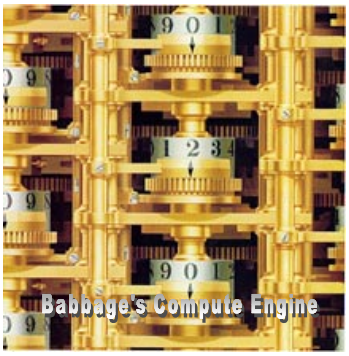
18-347 Lecture 5

Computer Arithmetic I: Adders & Shifters



Prof. Rob Rutenbar
rutenbar@ece.cmu.edu
http://www.ece.cmu.edu/~ece347

Note bug fixes on a few slides, as done in lecture…

Where Are We?

	M	T	W	Th	F	
Jan	15	16	17	18	19	1
	22	23	24	25	26	2
Feb	29	30	31	1	2	3
	5	6	7	8	9	4
	12	13	14	15	16	5
	19	20	21	22	23	6
Mar	26	27	28	1	2	7
	5	6	7	8	9	8
	12	13	14	15	16	9
	19	20	21	22	23	10
Spring Break	26	27	28	29	30	11
Apr	2	3	4	5	6	12
	9	10	11	12	13	13
	16	17	18	19	20	14
	23	24	25	26	27	15
May	30	1	2	3	4	16

- We've seen the *programmer's* view
- Now we'll see the *hardware designers* view
- Today:
  - Adders & shifters
- Monday:
  - Multipliers

Readings for the Week/Announcements

- **Today**
  - Chapter 4, Sections 4.1-4.5
- **Wednesday**
  - Chapter 4, Section 4.6
- **Readings for each lecture: on the class web page**
  - **http://www.ece.cmu.edu/~ece347/lectures**

Computer Arithmetic—*Why Bother?*

- **Computer architecture sounds "*cool*"**
  - Easy to impress your friends, potential employers, Mom
- **Computer arithmetic sounds "*not*"**
  - Sounds remedial, low-level, tedious
- **So…why do this? 3 big reasons**
  - Lots of microarchitecture ends up composed of fast adders, shifters, etc,
  - Increasing number of applications depend on fast or special computation
    - Scientific apps – predicting the weather; media apps – mpeg, mp3
  - You don't know how to build the very fast components we need to use today
    - There are *standard* digital designs for fast adders, shifters, etc.
    - Present several interesting speed/complexity tradeoffs

Today's Menu:

- **Stuff we assume you remember**
  - Basic signed representations, basic ripple-carry adders
- **Stuff we assume you don't remember (or never saw)**
  - Fast adder design—basic lookahead carry architectures
  - Recursive lookahead architectures for very wide, fast adders
- **New stuff**
  - ALU design—for the MIPS ISA
  - Shifter design

Basics: Two's Complement Numbers

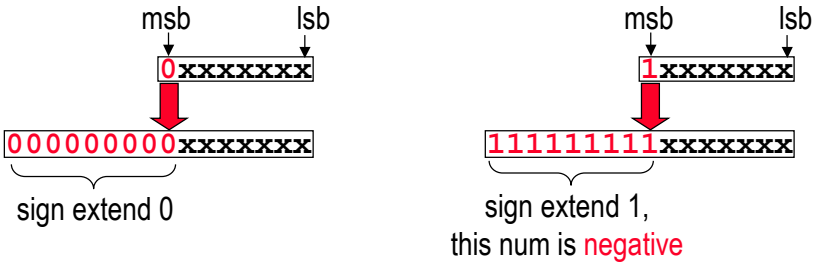
- **2s comp. encodes negative nums via an arithmetic *transform***
  - Like a regular, weighted binary representation, *but most significant bit weight is negative*
  - For example, for 32 bits

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = +2_{ten}$
...
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +2{,}147{,}483{,}647_{ten}$ *maxint*
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -2{,}147{,}483{,}647_{ten}$ *minint*
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -2{,}147{,}483{,}646_{ten}$
...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1_{ten}$

$$b_{31}(-2^{31}) + b_{30}(2^{30}) + b_{29}(2^{29}) + \ldots + b_1(2^1) + b_0(2^0)$$

Two's Complement Operations

- **Negating a 2s complement number:** ***invert all bits and add 1***
  - Remember: "negate" and "invert" are quite different!
- **Converting n-bit numbers into numbers with more than n bits:**
  - You have to do sign extension: *copy* 2s comp sign bit into higher order bits



Application in the MIPS ISA

- **Arithmetic on MIPS 16 bit immediates**
  - MIPS 16 bit immediate gets converted to 32 bits 2s complement for arithmetic
- **MIPS ISA weirdness…**
  - MIPS instruction *add immediate unsigned* addiu sign-extends it 16-bit immediate field
  - This is not what the name suggests the instruction does
  - Despite its name, addiu is used to add constants to signed integers when we don't care about overflow (more later – ie, when the num gets too big or too negative)
  - MIPS has no *subtract immediate* instruction and negative nums need sign extension, so the MIPS architects decided to sign-extend the immediate field to make it possible to do a sort of "subtract immediate" by adding a negative 16bit immediate

Basics: Binary Addition & Subtraction

- **Just like in grade school (carry/borrow 1s)**

```
   0111          0111          0110
 + 0110        - 0110        - 0101
```

- **Two's complement operations easy**
  - Subtraction accomplished by doing addition of negative numbers

```
        0111   <=  positive 7
  +     1010   <=  negative 6
carry=> 1 0001   <=  positive 1, and we usually ignore carry/borrow out
```

- **….except in cases of overflow and underflow**
  - Overflow: result too positive (too big) for finite computer word)
  - Underflow: result is too negative for finite computer word
  - And, it's NOT just the presence of a carry or borrow out of the top bit!

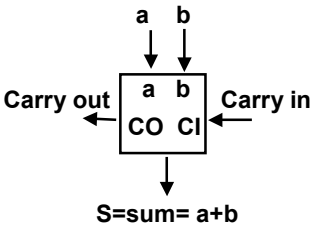
Detecting 2s Complement Overflow

- **Its generically just called "overflow"**
- **When can it not happen?**
  - No overflow when adding a positive and a negative number
  - No overflow when signs are the same for subtraction
- **When can it actually happen?**
  - You overflowed when adding two positives yields a negative
  - or, adding two negatives gives a positive
  - or, subtract a negative from a positive and get a negative
  - or, subtract a positive from a negative and get a positive
- **Consider the operations A + B, and A – B**
  - Can overflow occur if B is 0 ?
  - Can overflow occur if A is 0 ?

Effects of Overflow

- An exception (interrupt) occurs
  - Control jumps to predefined address for exception
  - Interrupted address is saved for possible resumption
  - Details based on software system / language
- Don't always want to detect overflow: unsigned MIPS instructions
  addu, addiu, subu
  - Remember: addiu still sign-extends!
  - Note: sltu, sltiu for unsigned comparisons
- Let's look at implementing addition...

