CS151B Winter 2022

Discussion (Week 2)

CS151B Teaching Team

Agenda

- Logistics
- Review of this week's material
    - Architectural tradeoffs
    - ALU
    - Adders (ripple carry, carry look ahead)
- Practice problems
- Q&A

Logistics

- Homework 2 is due tonight 11:59 PM PST
- No class on next Monday in observance of MLK day

* apology for my voice

Clarification week 1

What insn type is:

J,jal,jr,jral?

bne,beq?

I type:

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

R type:

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

J type:

6-bit Opcode	26-bit Immediate

changing ip (instruction pointer) x4

Suppose bne type had 2 bit address space. How many address can it jump to?

What is the range of address it can span?

Clarification week 1

What insn type is:

J,jal,jr,jral?

bne,beq?

I type:

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

R type:

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

J type:

6-bit Opcode	26-bit Immediate

J, jal = J type / Jr jral = R type / bne,beq = I type

changing ip (instruction pointer) x4

Suppose bne type had 2 bit address space. How many address can it jump to? Total of 4 address:

What is the range of address it can span? 16 bytes

Review

Review

Know, recall, question

Architectural Tradeoff Analysis

- In general, not an easy problem… a small change can have a chain of effects
- We will only focus on performance (execution time) for now

Some tips to reason about such problems

- How is each term effected? (**ET** = **IC** x **CPI** x **CT**)
- How does the **ISA** need to be modified? What are the implications?
- Pressure on **memory** and caching? (e.g., **extra space** needed?)
- Pressure on registers? (e.g., more registers can mean less **register spilling**)
- Constraints from physical hardware? (e.g., **longer wire** can bring higher latency)
- …

Trade-offs of increasing register file size

If we add more registers (32 -> 64), what are the *possible* *positive* effects?

A: Compilers can use more registers => less **register pressure** => might be less spilling => fewer (expensive) lw and sw => lower IC, lower CPI

Trade-offs of increasing register file size (cont.)

- **Physical design constraints:** *higher latency to access register values*
    - Latency might increase - but if I pipelined my CPU into multiple stages, and the longest latency was not affected, latency may not be increased in this case.
    - But if the reg file is the critical path, then CT could increase
        - If I want to mitigate this by splitting up the bottleneck stage, RF in this case, CPI will go up

Trade-offs of increasing register file size (cont.)

- **Bit Constraints**: More bits needed to index registers
    - If we want to keep the instructions to be 32 bits long, we need to sacrifice bits from other fields in the instruction encoding
    - Effects on instruction set and **instruction encoding**
        - **Fewer bits for opcode** => less instructions available => might need multiple instructions to do what could be done with one instruction before => higher IC
        - **Fewer bits for immediate field**
            - Might need more instructions, e.g., addi, bne => higher IC
            - Might need to put more values into registers => **more pressure on register space** => more spilling => higher IC, higher CPI
        - **Fewer bits for shift-amount field** => one shift instruction can no longer shift the full 32-bit range => higher IC
        - **Fewer bits for funct field** => higher IC

Trade-offs of increasing register file size (cont.)

- If we decide to use more than 32 bits to encode an instruction, the instructions would take more memory space to store
- More **pressure on memory and caching** => takes longer to retrieve instructions => higher CPI, possibly higher CT

ALU

ALU

What is it?

ALU

What is it?

ALU (Arithmetic Logic Unit)

Part of the CPU which handles arithmetic and logical operations.

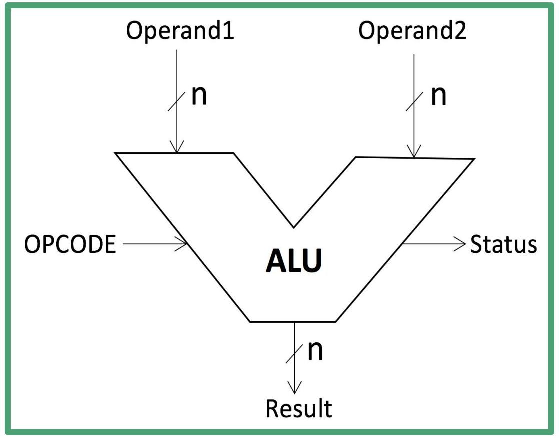
Inputs

- 2 operands -> 32 bits each
- operation bits -> 4 bits in total
  - 2 operations bits
  - A-invert, B-negate each 1 bit

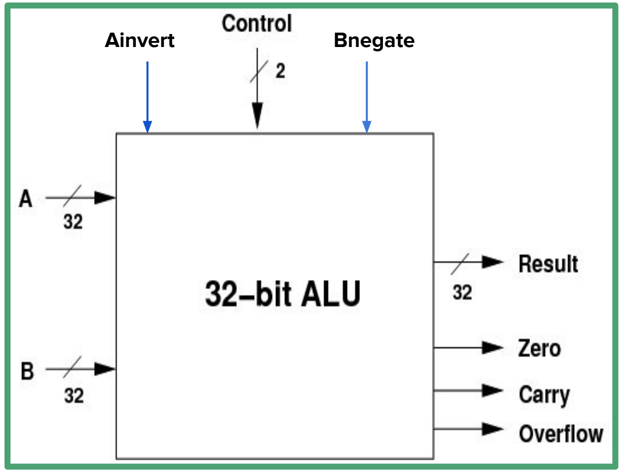
Outputs

- zero -> indicate if the last calculation produced a 0 result
- result -> 32-bit result of the operation performed
- overflow -> single bit to indicate if an overflow has occurred
- carryout -> carryout bit from the last unit, we don't really use it for any operations

Generalized View



MIPS 32-bit View



Img source: http://www.csc.villanova.edu/~mdamian/Past/csc2400fa13/assign/ALU.html

ALU (Arithmetic Logic Unit)

Part of the CPU which handles arithmetic and logical operations.

