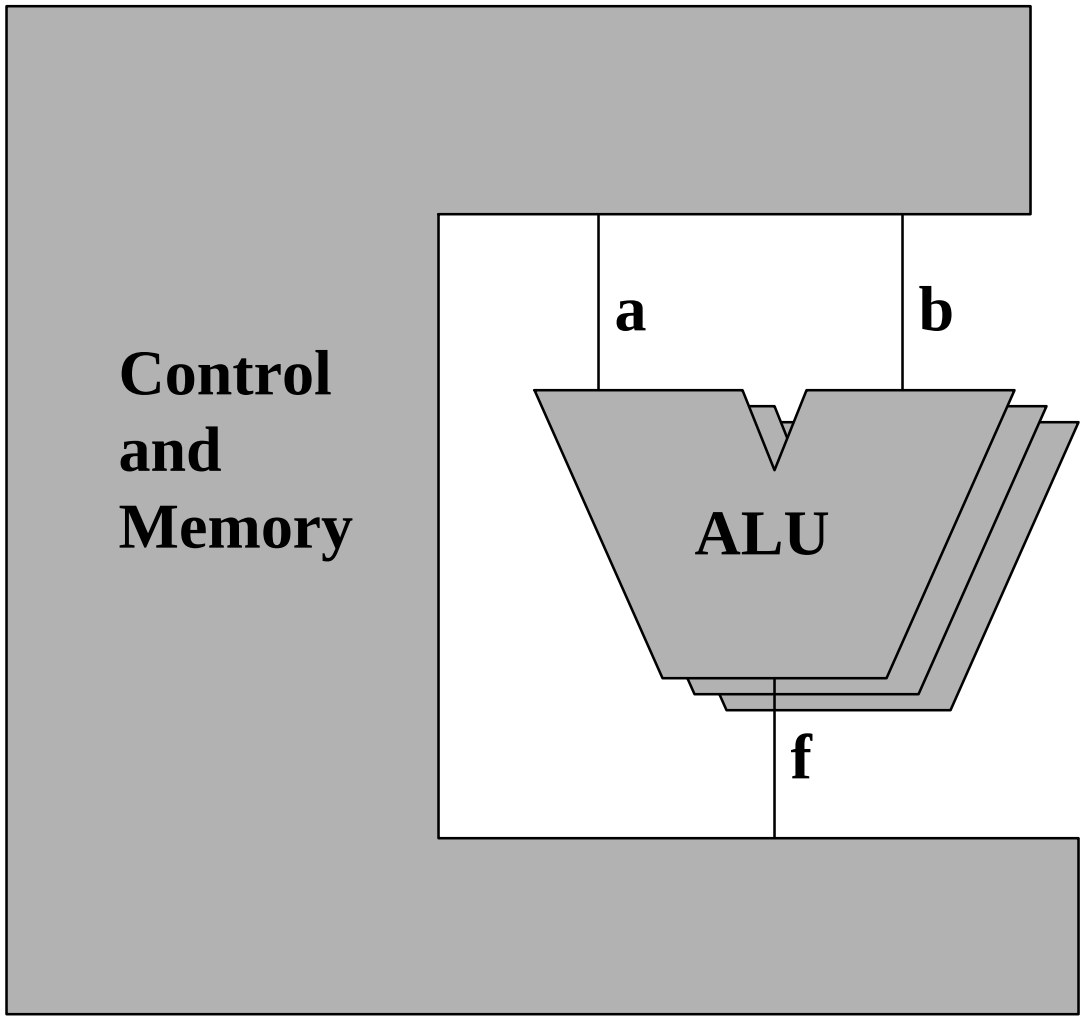
Arithmetic circuits

- Excellent examples of multi-level combinational logic design
- Time vs. space trade-offs
  - doing things fast may require more logic and thus more space
- Arithmetic and logic units
  - general-purpose building blocks
  - critical components of micro-processor
  - used within most computer instructions