Basics: 1-bit Full Adder Implementation



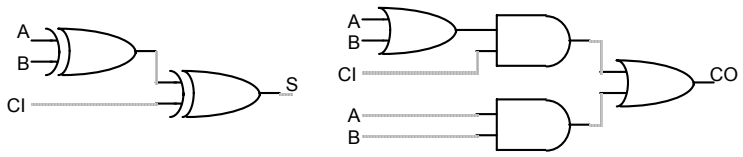
Truth Table

A	B	CI	S
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

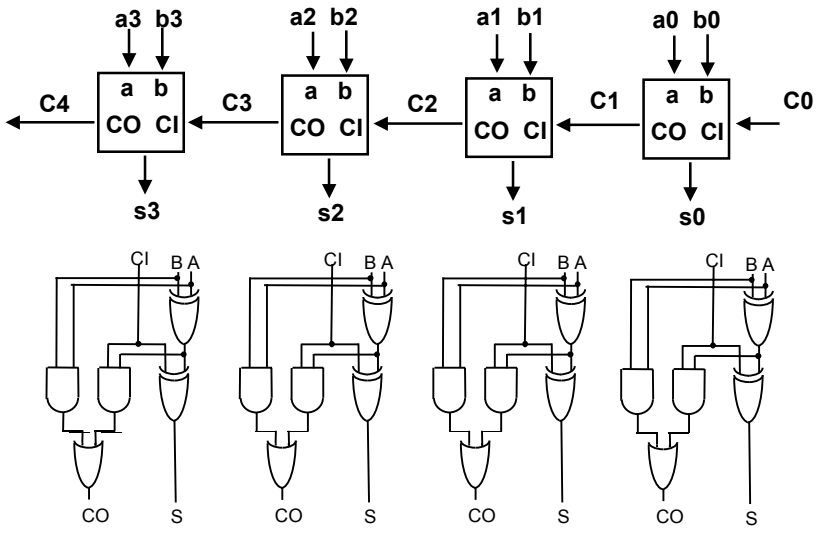
Truth Table

A	B	CI	CO
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Standard Approach: 6 Gates (or 5 Gates)

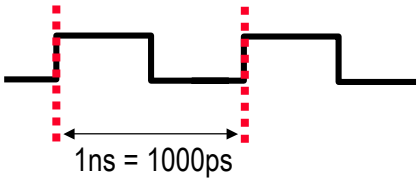


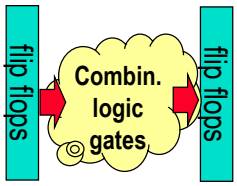
Basics: Ripple-Carry Adder Revisited



What's Wrong with the Ripple Carry Adder?

- It's too slow for wide (32bit, 64 bit) addition.
- How slow…? Consider a fast modern processor
  - Runs at ~ 1GHz, so clock period is ~ 1ns



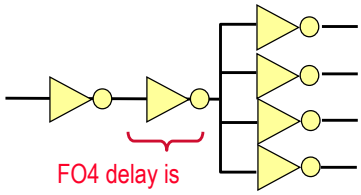


You have roughly 1000ps to get out of the flip flops (FFs), thru the combinational logic, and back into the next FFs.

How many gates deep can this be?

What's Wrong with the Ripple Carry Adder?

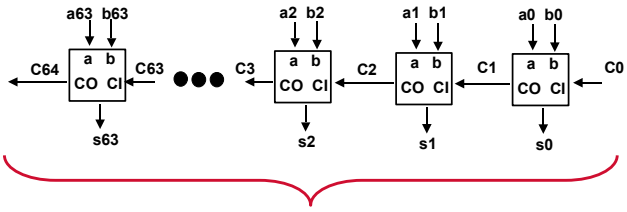
- **Logic depth depends on semiconductor technology**
  - A reasonable, current model of "the delay of 1 typical gate" is called the FO4 delay
  - It's the delay thru one ordinary inverter, driven by an inverter, loaded by 4 inverters
  - Metric is from Mark Horowitz of Stanford, one of the original MIPS guys



- **FO4 delay has been falling off linearly with technology scaling**
  - Pretty good formula for worst case FO4 delay: 0.5 ns/micron * (process feature size)

What's Wrong with the Ripple Carry Adder?

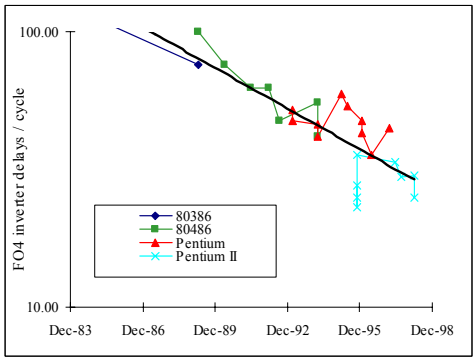
- **Using the FO4 formula**
  - In a process with 0.5micron CMOS features: FO4 = 0.5 * 0.5 = 0.25ns = 250ps
  - In a leading edge 0.15micron process: FO4 = 0.5 * 0.15 = 0.075ns = 75ps
  - At 1GHz, with FO4=75ps/gate, you get 1000ps/75ps = ***13 gate delays in 1 clock tick***



At roughly 2 gate delays per full adder, this ripple Adder is at ~ 64*2 FO4 delays.
Can YOU build a 64 bit adder with only 13 gate delays??

Aside: Levels of Gates Per Clock in uPs

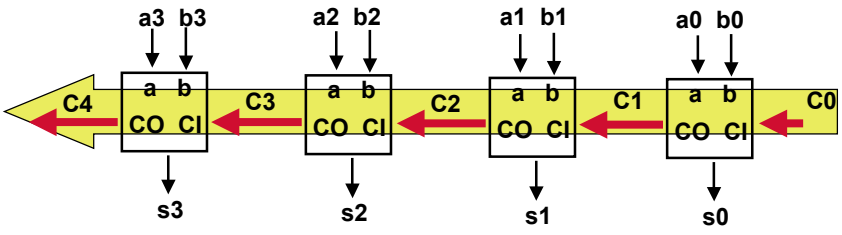
- **Gates/clock, normalized via FO4 delay, have been falling**
  - Clock speeds have just been scaling aggressively, but…there's a limit here
  - It's hard to design a processor with only 16 gate delays per clock tick. Very hard for 8/tick



