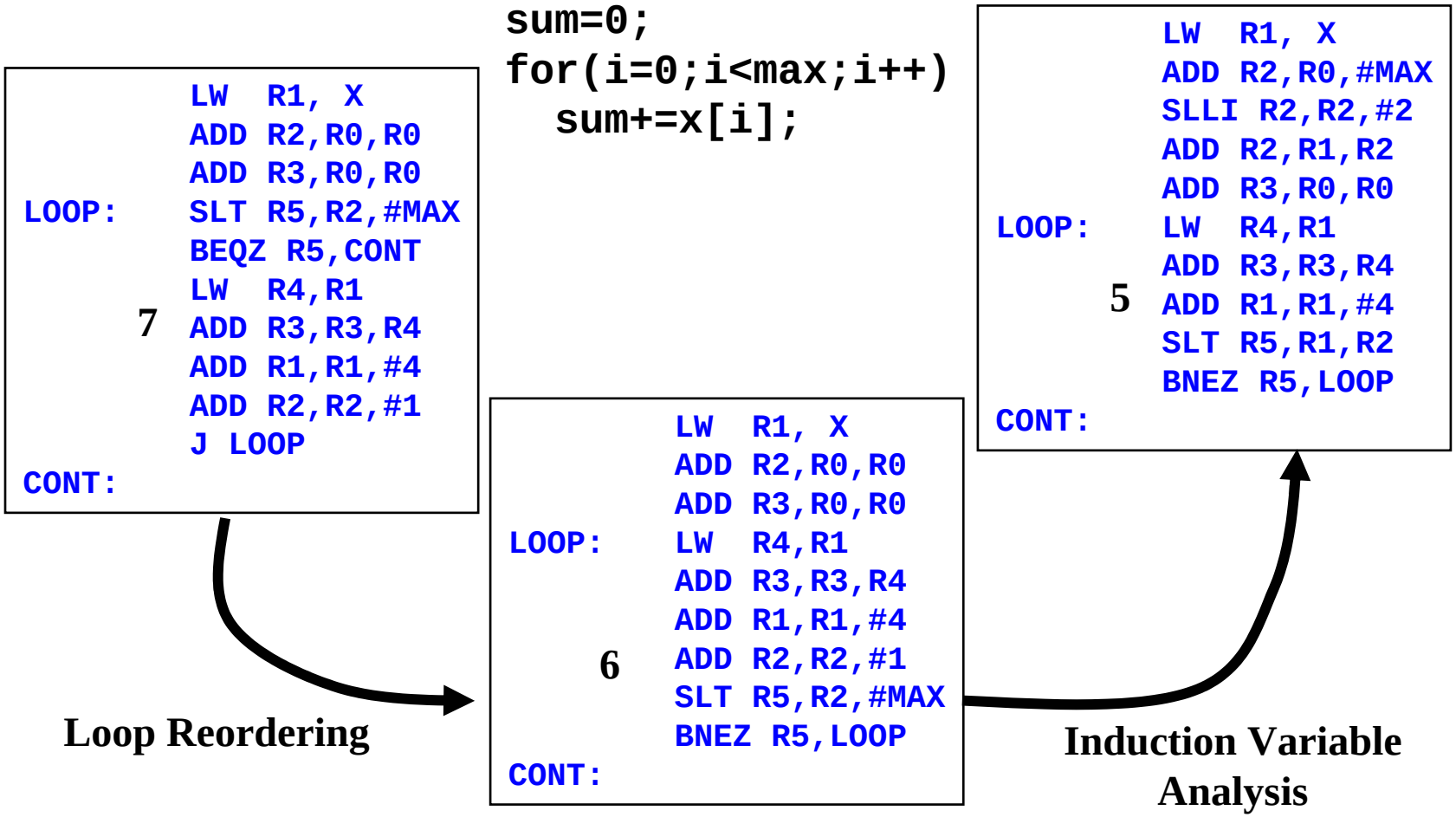
Review of ISA Principles

- Good ISA design
  - KISS! - only implement necessities (encodings, address modes, etc.)
  - FOG: Frequency, Orthogonality, Generality
- Instruction Types
  - ALU ops, Data movement, Control
- Addressing modes
  - Matched to program usage (local vars, globals, arrays)
- Program Control
  - Conditional/unconditional branches and jumps
  - Where to store conditions
  - PC relative and absolute