Inputs

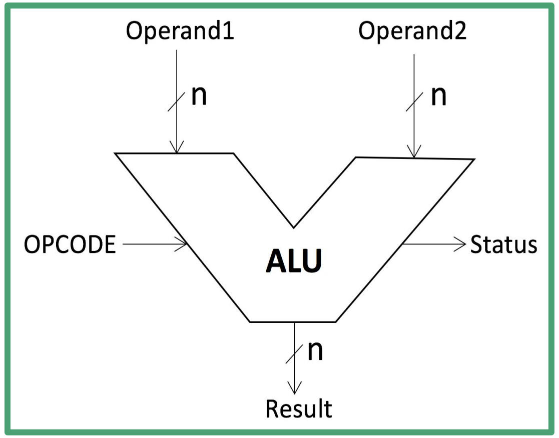
- 2 operands -> 32 bits each
- operation bits -> 4 bits in total
    - 2 operations bits
    - A-invert, B-negate each 1 bit

most integer op
Add, subtract,
SLT, equal,not equal
Or and nor etc...

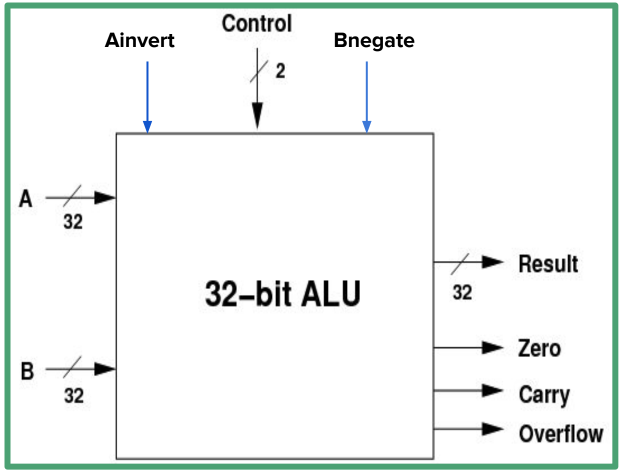
Outputs

- zero -> indicate if the last calculation produced a 0 result
- result -> 32-bit result of the operation performed
- overflow -> single bit to indicate if an overflow has occurred
- carryout -> carryout bit from the last unit, we don't really use it for any operations

Generalized View



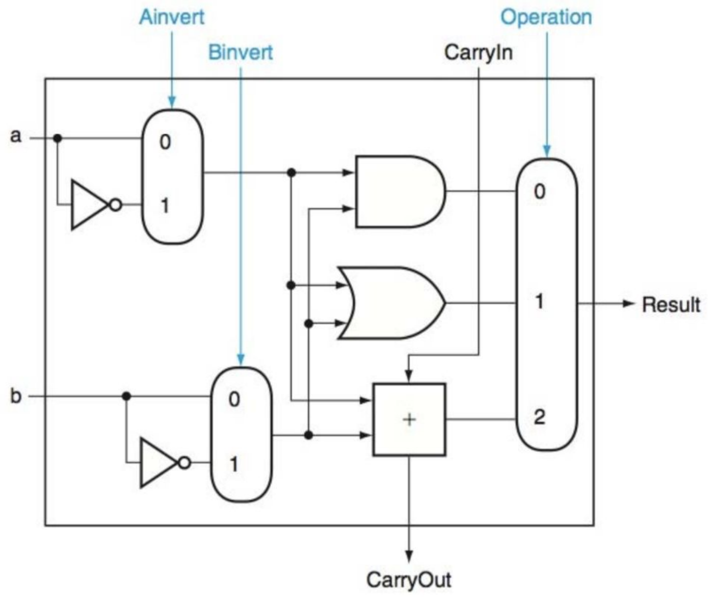
MIPS 32-bit View



Img source: http://www.csc.villanova.edu/~mdamian/Past/csc2400fa13/assign/ALU.html

ALU (detail)

A simple 1-bit ALU



Note: Binvert and CarryIn are from the same bit -> Bnegate for the least significant bit ALU

Q1: Implementations of ADD, OR, AND are obvious, but how do we implement NOR?