Data from
Mark Horowitz,
EE Dept
Stanford Univ

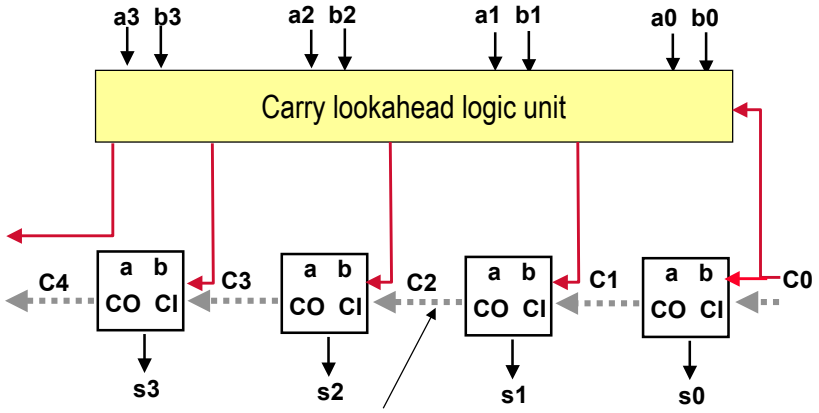
Design Trick: Fast Adders via Lookahead

- **Basic problem**
  - Ripple path for carry is proportional to number of bits in the adder
  - We need to fix this: it needs to be constant, at least for "small" adders
  - The only solution is more hardware in a "small chunk of adder", typically a 4bit adder
  - Luckily enough, there's a nice, elegant, fairly simple pattern to this stuff



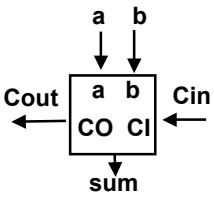
Basic Lookahead Adder

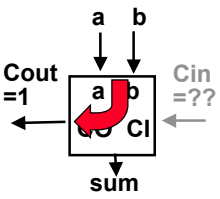
- For 4bit adder, can we compute all intermediate carries *directly*?



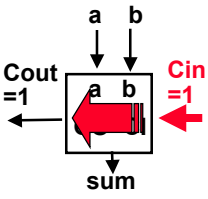
Basic Lookahead Adder

- Turns out there's a nice pattern to the logic in this lookahead box
  - Think about a single full adder, and how carries "happen" in it
  - Turns out, there's exactly 2 ways a carryout "happens", ie, can get set to be "1"





Question: when will a carryout be **generated independent** of value of the carryin bit?
Answer: when a=1 && b=1



Question: when will a carryout be **propagated** from carryin, **thru** the adder?

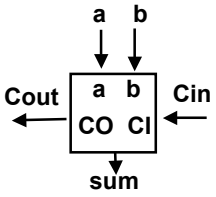
Answer: when a !=b

Basic Lookahead Adder

- Give these 2 unique "carry happens" events names
  - When a,b are set so that a carryout is just generated: g = *generate* = a*b
  - When a,b are set so that a carryin passes to be carrout: p = *propagate* = a ⊕ b
- Write equation for carryout for a single adder in this notation

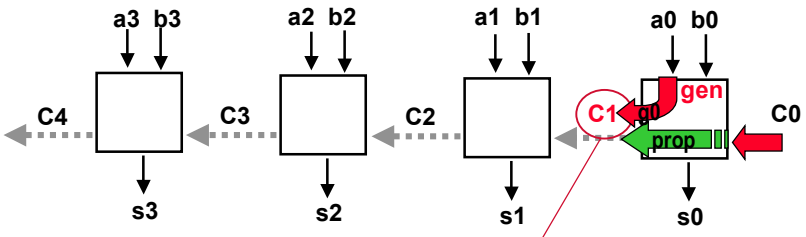
Carryout = "either I generated it, or, I propagated the carryin to carryout"

= g + p*Cin
= (ab) + (a ⊕ b)*Cin



Baic Lookahead Adder

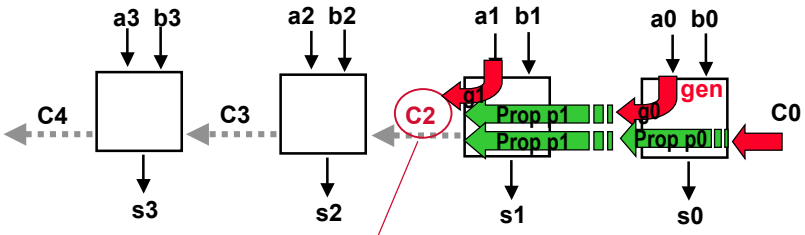
- With this notation, can see "pattern" for each intermediate carry
  - Look at the 4bit adder up close, let's write a direct equation for EACH carry we need



C1 = g0 + poC0
ie, either stage0 generated it
or, C0 propagated thru stage 0

Baic Lookahead Adder

- **Keep going, use the pattern**
  - Look at the 4bit adder up close, let's write a direct equation for EACH carry we need

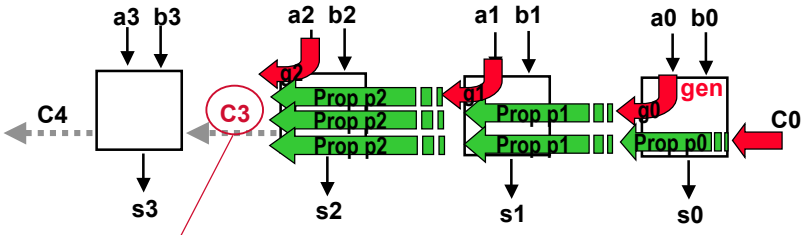


C2 = g1 + p1g0 + p1p0C0

ie, either stage1 generated it
or, stage1 propagated a carry generated in stage0
or, stage1 and stage3 propagated the Cin

Baic Lookahead Adder

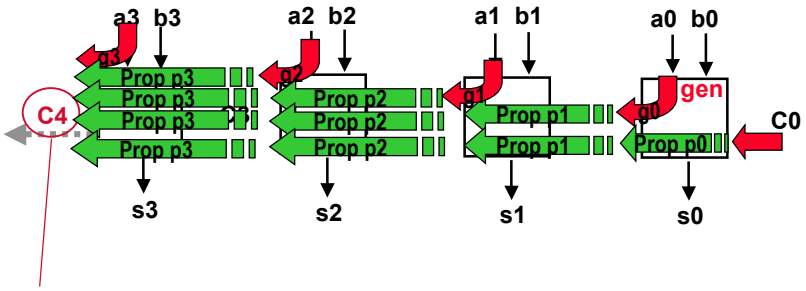
- **Keep going, use the pattern**
  - Look at the 4bit adder up close, let's write a direct equation for EACH carry we need