Heart of a Computer



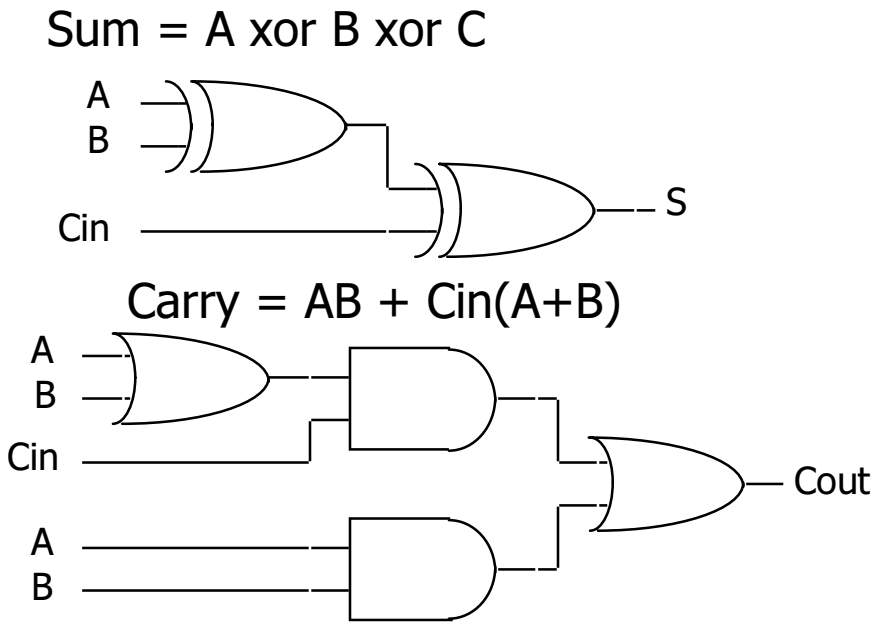
ALU =
Arithmetic and
Logic Unit

ADD (A+B)
- **INC (A+1)**
- **DEC (A+ -1)**
- **SUB (A+ -B)**
- **CMP (A+ -B)**
- **COMP(0+ -A)**
- **PASS (A + 0)**

SHIFT
AND
OR
XOR

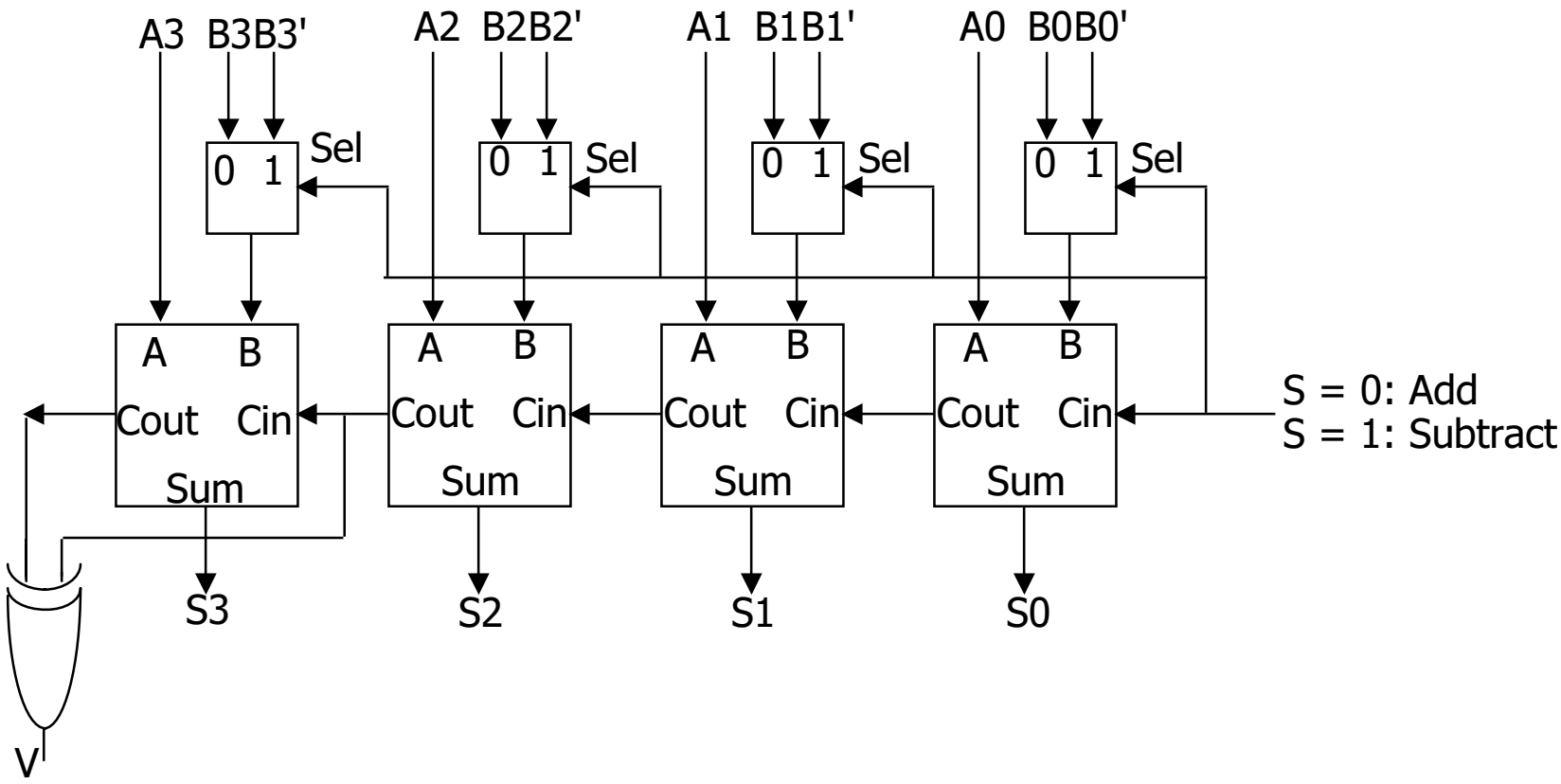
Full adder implementations

- Standard approach
  - 6 gates
  - 2 XORs, 2 ANDs, 2 ORs



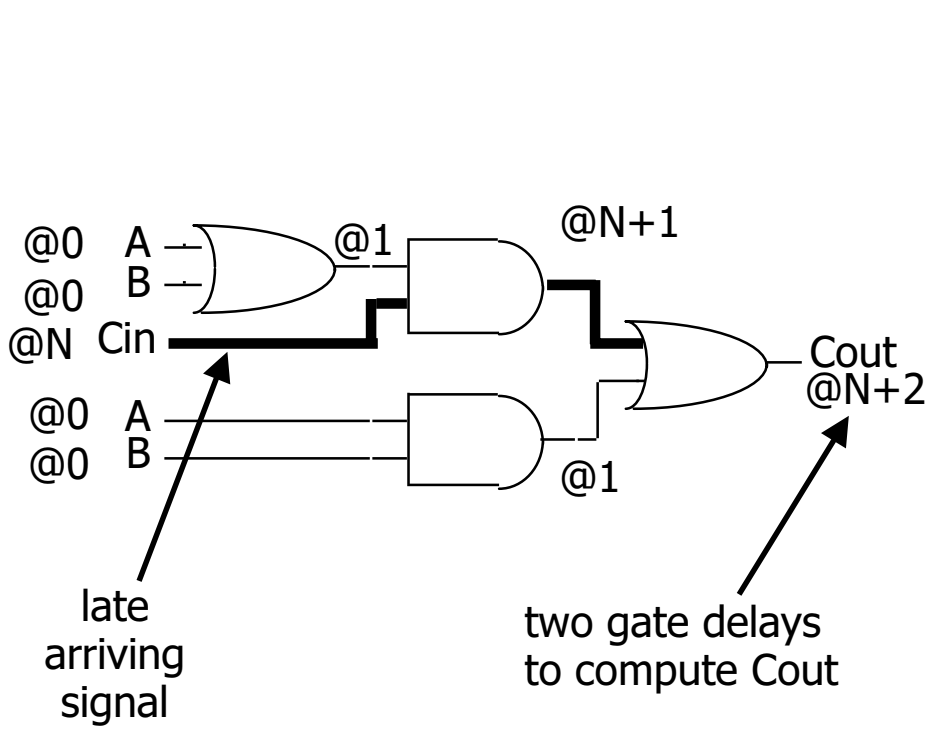
Adder/subtractor

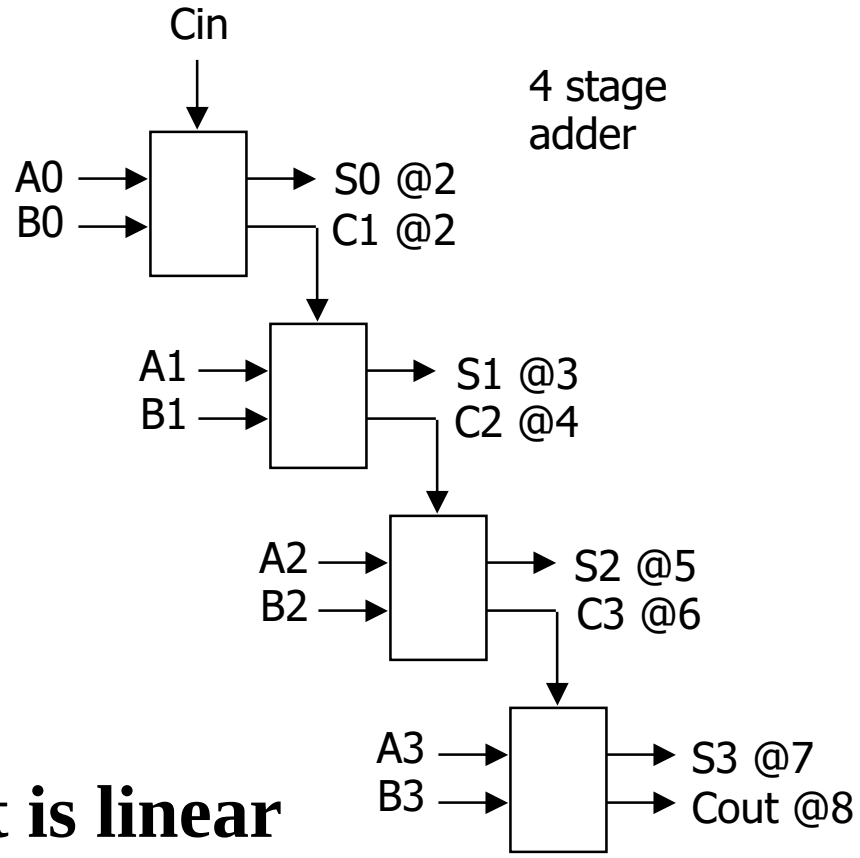
- Use an adder to do subtraction thanks to 2s complement representation