A: We already have Ainvert, Binvert bits, and an AND gate, we just have to set those two bits and select the result for an AND

Note: a **NOR** b ⇔ ~(a **OR** b) ⇔ ~a **AND** ~b (DeMorgan's law)

But how do we do subtraction in binary? (2's complement)

- **Subtracting a number is the same as adding the inverse of it**
  - Example: `7 - 4 = 3  ⇔ 7 + (-4) = 3`
- Same logic can be applied to binary numbers
  - Example: `0111 - 0100 =  0011` ⇔ `0111 + (-0100) = 0011`
  - represent `(-0100)` in **2's complement** notation ?
    - **Flipping all the bits and adding 1 gives the inverse of a number in 2's complement**
    - i.e) `(-0100) => 1011+0001=1100` **`(-4 in 2's complement form)`**
  - Thus, `0111 - 0100(unsigned)` ⇔ `0111 + 1011 + 0001` ⇔ `0111 + 1100(2's complement)`

  - **... we can use any adder to implement subtraction.**

Bottom Line: A - B = A + ~B + 1

Implementing Subtraction on the ALU

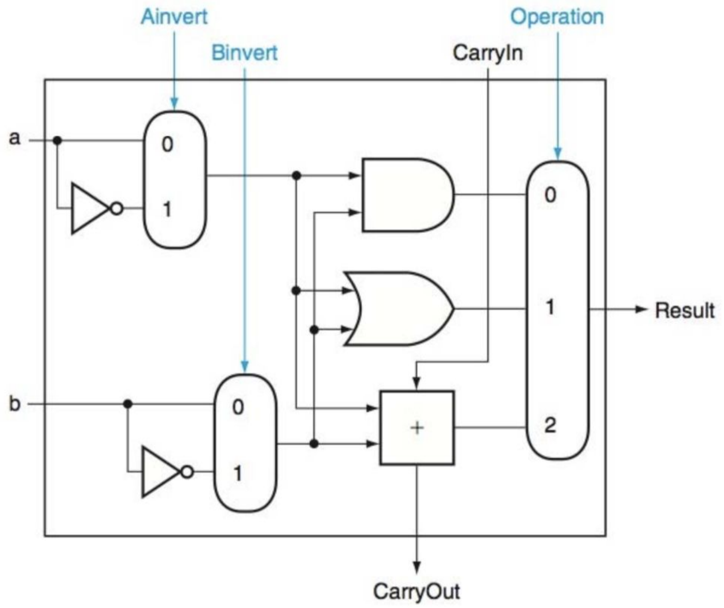
To implement a-b, we need to negate b, then we can just add a and -b using the ALU's adder.

First we invert b, how can we do that?

A: We set Binvert and send it to every ALU cell. Each cell will pass the inverted version of b's bit to the adder, rather than the original version.

Next we need to add 1 to b to negate it. How do we implement that?

A: We set the CarryIn signal from the least significant bit.



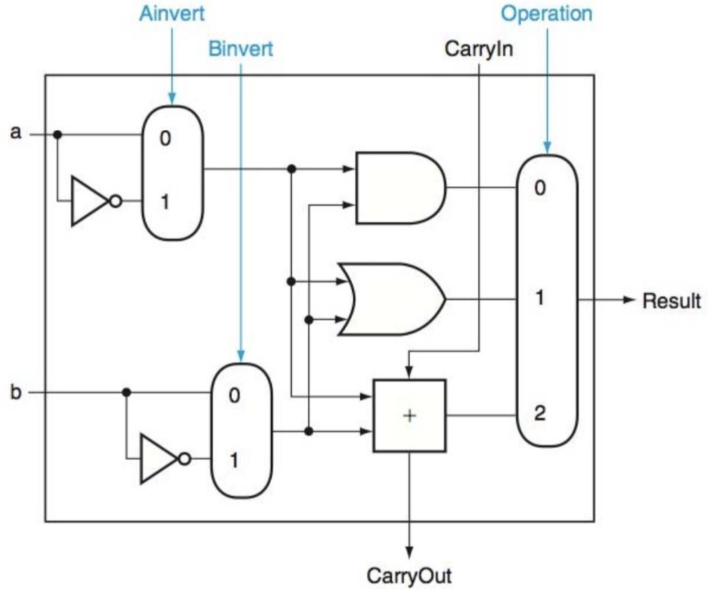
1-bit ALU Cell

Q: For the least significant bit, is there a case where the Binvert and CarryIn MUST be different for the ALU to implement an operation?

No. That's why we use Bnegate

ALU (simplified 1-bit ALU)

A simple 1-bit ALU



So far we can support ADD/Subtract, OR, AND, NOR operations with the above design. With these operations, we can execute most MIPS instructions, however, we're not done yet, we still need to implement SLT.

SLT (set on less than)

```
slt $t1,$t2,$t3
```

This will set `$t1` to 1 if `$t2<$t3`, to 0 otherwise.

`Can you implement this using addition/subtraction?`

```
•   $t2-$t3
      ○   if ($t2-$t3) <0 , that means $t2<$t3, so $t1=1
      ○   else , we know $t2 >= $t3, so $t1=0
```

However, we are working with 32-bit integers, so we need to output 32-bit results.