C3 = g2 + p2g1 + p2p1g0 + p2p1p0C0

Baic Lookahead Adder

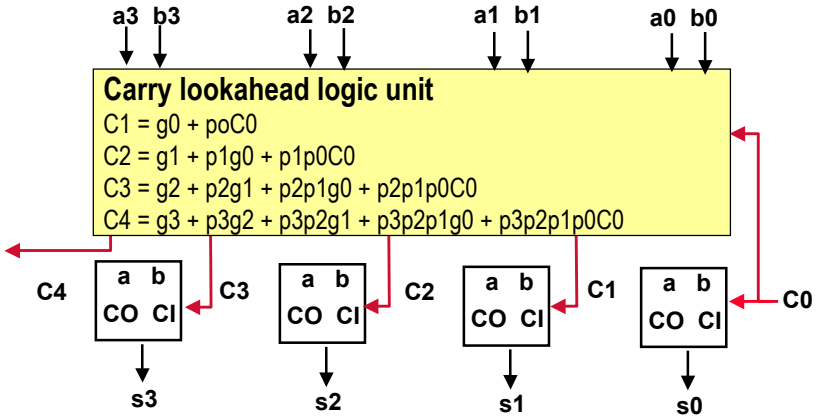
- **Keep going, use the pattern**
  - Look at the 4bit adder up close, let's write a direct equation for EACH carry we need



$$C4 = g3 + p3g2 + p3p2g1 + p3p2p1g0 + p3p2p1p0C0$$

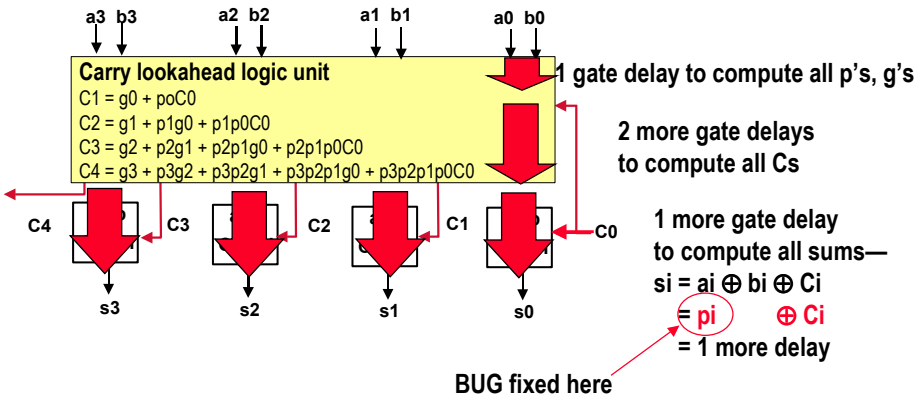
Basic Lookahead Adder

- **So—YES, we can do all the carries directly, no ripples at all**
  - Why is this fast? Each carry equation is a SOP 2-level form, 2 FO4 delays to compute



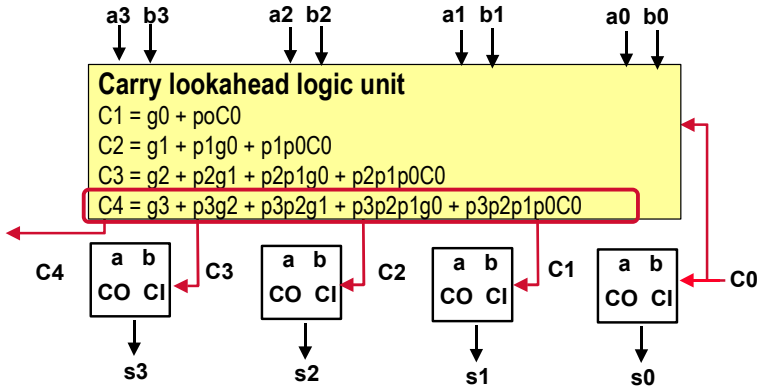
Basic Lookahead Adder

- How fast is it? ~4 gate delays thru the whole 4bit adder



Beyond Basic Lookahead

- Neat digital trick. What keeps us for doing this for 64bits?
  - The lookahead equations for the individual intermediate carries get too complex
  - Carry Cn has (n+1) terms ORed, and the biggest AND has n terms in it.

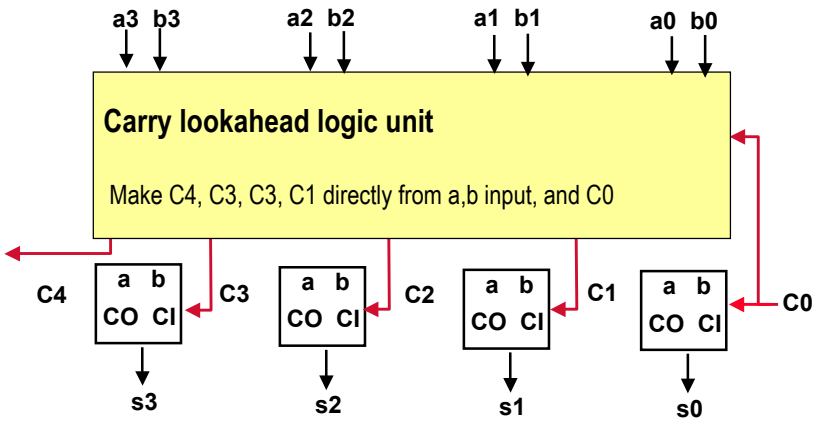


Beyond Basic Lookahead: Recursive Lookahead

- **Another wonderful, elegant trick that gives a useful pattern**
  - The exact same set of formulas works to apply these ideas recursively
  - The question is: what are we recursing on? And, in hardware?
- **Big trick: the lookahead equations for the carries do not care how big the individual adders were that gave us the g, p signals**
  - We derived these for the "generate from" and "propagate across" 1-bit adders
  - You can do the same think for N-bit adders. In our case, 4-bit adders
  - Now, the g, p signals are commonly written **G, P**, called **"group" generate, propagate**
  - Your book calls them **"super" generate and propagate**