Role of the Optimizing Compiler



Example: Loop Optimization



Architect ⇒ Compiler Writer

- Simplify, Simplify, Simplify
  - Feature difficult to use, it won't be used….Less is More!
- Regularity
  - Common set of formats, few special cases
- Primitive, not solutions
  - CALLS vs. Fast register moves
- Make performance tradeoffs simple
- Ultimately, the ISA will *not* be perfect

Compiler ⇒ Microarchitecture

- Instruction Scheduling
  - Instruction Level Parallelism
- Resource Allocation
  - Registers (minimize spills/restores to and from memory)
- Memory optimizations
  - Cache conscious data organization
  - Code layout
- Etc……

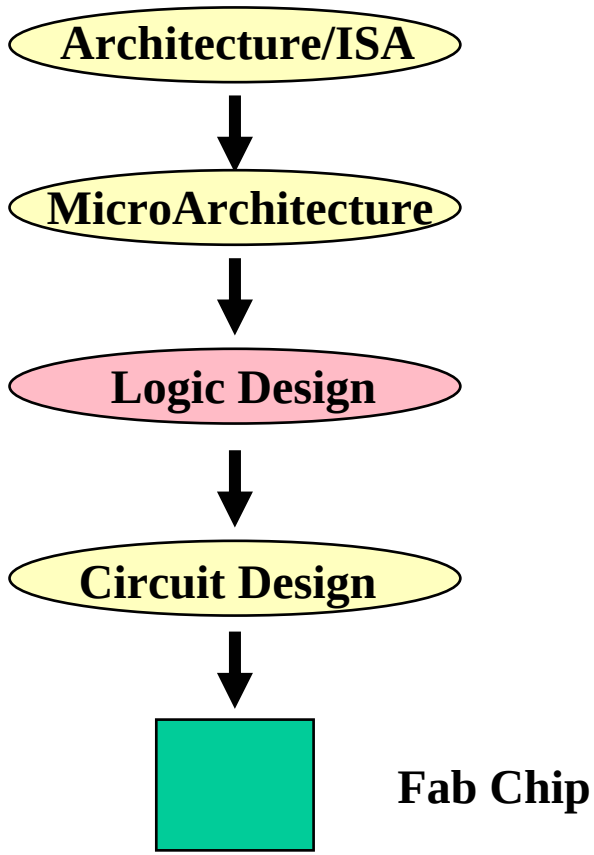
Building Blocks

- Arithmetic Units
  - adders, multipliers, dividers, shifters, ...
- Single Registers
- Register Files
- Memory Arrays
- Multiplexers
- Wires