More specifically, we need to output 1 and 0 in the following format:

0000 0000 0000 0000 0000 0000 0000 0001 `(if $t2<$t3)`

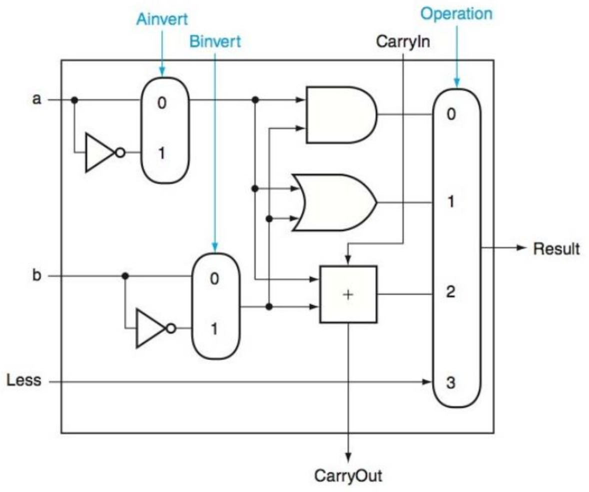
0000 0000 0000 0000 0000 0000 0000 0000 `(if $t2 >= $t3)`

But how do we get those bit patterns, what decides the least significant bits based on the comparison result ?

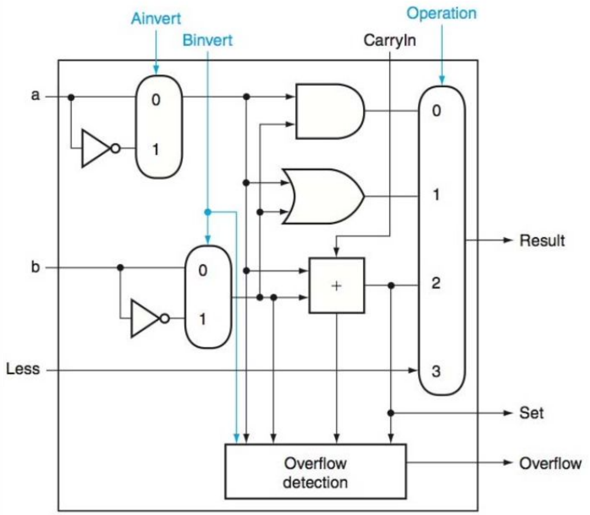
SLT Implementation

- We can hardwire all the bits other than the least significant bit, as **they're always zeros.**
- The least significant bit comes from the subtraction result of the 1-bit ALU for the **most significant bit**. Why?

A: Because the MSB of a negative number is 1 in 2's complement form. If the subtraction result is negative then the MSB is 1.



All but MSB



Most Significant Bit

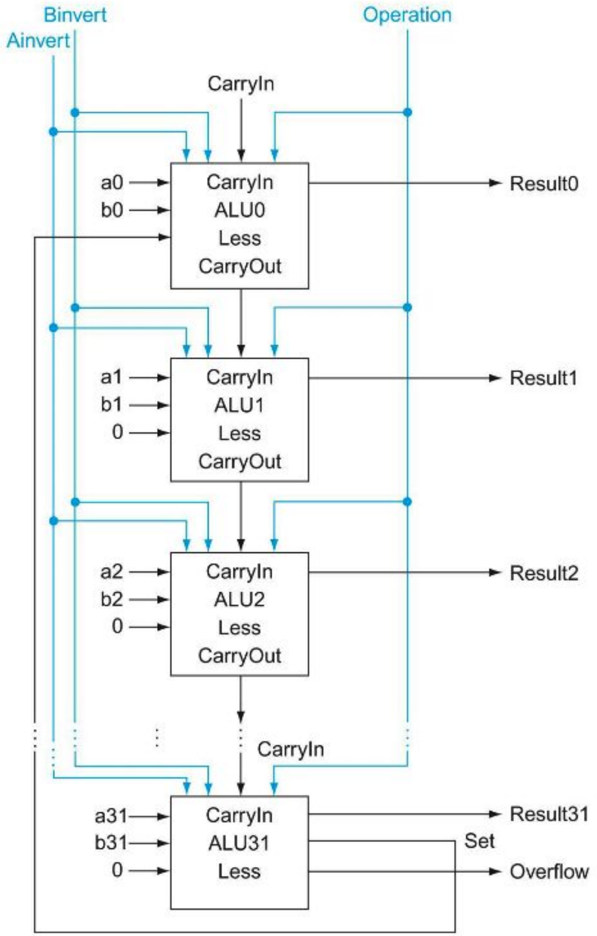
All we need to do is to connect the result of the MSB ALU to the result of the LSB ALU

ALU (full view-almost done)

All we need to do is to connect the result of the MSB ALU to the result of the LSB ALU. As shown in the picture here.

So far so good, but how do we check if we need to branch or not for **bne/beq** instructions?

MIPS word is 32 bits wide, and we need a 32-bit-wide ALU. We achieve this by connecting 32 1-bit ALUs. More on this in the "Adders" section.