  - A – B = A + (– B) = A + B' + 1
  - Use function select (S) to choose add or subtract



Ripple-carry adders

- Critical delay
  - the propagation of carry from low to high order stages

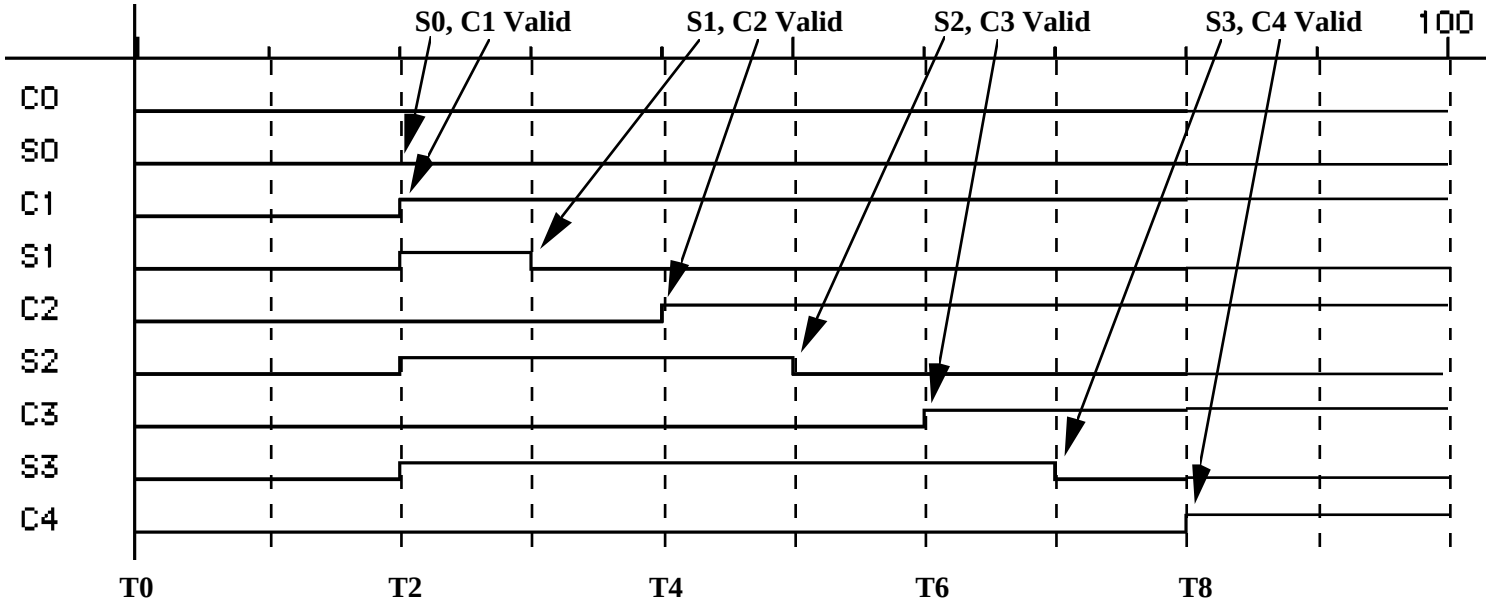




Delay is linear, Cost is linear

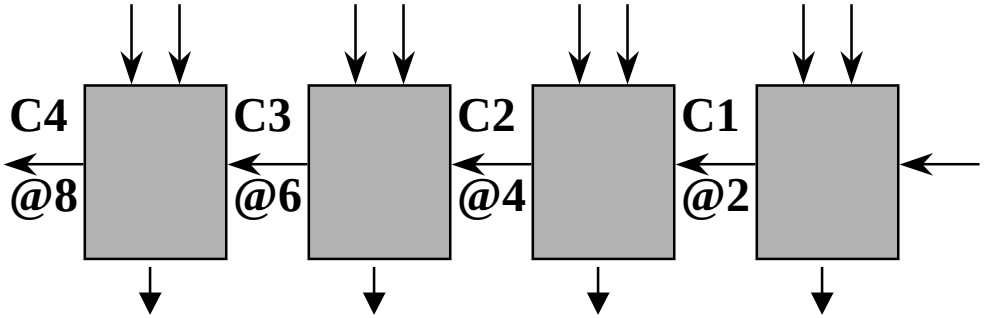
Ripple-carry adders (cont'd)

- Critical delay
  - the propagation of carry from low to high order stages
  - 1111 + 0001 is the worst case addition
  - carry must propagate through all bits