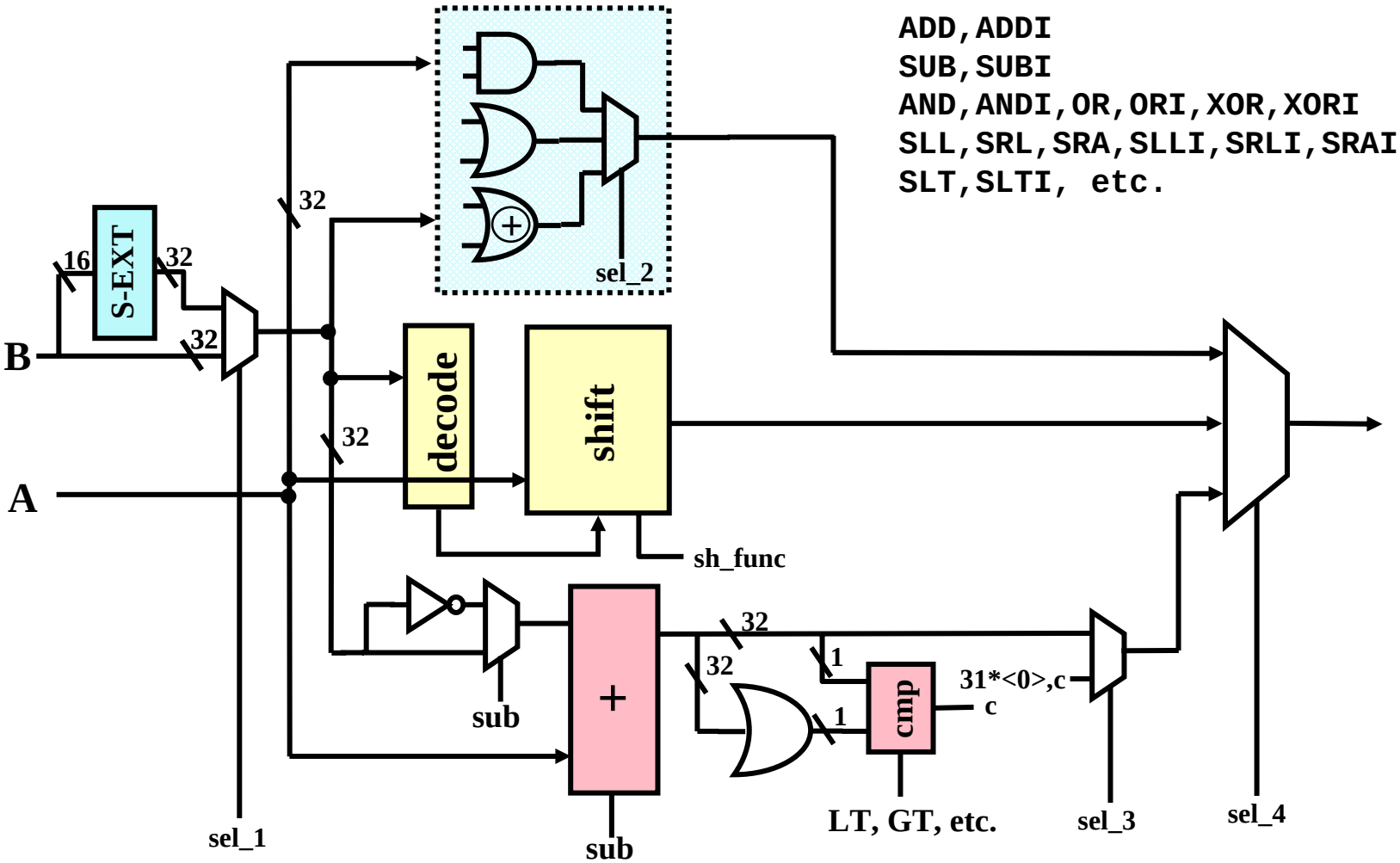
- Microarchitecture involves trading off area and delay of alternative organizations
- Units
  - area - tracks^2 (χ^2)
  - delay - fan-out of 4 inv (τ_4), "gate delay"
- Typically organized into datapaths
  - 10-20 tracks per bit slice
- Hard to estimate cost of control logic