ALU (full view-complete)

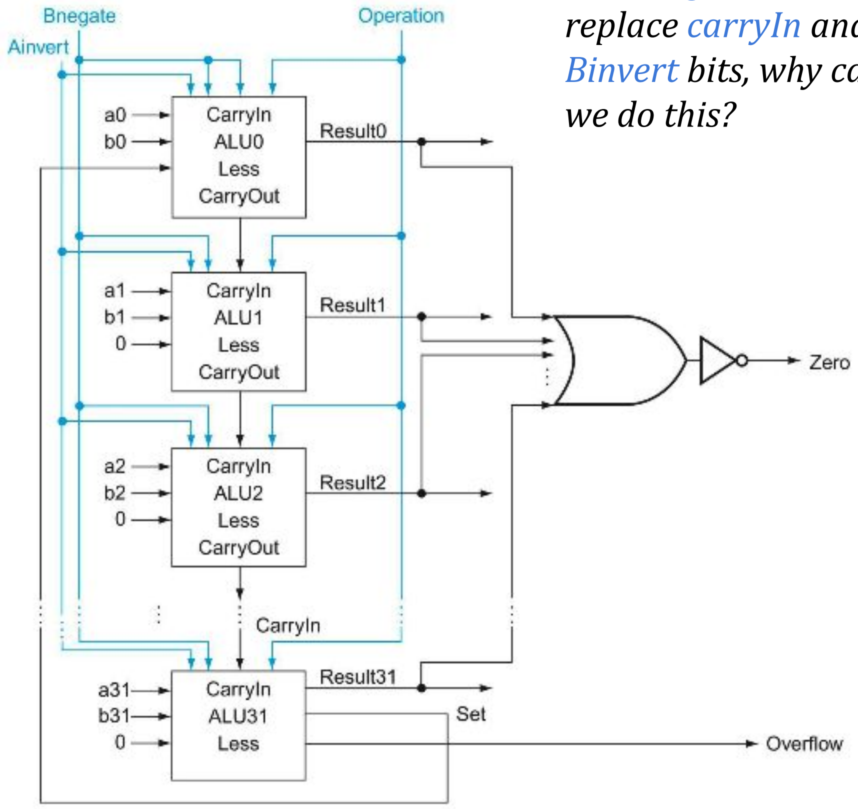
Simplification

Use Bnegate to replace carryIn and Binvert bits, why can we do this?

In order to support **bne/beq** instructions, we can still use subtraction, then check if the result is zero or not.

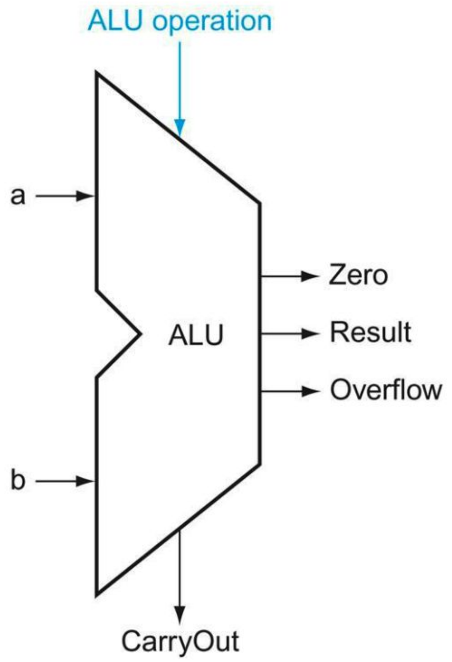
ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

1-bit Ainvert,1-bit Bnegate, 2-bit Op



Complete 32-bit ALU (detailed view)

ALU (complete-abstract view)



Complete 32-bit ALU (abstract view)

Adders (Latency)

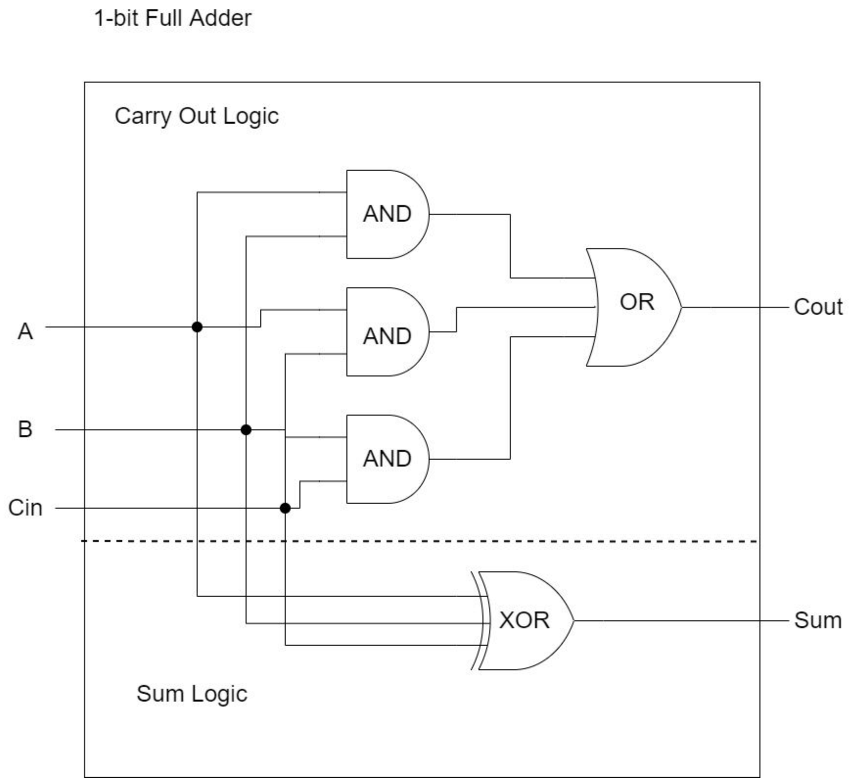
We have built an ALU, let’s take a closer look at the adders