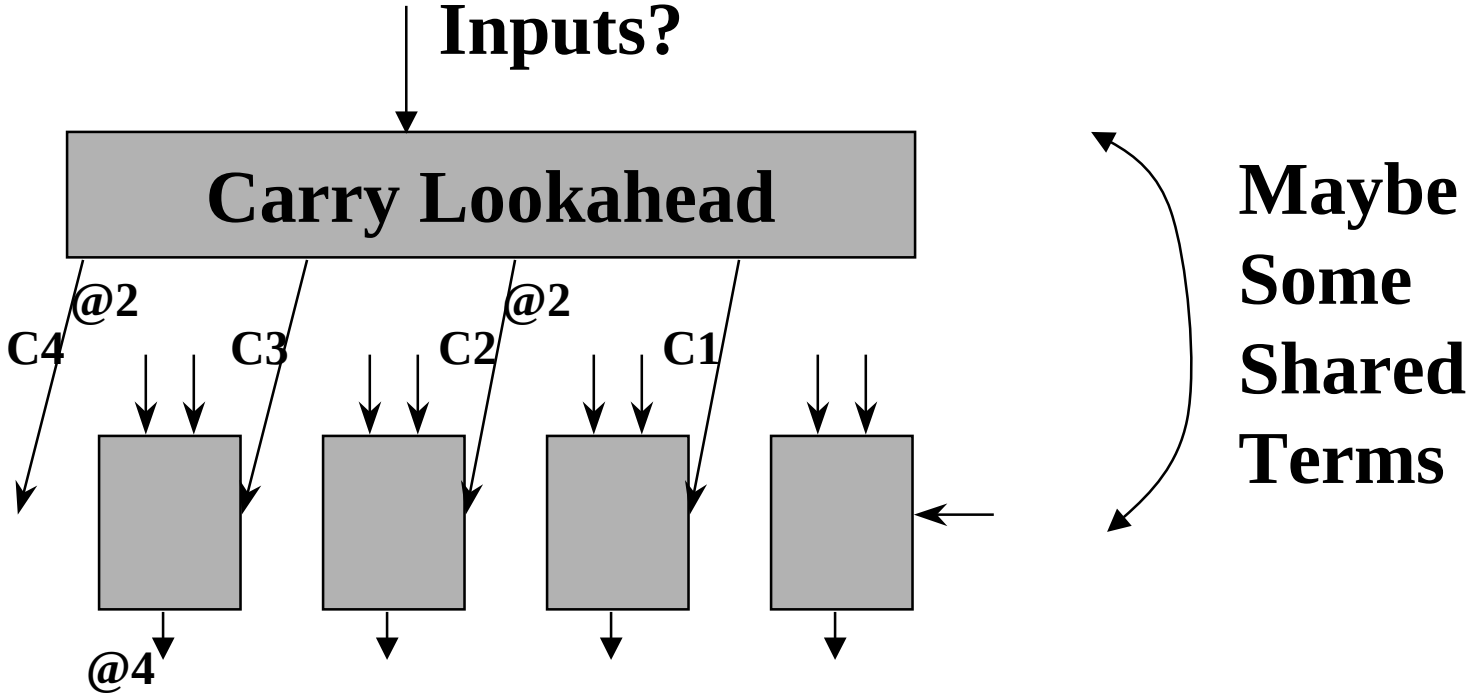


Need a Fast Adder

Size is linear, speed is linear



Find a 2-level implementation for each carry



Carry-lookahead logic

- The key is the carry. Compute as much as possible about Carry out before you get the carry in.
- Intutively Ci+1 is true of Ai and Bi generate and carry or if Ci is true and Ai and Bi propogate a carry
- Carry generate: Gi = Ai Bi
  - must generate carry when A = B = 1
- Carry propagate: Pi = Ai xor Bi
  - carry-in will equal carry-out here
- Sum and Cout can be re-expressed in terms of generate/propagate:
  - Si = Ai xor Bi xor Ci
    = Pi xor Ci
  - Ci+1 = Ai Bi + Ai Ci + Bi Ci
    = Ai Bi + Ci (Ai + Bi)
    = Ai Bi + Ci (Ai xor Bi)
    = Gi + Ci Pi

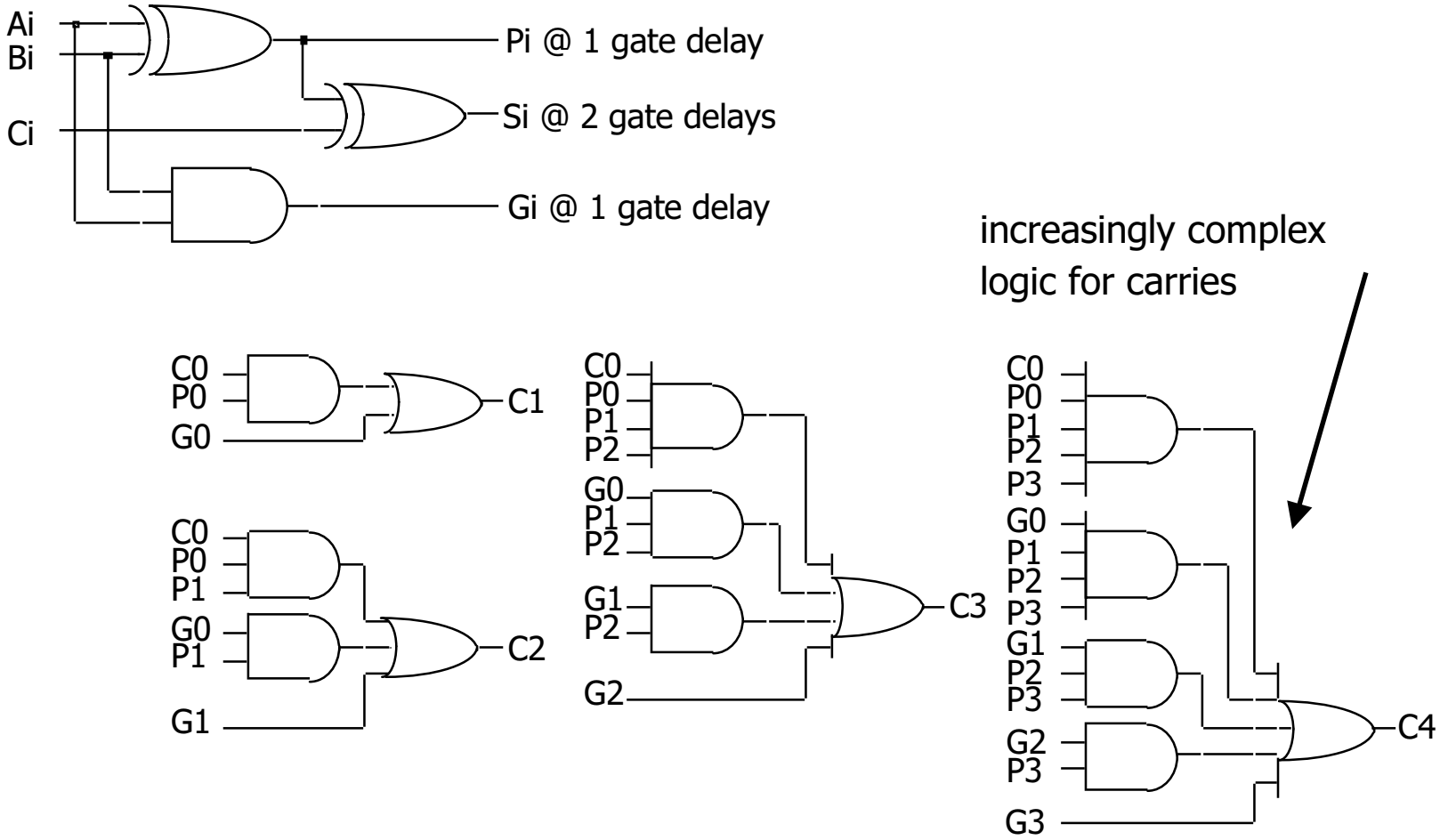
Carry-lookahead logic (cont'd)