What is Logic Design?

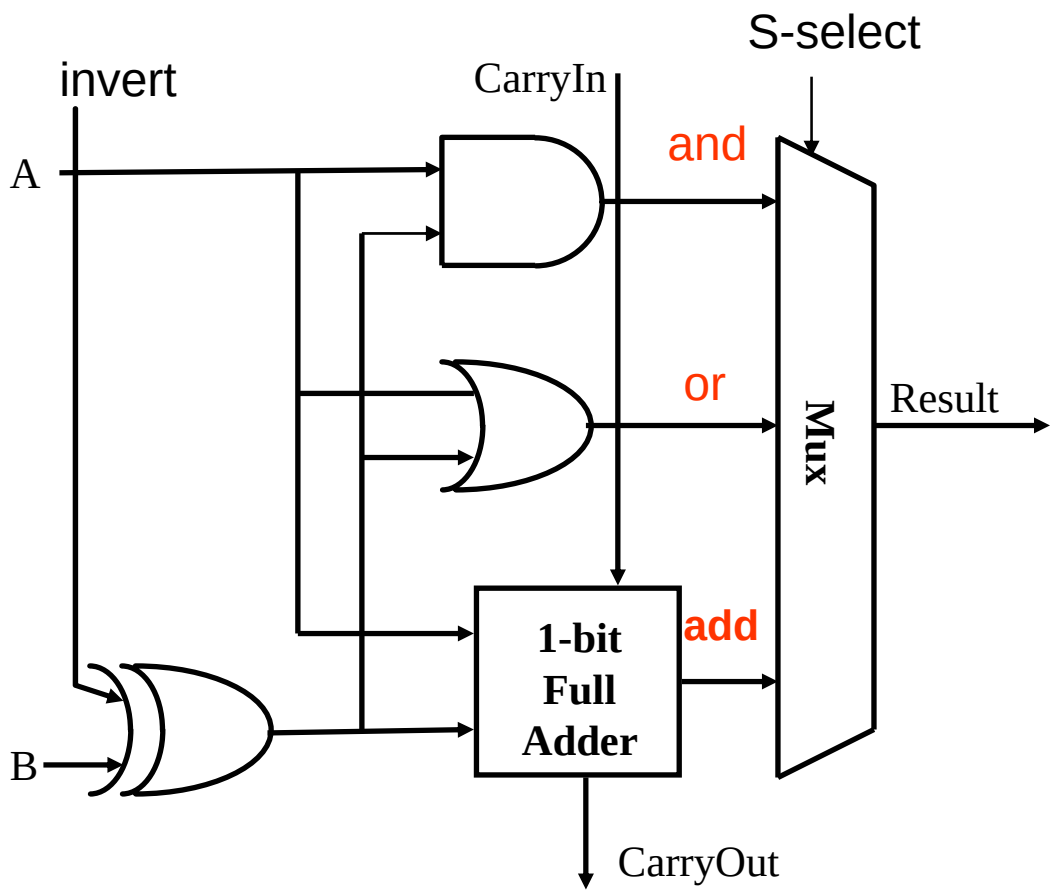
- Digital behavior of chip
- Specifies:
  - Actual states (ISA visible and non-visible)
  - Transitions between states
- Consists of:
  - Combinational logic (ie. ALUs)
  - Sequential logic ("memory")
    - Registers
    - State machines