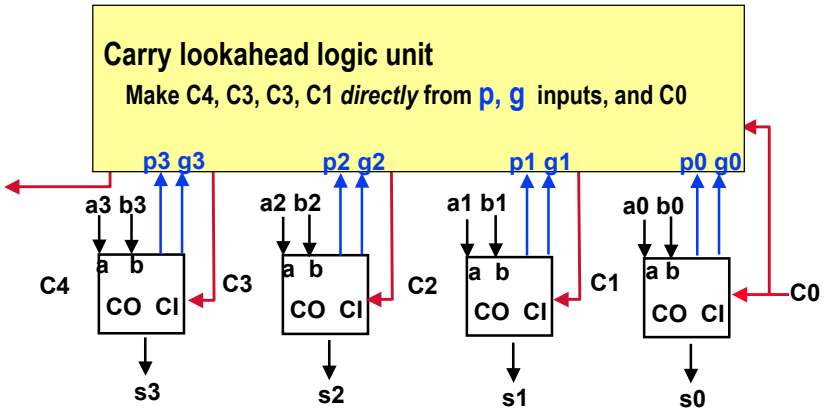
Recursive, Group Lookahead

- **We derived this lookahead structure**



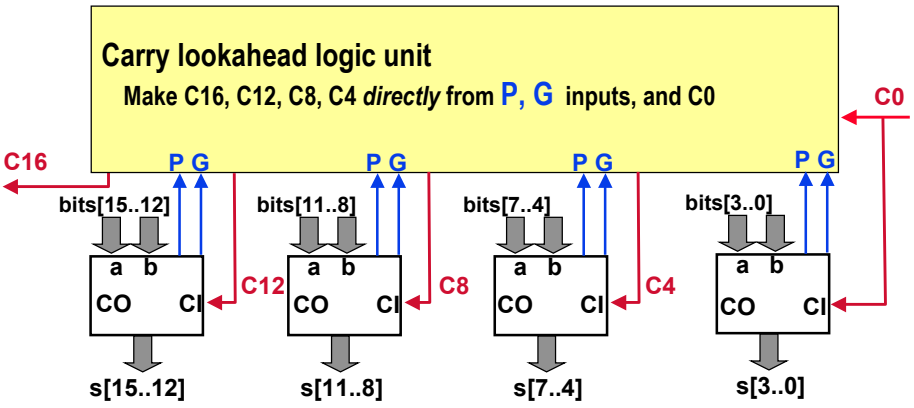
Recursive Group Lookahead

- Lets redraw it to separate out the p's, g's, and the carry logic

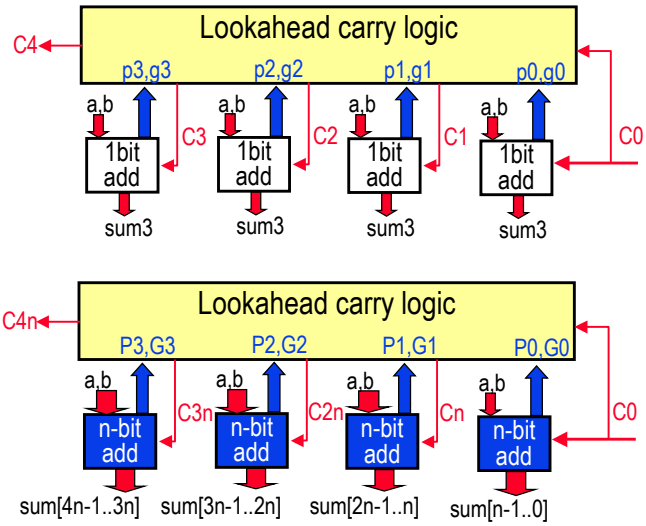


Recursive Group Lookahead

- Big idea: as long as the p's, g's are correct, same lookahead unit will work for wider adders at the bottom



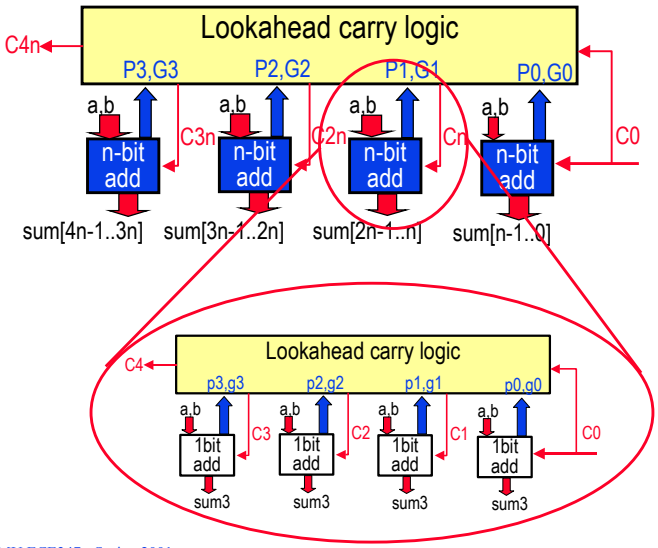
Recursive Group Lookahead



Lookahead logic for 1bit adders…

Is identical for wider, n-bit adders at the bottom!

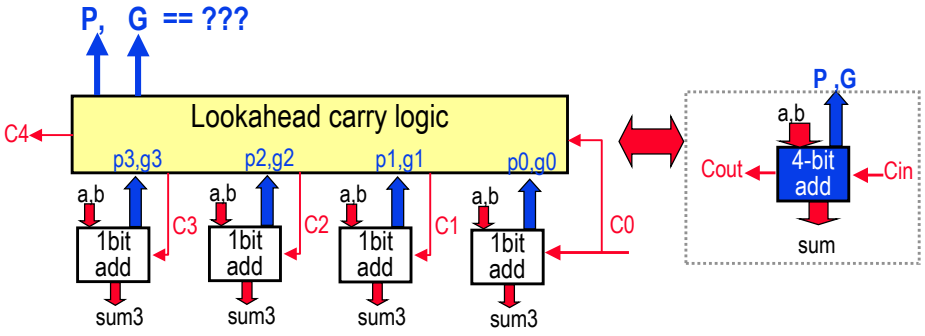
Why We Think of it as *Recursive*



If n=4 here, then each wider adder could be a lookahead 4-bit adder, as shown here

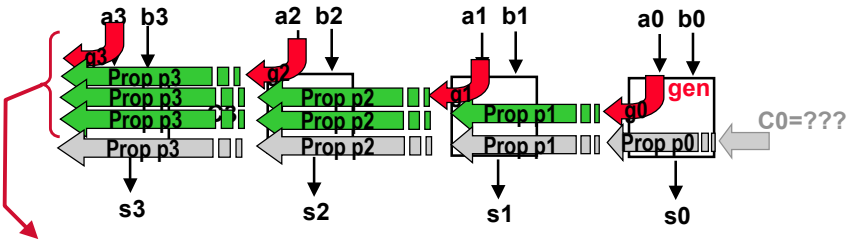
What's Missing Here?

- **We need to know how to generate the group-level signals P, G**
  - With these, we can use this fast 4 bit adder as a component in a wider, lookahead adder



Group Level Signals

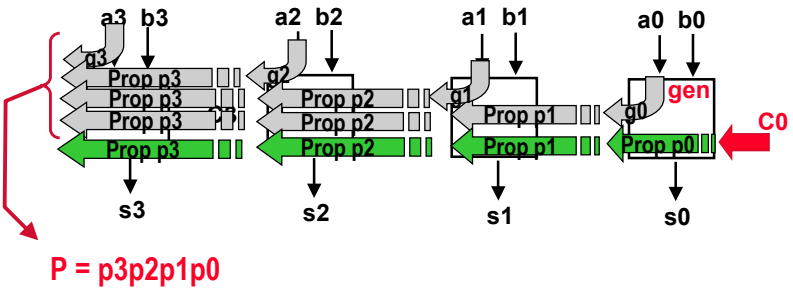
- **Actually, pattern still works fine. Consider group gen = G**
  - Group generate G = when does the whole 4-bit block generate a carry without us needing to know value of C0?



G = g3 + p3g2 + p3p2g1 + p3p2p1g0 } **BUG fixed here**

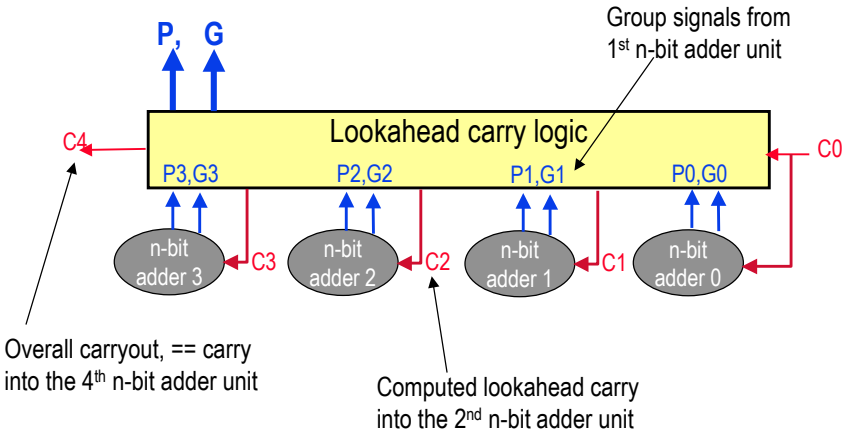
Group Level Signals

- **Consider group propagate P**
  - Group prop P = when does the whole 4-bit block propagate a carry across all 4 bits right back from the value of C0?