- Re-express the carry logic as follows:
  - $C1 = G0 + P0\ C0$
  - $C2 = G1 + P1\ C1 = G1 + P1\ G0 + P1\ P0\ C0$
  - $C3 = G2 + P2\ C2 = G2 + P2\ G1 + P2\ P1\ G0 + P2\ P1\ P0\ C0$
  - $C4 = G3 + P3\ C3 = G3 + P3\ G2 + P3\ P2\ G1 + P3\ P2\ P1\ G0 + P3\ P2\ P1\ P0\ C0$

- Each of the carry equations can be implemented with two-level logic (in terms of P and G)
  - all inputs are now directly derived from data inputs and not from intermediate carries
  - this allows computation of all sum outputs to proceed in parallel

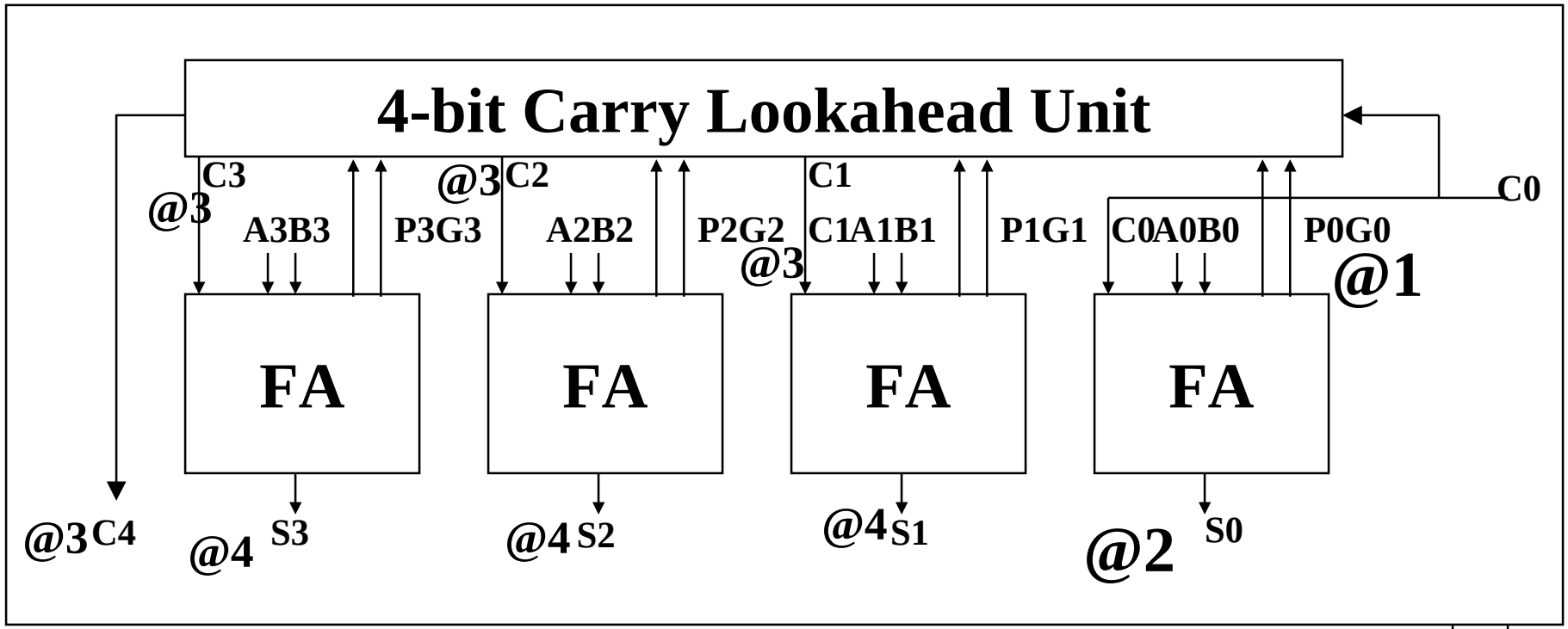
Carry-lookahead implementation

- Adder with propagate and generate outputs



Carry-lookahead implementation (cont'd)

- Carry-lookahead logic generates individual carries
  - sums computed much more quickly in parallel
  - however, cost of carry logic increases with more stages