Combinational Logic: The ALU



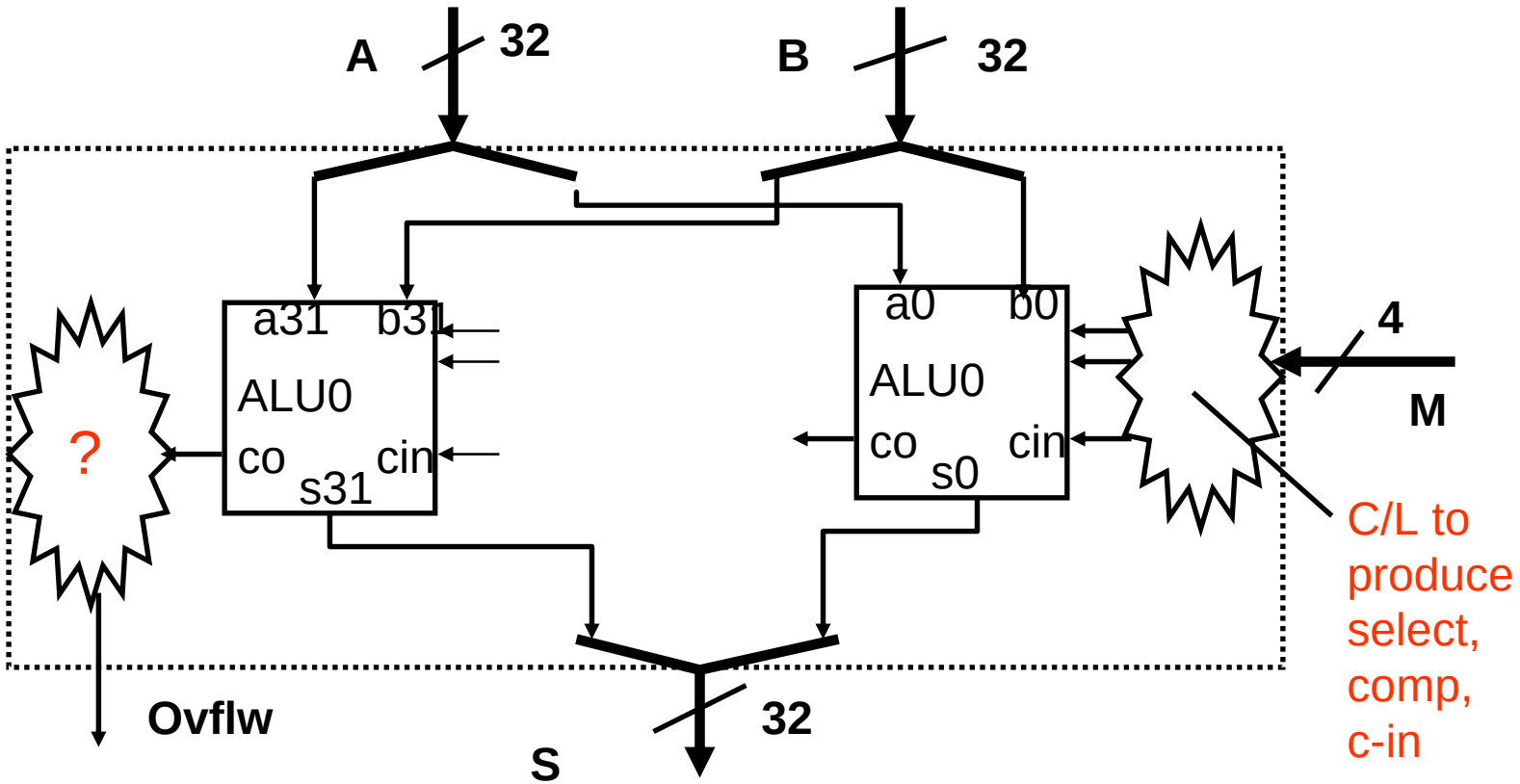
Bit Slice Approach to ALU design



Set-less-than? – left as an exercise

Bigger View of Bit Slicing

- LSB and MSB need to do a little extra

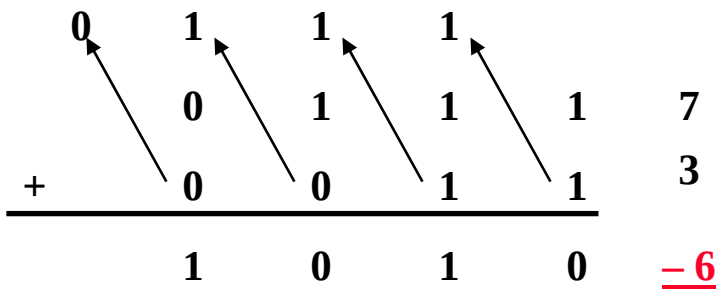


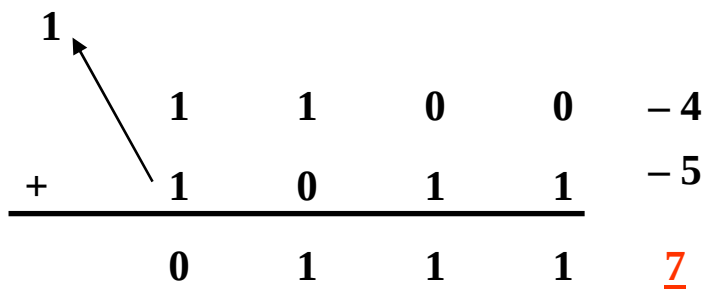
Overflow

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 ...

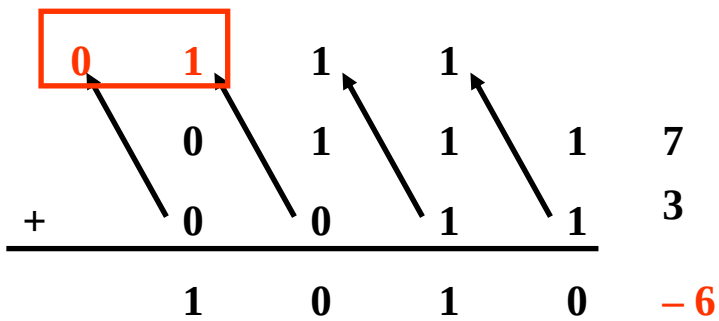
- 4 - 5 = - 9 but ...