Group Level Lookahead

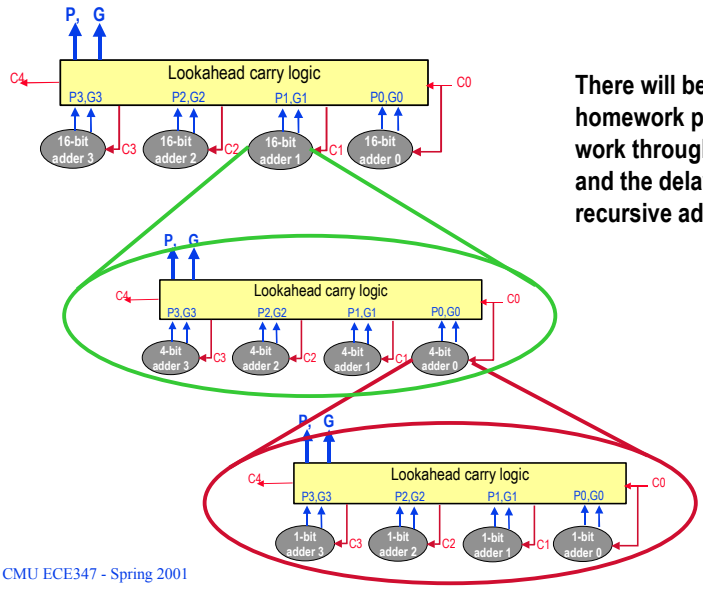
- **And, that's it. A *generic lookahead* carry logic unit that "looks across" 4 adders looks like this:**



Group Lookahead

- **Easiest to see how to do 2 levels of lookahead**
- **For example: 16bit adder**
  - Make fast 4 bit adder as we now know how: use 1st layer of lookahead logic
  - Then, make the group generate, propogate P,G signals for each 4 bit adder
  - Use another layer of lookahead – exact same lookahead logic !! – to combine 4 of these fast 4-bit adders, and do lookahead across each 4-bit adder, to get to 16 bits
- **Don't have to stop at 2 levels of lookahead**
  - To get to 64 bit adder, take this fast 16-bit adder, and combine 4 of them with a lookahead unit – exact same lookahead logic again !! – to get to 4*16=64 bits
- **Variants of these ideas are how wide, fast adders get built**

64 Bit Adder: How Fast, in Gate Delays?



There will be some homework problems to work through the details, and the delay, on these recursive adder structures.

New Problem: Design a "Fast" ALU for MIPS

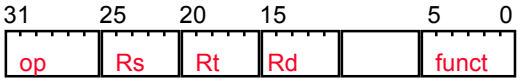
- **Requirements?**
  - Its not just adding (and subtracting)
  - It also must support the Logic operations – whole-word bit ops like AND, OR
- **How?**
  - Think about what we can do with each individual bit of this computation (like 1 bit of a ripple adder is simple to do)
  - Think about how to generalize from the single bit up to the whole ALU…

MIPS ALU Requirements

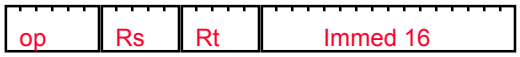
- **Add, AddU, Sub, SubU, Addl, AddlU**
  - => 2's complement adder/subtractor with overflow detection
- **And, Or, Andl, Orl, Xor, Xori, Nor**
  - => Logical AND, logical OR, XOR, nor
- **SLTI, SLTIU (set less than)**
  - => 2's complement adder with inverter, check sign bit of result

MIPS Arithmetic Instruction Format

- R-type:



I-Type:



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

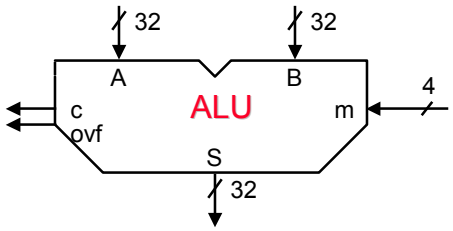
Design Trick: Divide & Conquer

- **Break the problem into simpler pieces, solve each, glue together**
- **Example:**
  - Assume the immediates have been taken care of before the ALU
  - 10 operations (4 bits)

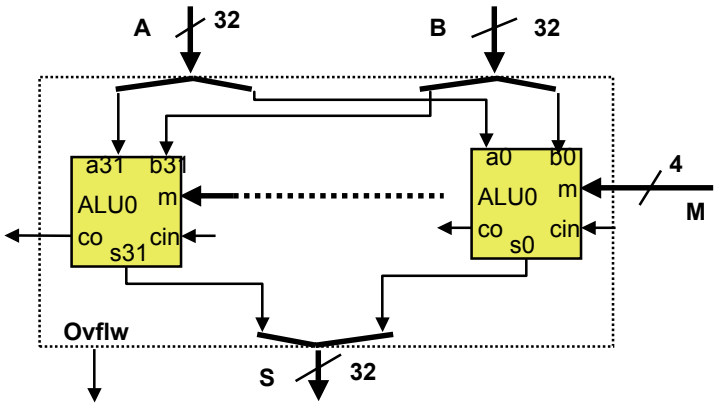
00	add
01	addU
02	sub
03	subU
04	and
05	or
06	xor
07	nor
12	slt
13	sltU

Refined Requirements

- **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
- **Block Diagram**

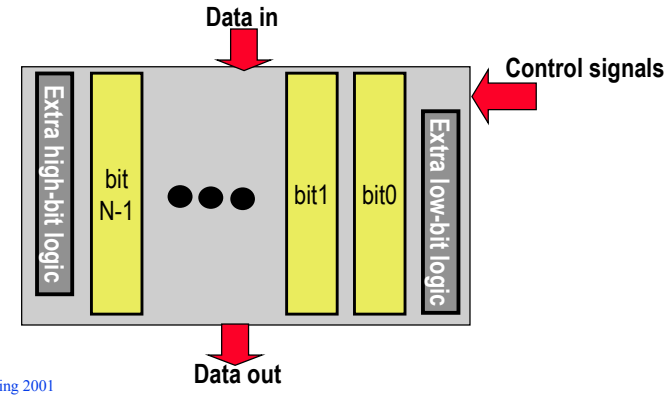


Refined Diagram: *Bit-slice ALU*



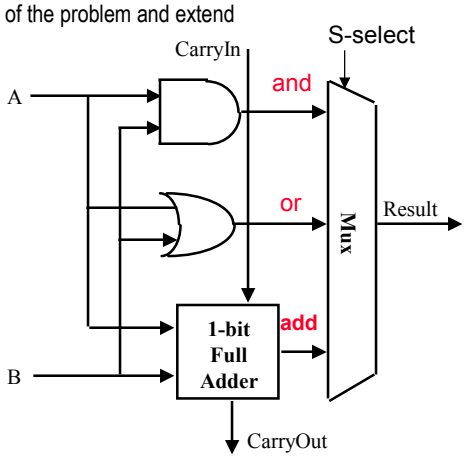
Another Way to Think About It

- **We want an N-bit ALU. Design 1-bit "slices" of this ALU.**
  - Then, try to glue them togther like a ripple carry adder
  - Remember—ripple adder makes a big adder by letting the carryin-carryout connects glue all the 1-bit pieces together