Closer look at a 1-bit full adder

- Given `a, b, carry_in`
- How to implement `sum(s)`?
  - 3-input XOR or 2 XOR gates
- How to implement carry out?
  - Depends on what type of adder we are using
    - Ripple carry: ab + bc_in + ac_in
    - Carry Look Ahead: e.g. $C_1 = G_0 + C_0 P_0$

      (Note: the multiplication here means the logical AND, the addition here means the logical OR)

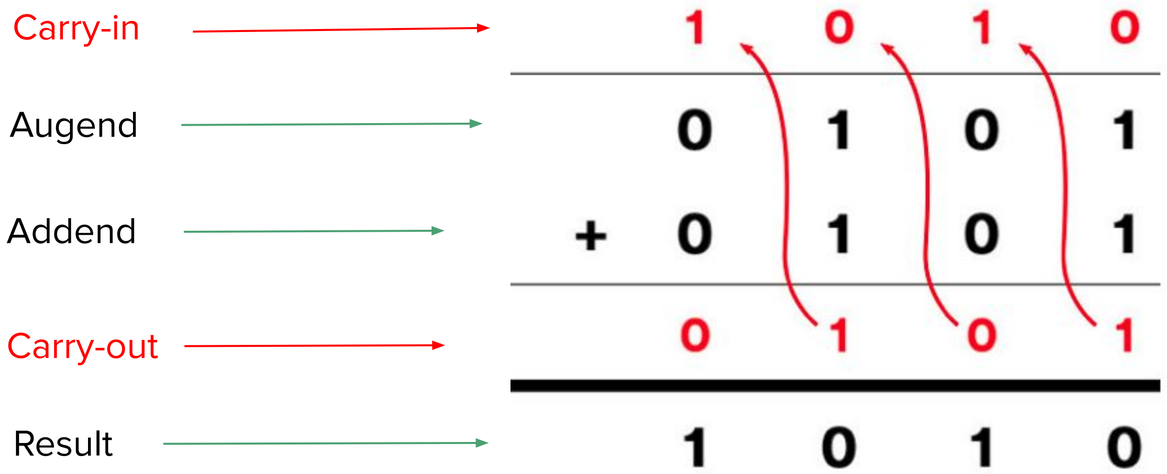
A closer look at a 1-bit adder(not important)



But how do we calculate multiple-bit additions? I.e, how do we put multiple such 1-bit adders together ?

First, how do we do addition in binary?

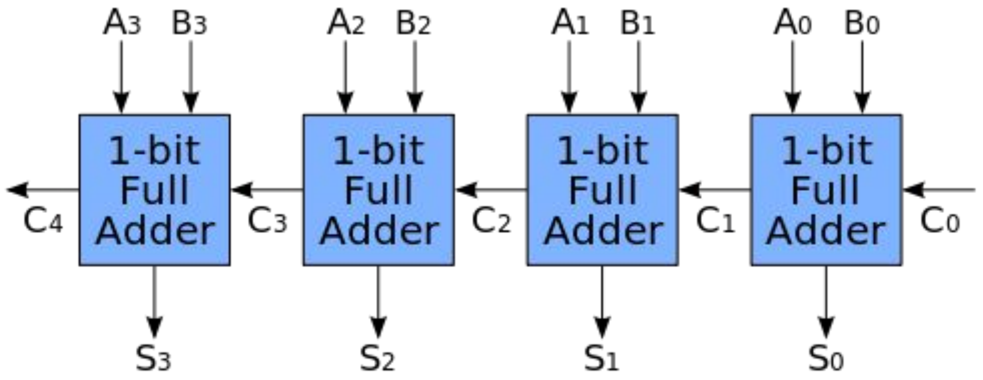
Let's try `0101 + 0101`



Implementation 1: (Naive ripple carry adder)

We just implement the paper and pencil calculation of addition in hardware.

Easy to design and understand, simple logic



A_3 B_3 A_2 B_2 A_1 B_1 A_0 B_0
1-bit Full Adder
C_4 C_3 C_2 C_1 C_0
S_3 S_2 S_1 S_0

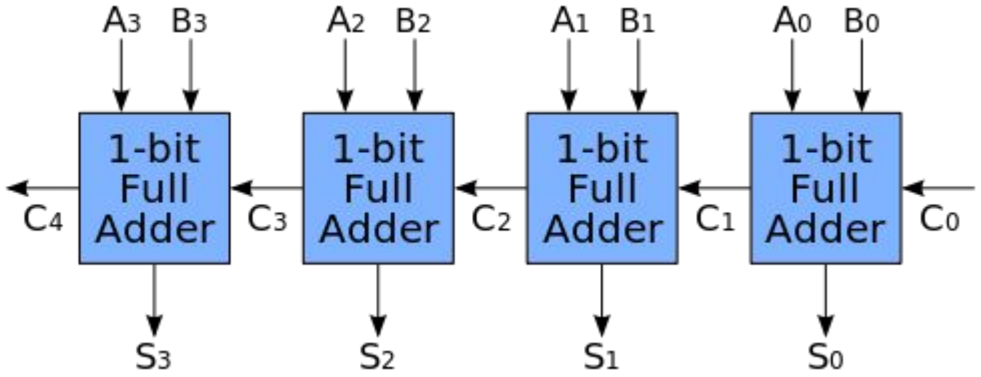
Problem:

Image source: https://commons.wikimedia.org/wiki/File:4-bit_ripple_carry_adder.svg

Implementation 1: (Naive ripple carry adder)

We just implement the paper and pencil calculation of addition in hardware.