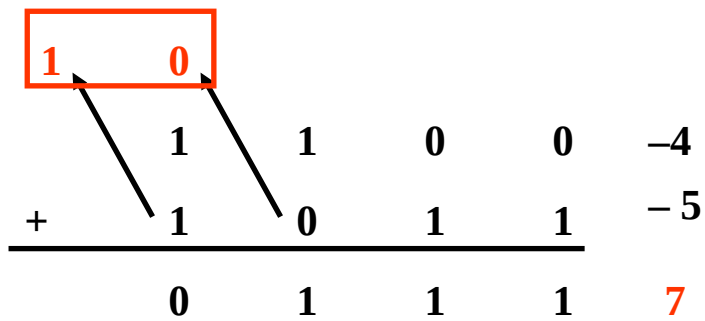




Overflow Detection

- 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 xor Carry out of MSB





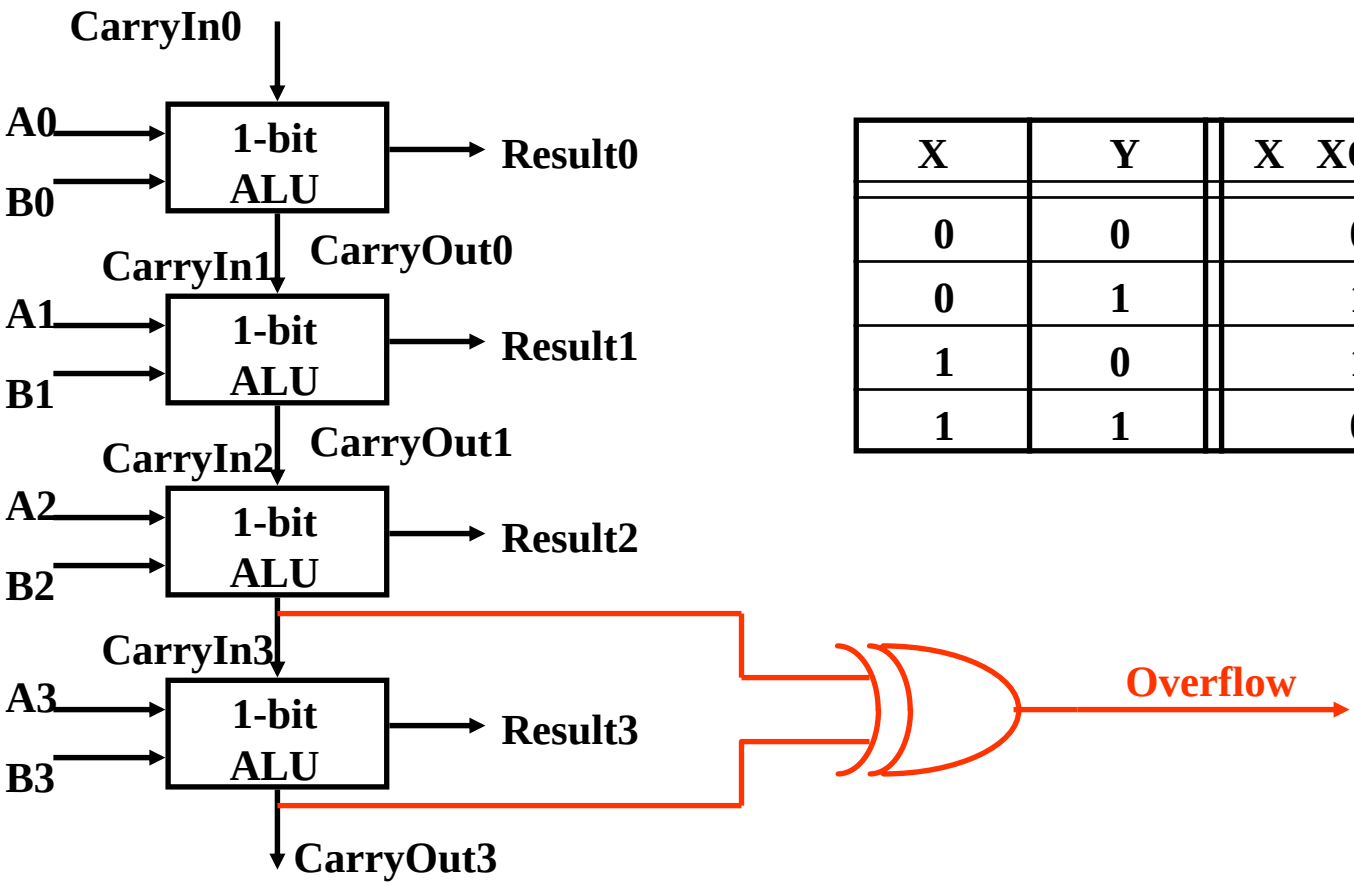
What's the difference between….

- …these instruction pairs
  - add/addu
  - addi/addiu
  - div/divu
  - sub/subu

- The unsigned versions don't check for overflow
  - But otherwise the arithmetic algorithm is the same