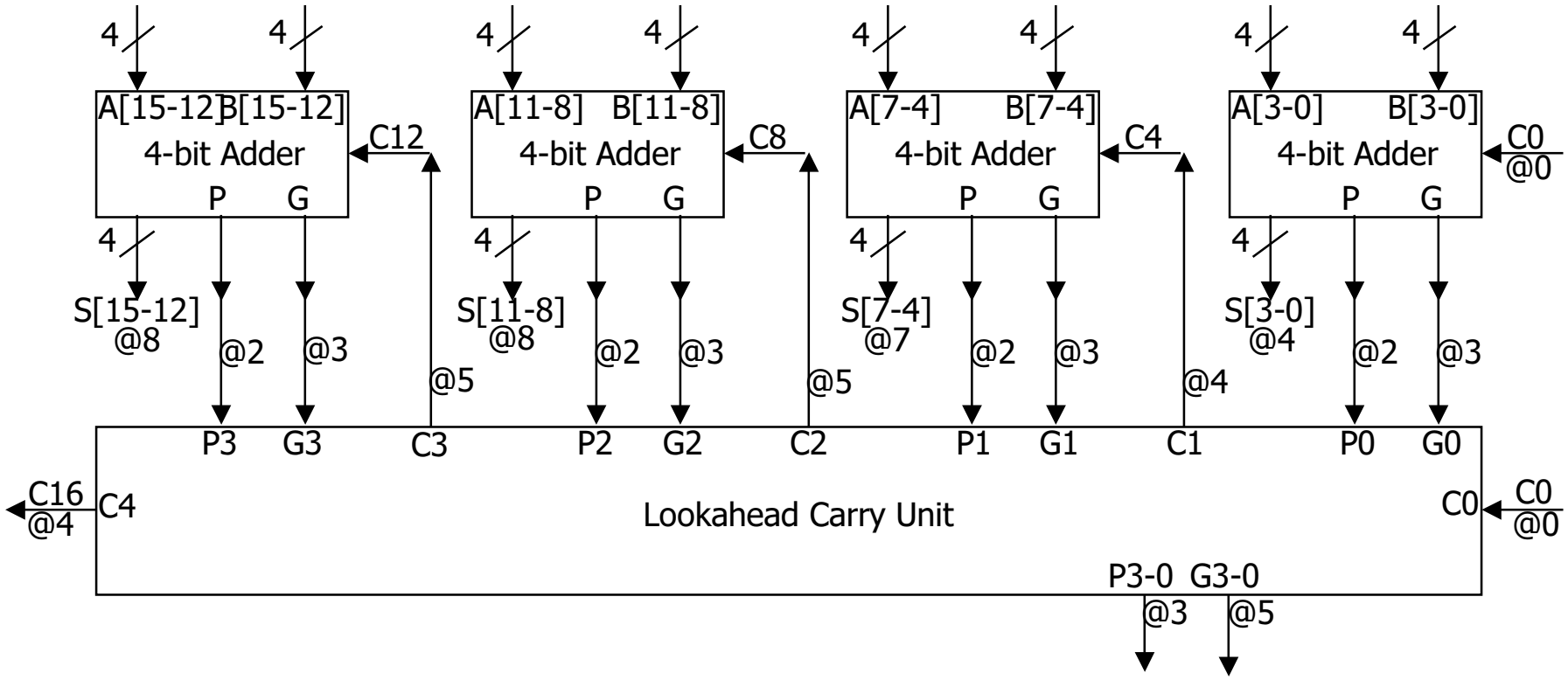
G4 = G3+P3G2+P3P2G1 + P3P2P1G0 @3

P4 = P3P2P1P0 @2

PG

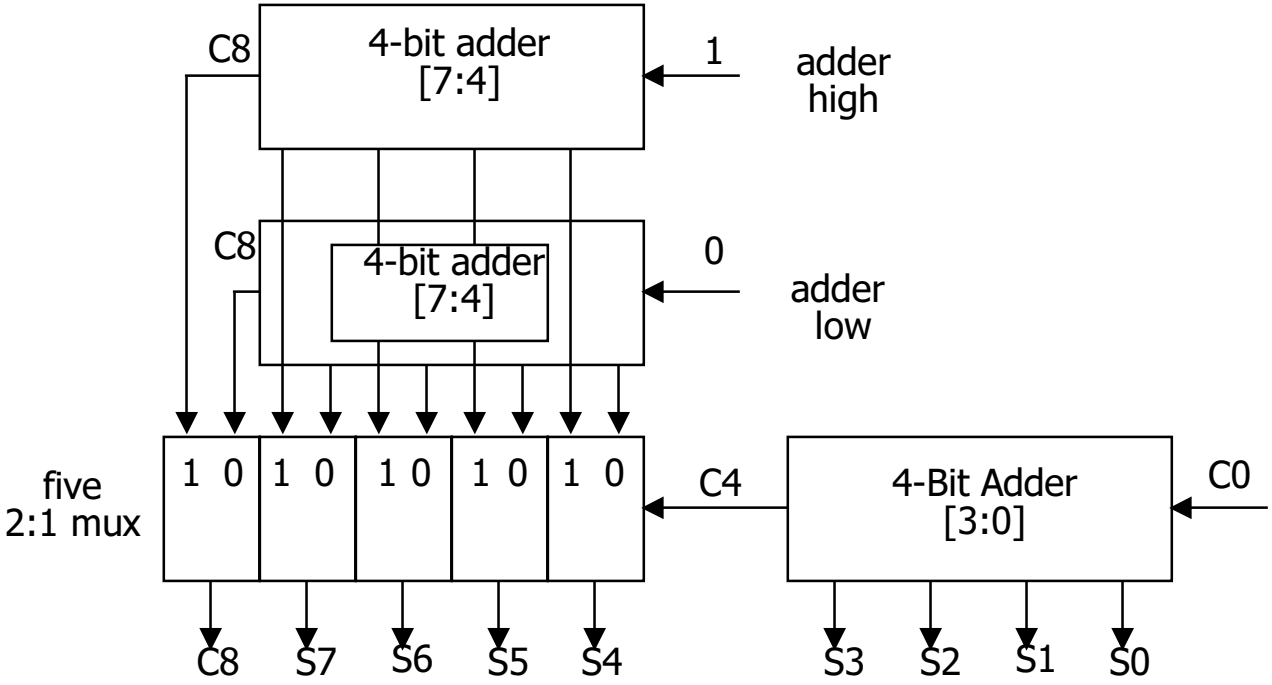
Carry-lookahead adder with cascaded carry-lookahead logic

- Carry-lookahead adder
  - 4 four-bit adders with internal carry lookahead
  - second level carry lookahead unit extends lookahead to 16 bits