Easy to design and understand, simple logic



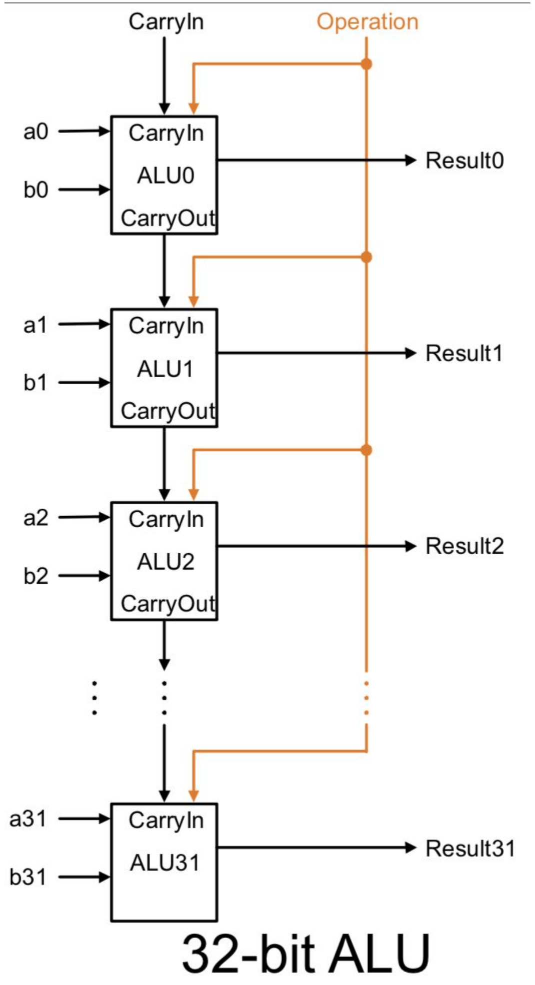
Problem: this is too slow, as each unit needs to know the carry out from the previous unit's calculation in order to continue

Image source: https://commons.wikimedia.org/wiki/File:4-bit_ripple_carry_adder.svg

Ripple-carry Adder(full view)

- Ripple carry adder has an intrinsic dependency on the **carryIn** bits.
- In other words, each unit adder needs to wait for the previous **carryOut** to be computed.

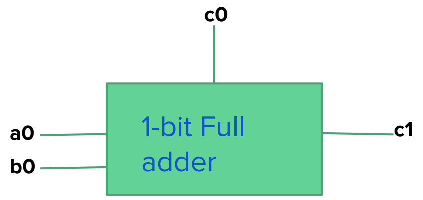
What's the solution to this?



It's made by putting 32 1-bit ALUs together

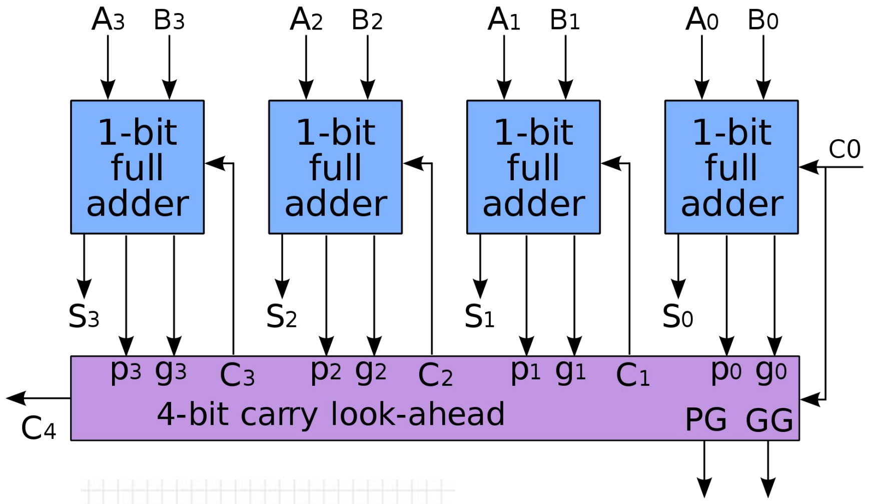
Implementation 2: (Carry look ahead)

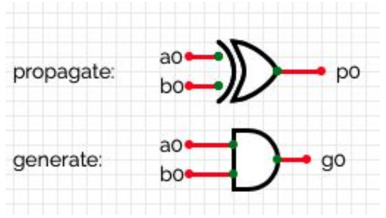
- We can exploit some information available at the beginning of the computation to speed up the process.
    - We know all the input bits (`a0,b0; a1,b1; a2,b2; a3,b3`) at the beginning.
- Let’s use them in a smart way to accelerate our computation:
    - If **a0==1 and b0==1**, we will certainly have a carry out bit from the first full adder.
        - Idea of generate (certainly have a carry): **g0 = a0 AND b0 (g0 = a0b0)**
    - If **a0==1 and b0==0**, or **a0==0 and b0==1**, and once we have a carry **(i.e, c0==1)** then we will have a carry out bit.
        - Idea of propogate (potential to have a carry): **p0=a0 XOR b0 ($p0 = a0 \oplus b0$)**