Overflow Detection Logic

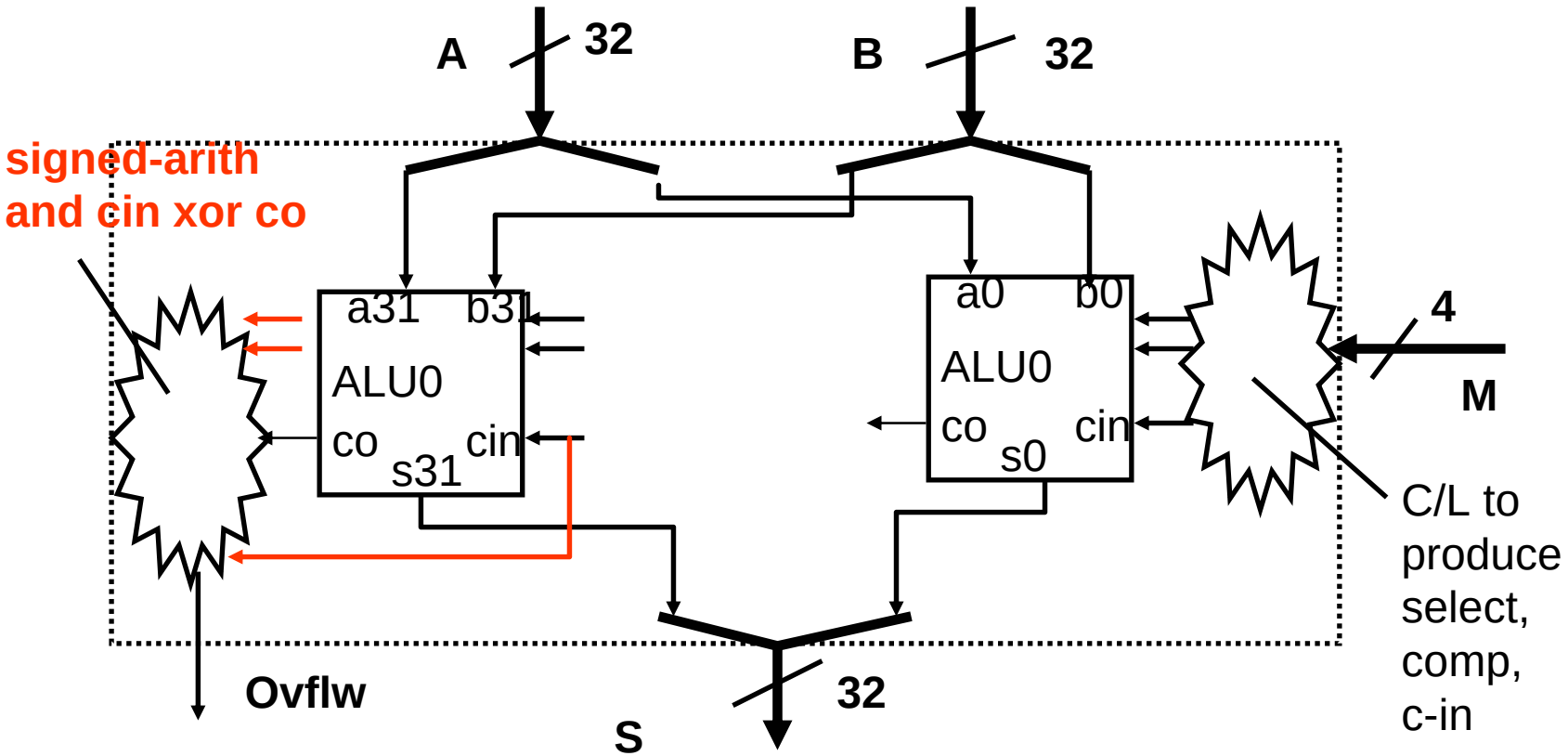
- Carry into MSB xor Carry out of MSB
  - For a N-bit ALU: Overflow = CarryIn[N - 1] XOR CarryOut[N - 1]