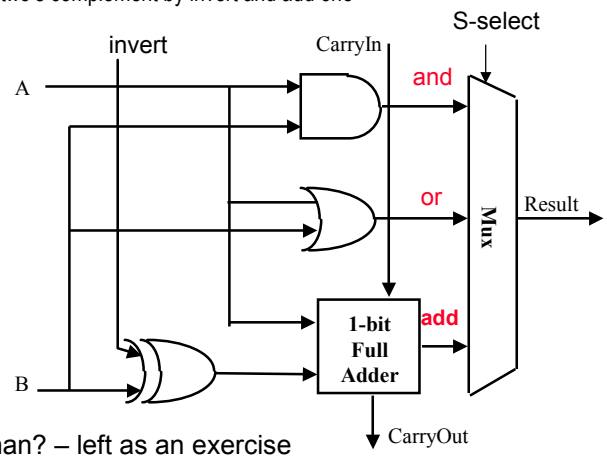
One Bit of the Bit-Slice Design

- **Design trick:**
  - Take pieces you know (or can imagine) and try to put them together
  - Solve part of the problem and extend



Additional Operations

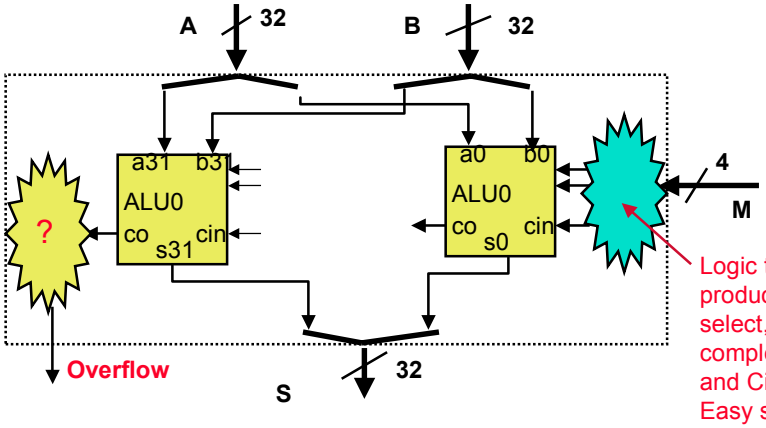
- A - B = A + (– B)
  - Form two's complement by invert and add one



Set-less-than? – left as an exercise

Revised Diagram

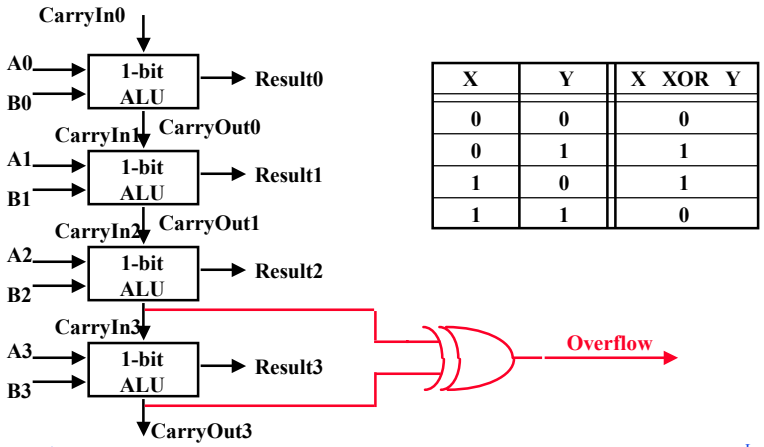
- LSB and MSB: we need to do a little extra work on these



Logic to produce select, complement, and Cin; Easy stuff

Overflow Detection Logic

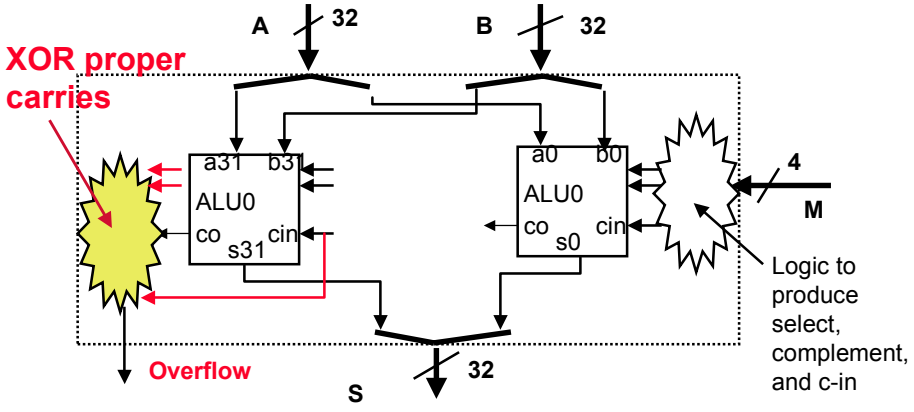
- Carry into MSB xor Carry out of MSB
  - For a N-bit ALU: Overflow = CarryIn[N - 1] XOR CarryOut[N - 1]



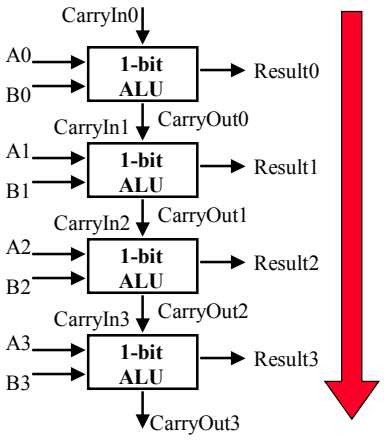
X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

Updated Diagram

- LSB and MSB need to do a little extra



But What About Performance?