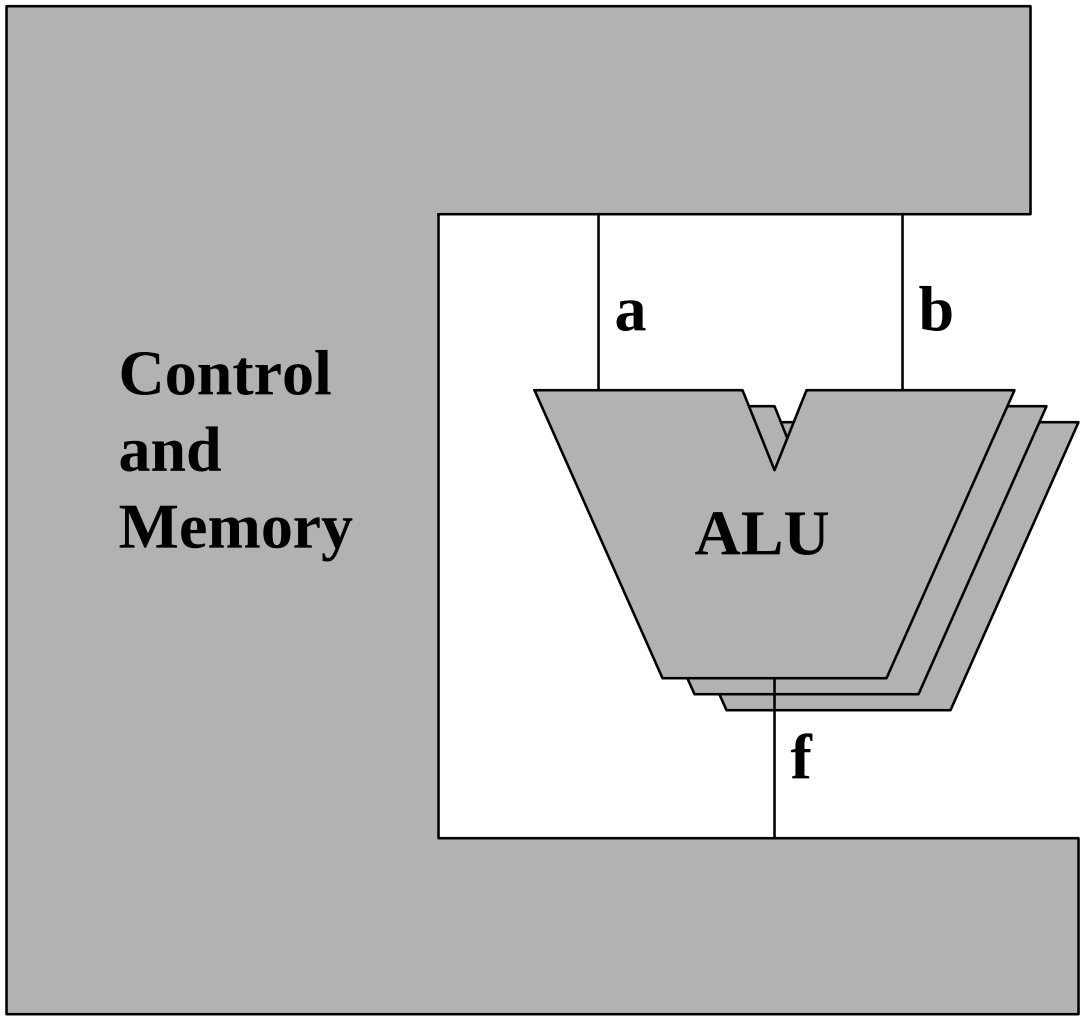


Carry-select adder

- Redundant hardware to make carry calculation go faster
  - compute two high-order sums in parallel while waiting for carry-in
  - one assuming carry-in is 0 and another assuming carry-in is 1
  - select correct result once carry-in is finally computed



Heart of a Computer



ALU =
Arithmetic and
Logic Unit

ADD (A+B)
- **INC (A+1)**
- **DEC (A+ -1)**
- **SUB (A+ -B)**
- **CMP (A+ -B)**
- **COMP(0+ -A)**
- **PASS (A + 0)**

SHIFT
AND
OR
XOR

Arithmetic logic unit design specification

M = 0, logical bitwise operations

S1	S0	Function	Comment
0	0	Fi = Ai	input Ai transferred to output
0	1	Fi = not Ai	complement of Ai transferred to output
1	0	Fi = Ai xor Bi	compute XOR of Ai, Bi
1	1	Fi = Ai xnor Bi	compute XNOR of Ai, Bi

M = 1, C0 = 0, arithmetic operations

0	0	F = A	input A passed to output
0	1	F = not A	complement of A passed to output
1	0	F = A plus B	sum of A and B
1	1	F = (not A) plus B	sum of B and complement of A

M = 1, C0 = 1, arithmetic operations

0	0	F = A plus 1	increment A
0	1	F = (not A) plus 1	twos complement of A
1	0	F = A plus B plus 1	increment sum of A and B
1	1	F = (not A) plus B plus 1	B minus A

logical and arithmetic operations
not all operations appear useful, but "fall out" of internal logic

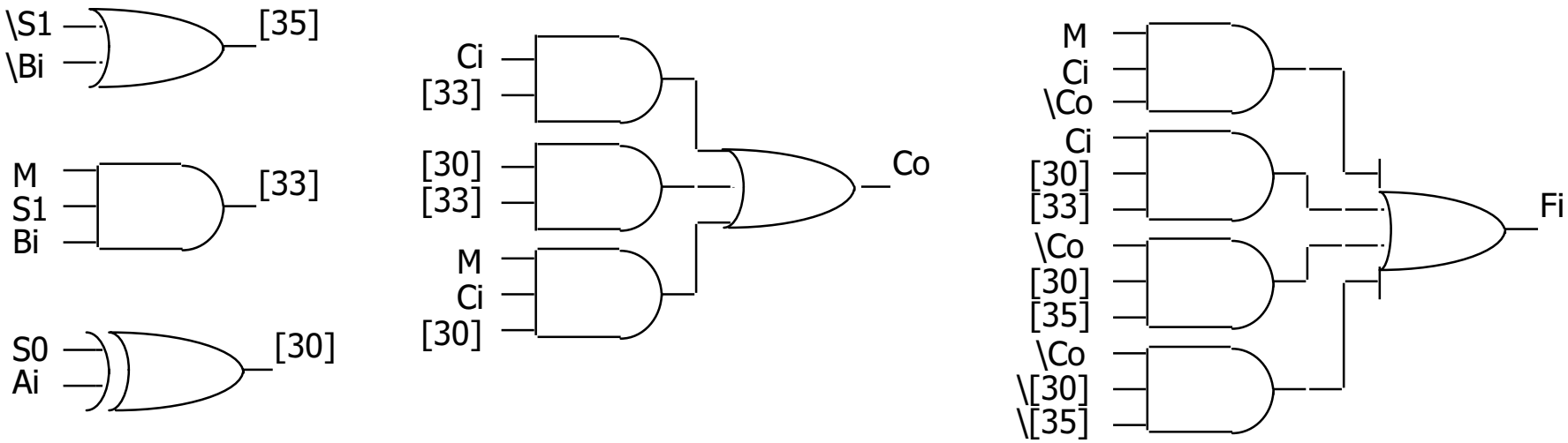
Arithmetic logic unit design (cont'd)

- Sample ALU – truth table

M	S1	S0	Ci	Ai	Bi	Fi	Ci+1
0	0	0	X	0	X	0	X
			X	1	X	1	X
	0	1	X	0	X	1	X
			X	1	X	0	X
	1	0	X	0	0	0	X
			X	0	1	1	X
			X	1	0	1	X
			X	1	1	0	X
	1	1	X	0	0	1	X
			X	0	1	0	X
			X	1	0	0	X
			X	1	1	1	X
1	0	0	0	0	X	0	X
			0	1	X	1	X
	0	1	0	0	X	1	X
			0	1	X	0	X
	1	0	0	0	0	0	0
			0	0	1	1	0
			0	1	0	1	0
			0	1	1	0	1
	1	1	0	0	0	1	0
			0	0	1	0	1
			0	1	0	0	0
			0	1	1	1	0
1	0	0	1	0	X	1	0
			1	1	X	0	1
	0	1	1	0	X	0	1
			1	1	X	1	0
	1	0	1	0	0	1	0
			1	0	1	0	1
			1	1	0	0	1
			1	1	1	1	1
	1	1	1	0	0	0	1
			1	0	1	1	1
			1	1	0	1	0
			1	1	1	0	1