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

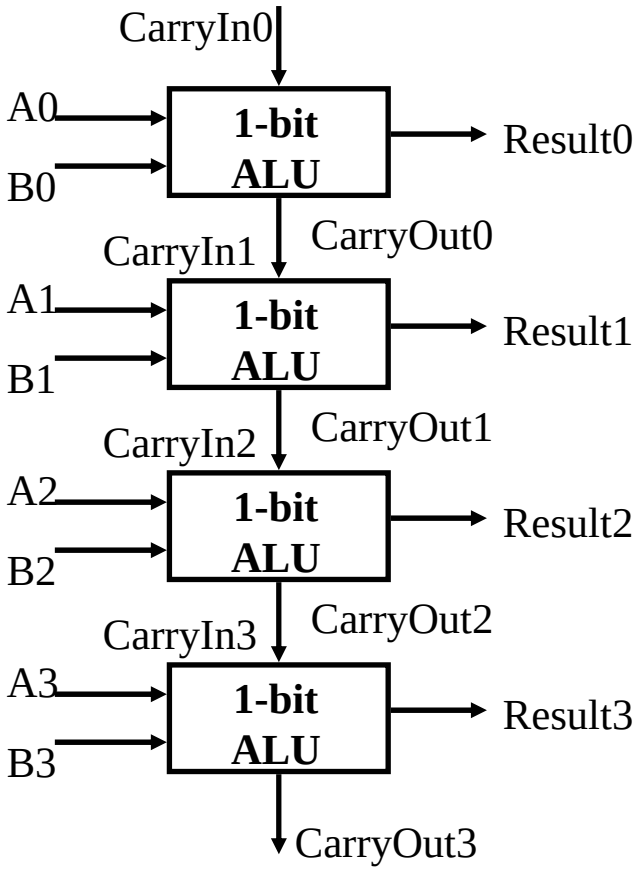
More Revised Diagram

- LSB and MSB need to do a little extra



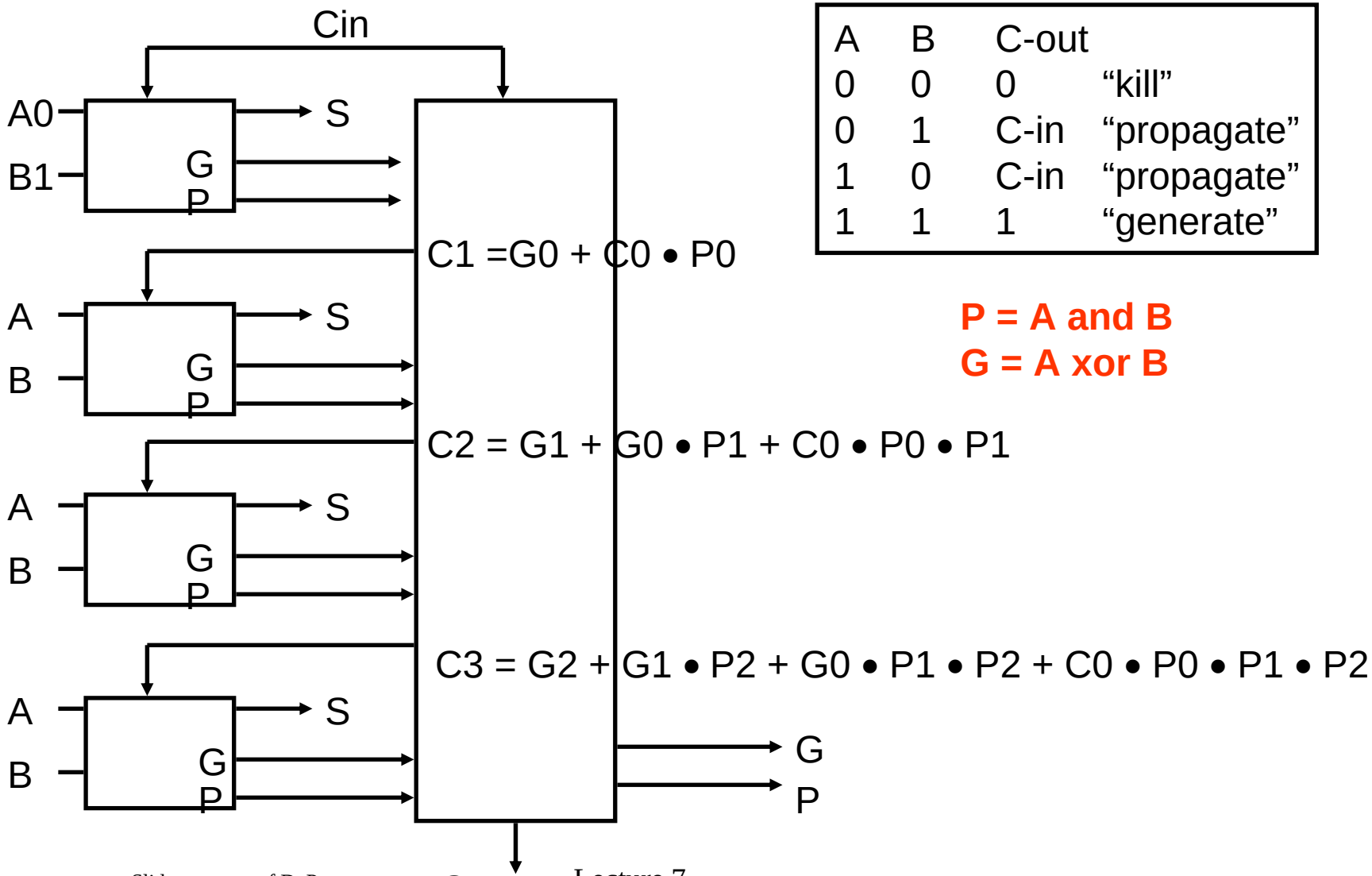
But What about Performance?

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



Design Trick: throw hardware at it

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

Summary

- ISA principles
- Compiler/ISA interaction
- Computer arithmetic (add/shift)

- Next Time
  - RISC vs. CISC
  - Computer arithmetic
  - Memory
  - Simple pipeline

- Reading assignment – P&H 4.1-4.7