- Critical Path of n-bit ripple adder way too slow…
- *Perfect* place to use the fast lookahead ideas
- Just adds some more "extra logic" around bits in the bitslice to do the recursive lookahead

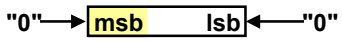
Additional MIPS ALU Requirements

- **Mult, MultU, Div, DivU**
  - Need 32-bit multiply and divide, signed and unsigned
  - Next lecture…
- **Sll, Srl, Sra**
  - Need left shift, right shift, right shift arithmetic by 0 to 31 bits
- **Nor**
  - Logical NOR or use 2 steps: (A OR B) XOR 1111....1111

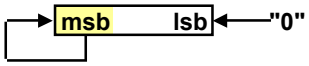
Combinational Shifters

- 2 types: issue is what bit value gets "shifted in" on the ends?
  - 0 is obvious first answer, but its not always 0 that gets shifted in…

logical-- value shifted in is always "0"



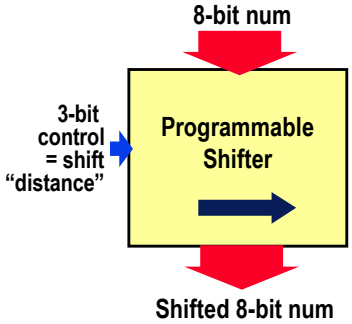
arithmetic-- on right shifts, sign extend (ie, copy msb back in)

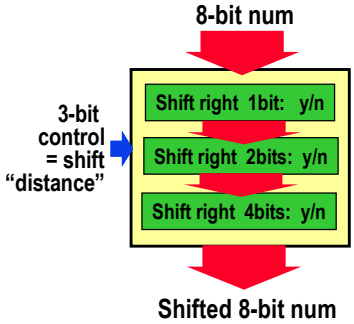


- Note:
  - These are **single bit shifts.**
  - A given instruction might request **0 to 32 bits to be shifted!**

New Problem: Big, Fast Shifters

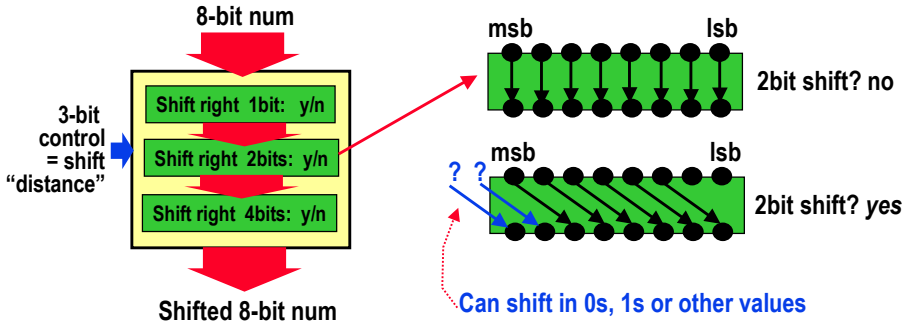
- Take an n-bit word, left or right shift k-bits, *programmably. How?*
  - Answer: a *logarithmic* shifter structure, done as layers of shifters
  - Each layer of the shifter structure can shift 2^M bits in one direction.
  - Each layer is programmable – either it shifts or not.
  - If your word is 2^N bits in all, you need N layers of shifters, hence the "log" idea



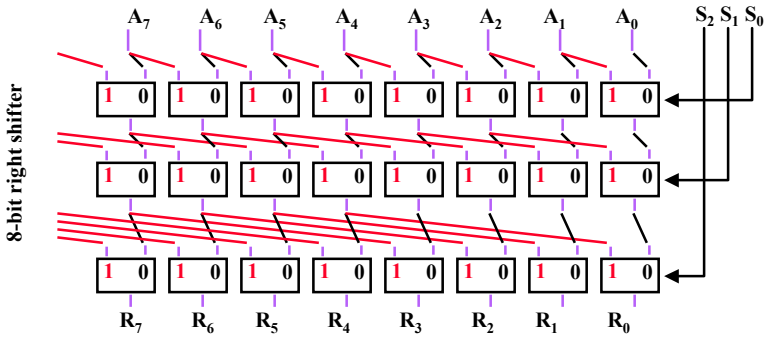


Big, Fast Shifters

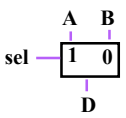
- How do you make any one of these layers of the shifter?
  - Out of multiplexors. Its pretty simple –mainly just MUXs and wires



Details: Big, Fast Shifter From MUXes

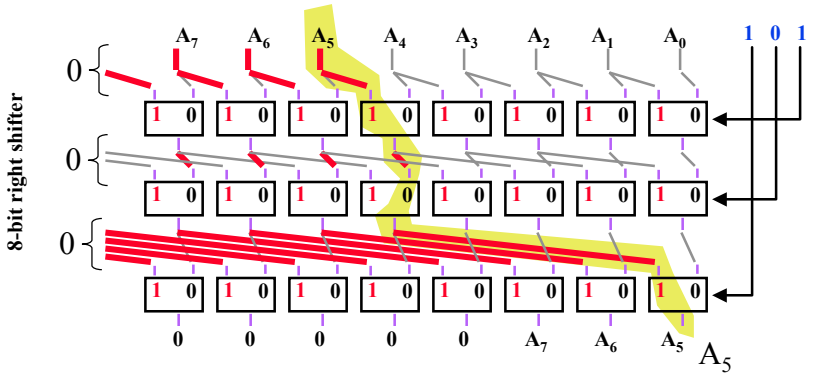


Basic MUX Building Block

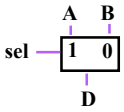


- What comes in the MSBs?
- How many levels for a bigger shifter?
  - 32 bit shifter? 64bit shifter?

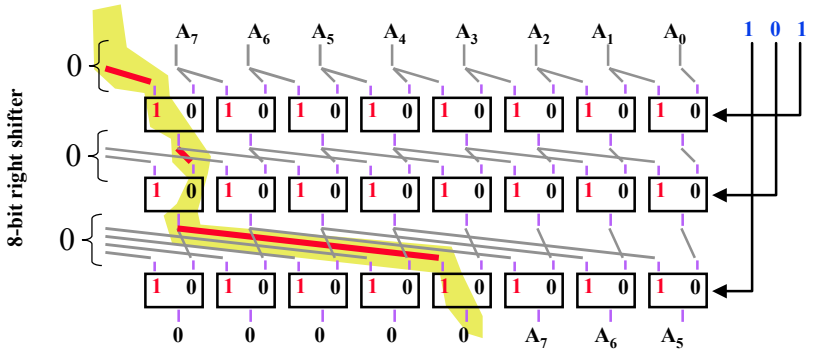
Combinational Shifter: Basic Operation



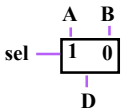
Basic MUX Building Block



Combinational Shifter: Basic Operation



Basic MUX Building Block



- **What comes in the MSBs?**
  - 0s here, shifted in from the left
  - Could be 1s, could be the topmost msb if we wanted

Summary

- **Adders**
  - Always get built using carry lookahead ideas
- **ALUs**
  - Always get built as regular bit-slices, repeating a basic unit bit design
  - Some extra stuff usually requires for lowest and highest bits, and for lookahead
- **Shifters**
  - For a single, fixed shift distance, can just hardwire up the MUXes
  - For arbitrary programmable shift distances: barrel shifter, with layers of MUXes