Arithmetic logic unit design (cont'd)

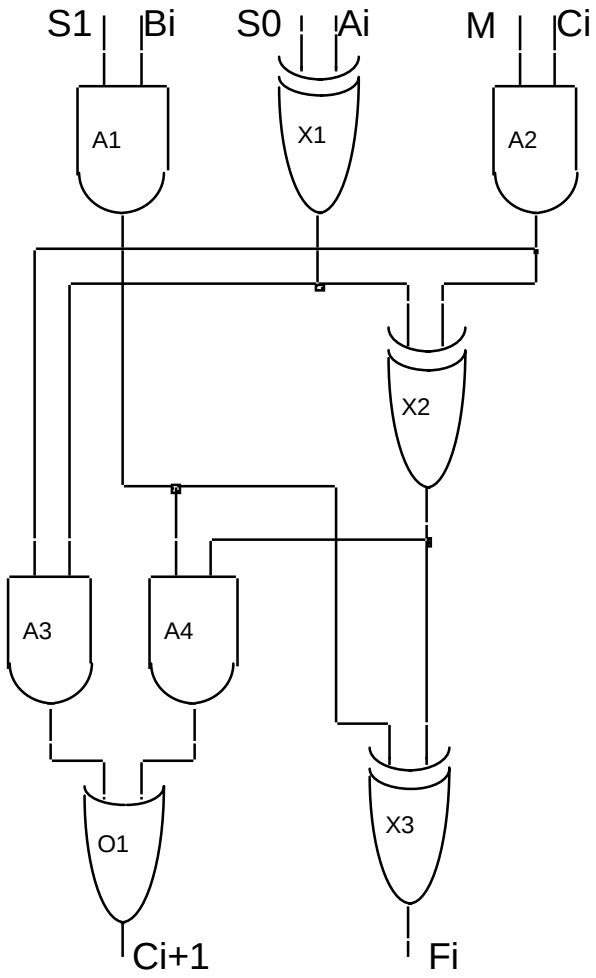
- Sample ALU – multi-level discrete gate logic implementation



12 gates

Arithmetic logic unit design (cont'd)

- Sample ALU – clever multi-level implementation

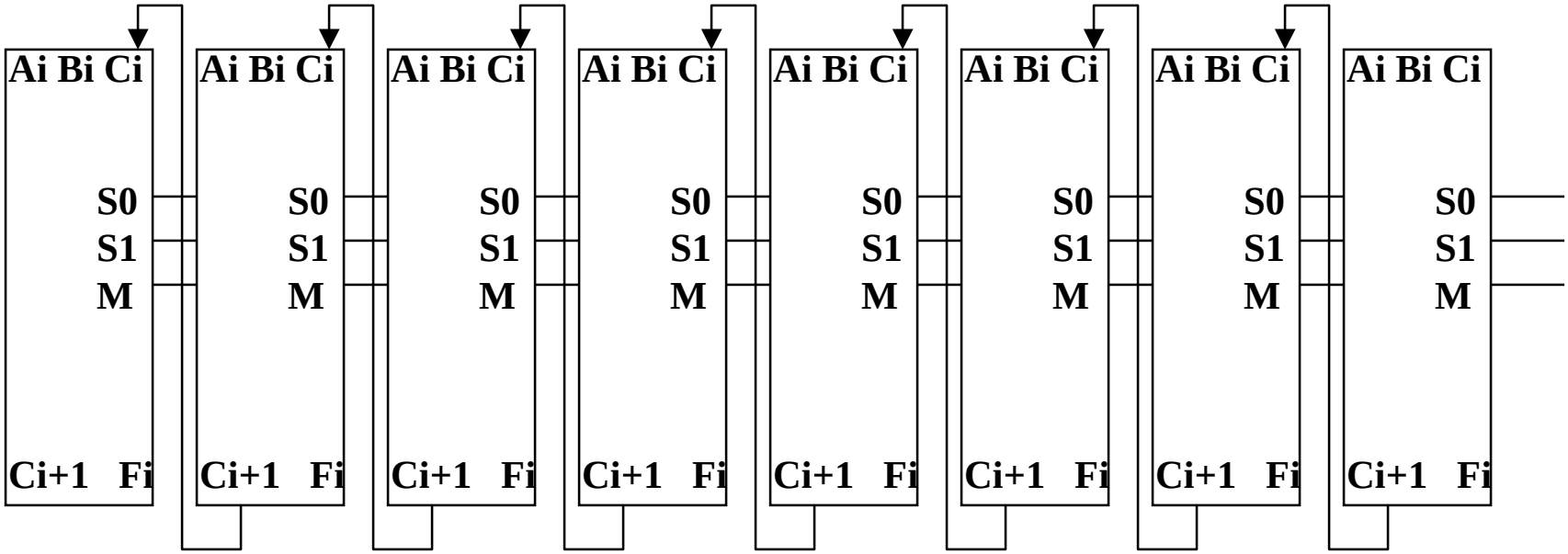


first-level gates
  use S0 to complement Ai
    S0 = 0 causes gate X1 to pass Ai
    S0 = 1 causes gate X1 to pass Ai'
  use S1 to block Bi
    S1 = 0 causes gate A1 to make Bi go forward as 0
            (don't want Bi for operations with just A)
    S1 = 1 causes gate A1 to pass Bi
  use M to block Ci
    M = 0 causes gate A2 to make Ci go forward as 0
           (don't want Ci for logical operations)
    M = 1 causes gate A2 to pass Ci

other gates
  for M=0 (logical operations, Ci is ignored)
    Fi = S1 Bi xor (S0 xor Ai)
       = S1'S0' (Ai) + S1'S0 (Ai') +
          S1 S0' (Ai Bi' + Ai' Bi) + S1 S0 (Ai' Bi' + Ai Bi)
  for M=1 (arithmetic operations)
    Fi = S1 Bi xor ((S0 xor Ai) xor Ci) =
    Ci+1 = Ci (S0 xor Ai) + S1 Bi ((S0 xor Ai) xor Ci) =

    just a full adder with inputs S0 xor Ai, S1 Bi, and Ci

8-Bit Bit-Sliced ALU



Carry lookahead techniques can be applied to this too

Summary for examples of Math Hardware

- Binary number representation
  - positive numbers the same
  - difference is in how negative numbers are represented
  - 2s complement easiest to handle: one representation for zero, slightly complicated complementation, simple addition
- Circuits for binary addition
  - basic half-adder and full-adder
  - carry lookahead logic
  - carry-select
- ALU Design
  - specification, implementation