More parallelism, more hardware…

Carry-Lookahead Adder (CLA)





pi = Ai XOR Bi
gi = Ai AND Bi

c0: ready at the beginning of the computation.
c1 = g0 + c0p0
c2 = g1 + g0p1 + c0p0p1
c3 = g2 + g1p2 + g0p1p2 + c0p0p1p2
c4 = g3 + g2p3 + g1p2p3 + g0p1p2p2 +c0p0p1p2p3

We discussed the cases for producing a carry out in a single full adder in the previous slide. We can reason the same way here:

- **g3:** (when set to 1) means the last adder certainly will produce a carry out.
- **g2p3:** (when both set to 1) means the third adder will certainly produce a carry out, so the last adder will certainly have a carry in.
- ...

According to previous logic, if only one of the input bits is 1 and the other is 0, then we will certainly have a carry out once we have a carry in.

Notice: for an N-bit adder, the last product in our equations has N+1 terms. And we will have sum of N+1 products in the Nth carry out bit.

Image source: https://en.wikipedia.org/wiki/Carry-lookahead_adder#/media/File:4-bit_carry_lookahead_adder.svg

Carry Lookahead Adder (example)

- Calculate C4 using the pattern from C1-C3.

  C4 = G3 + G2*P3 + G1*P2*P3 + G0*P1*P2*P3 + C0*P0*P1*P2*P3

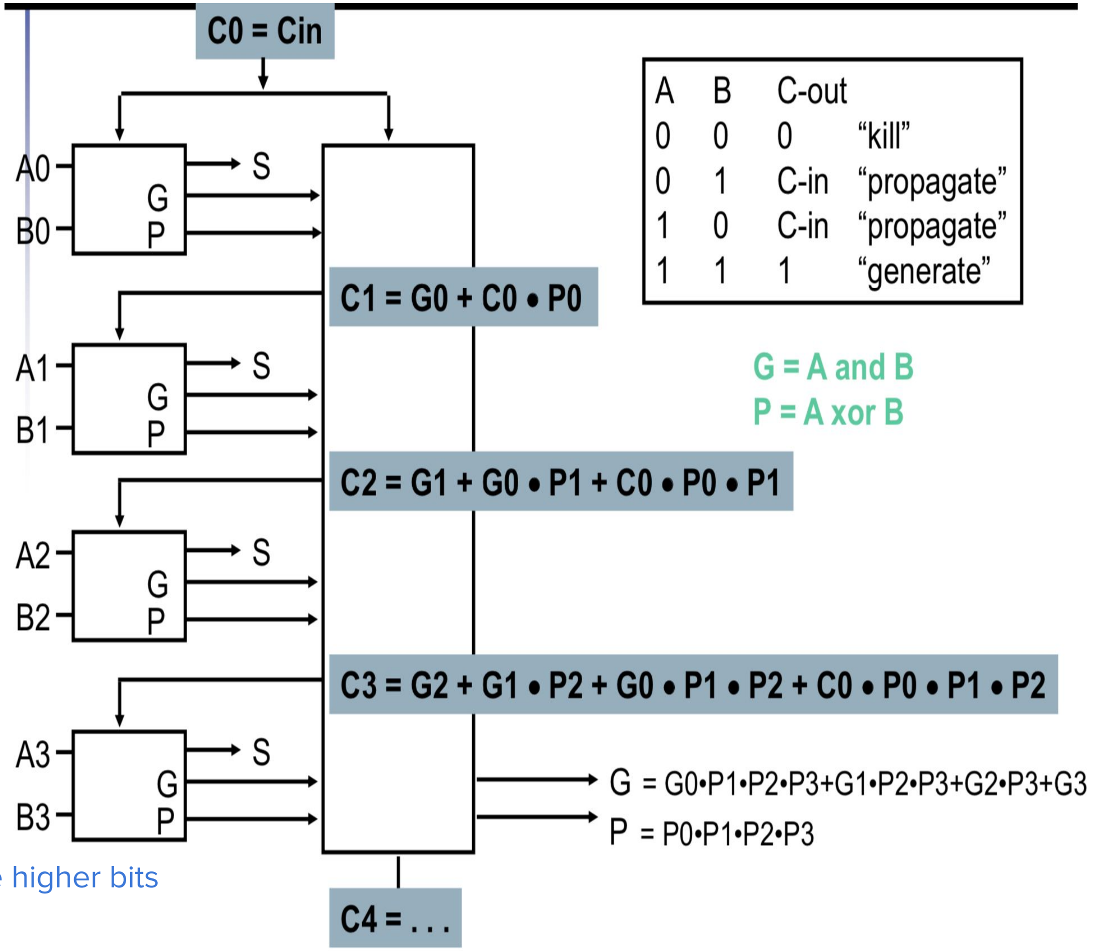
Note: We can treat C0 as G-1 for simpler reasoning

- What about C5?

  C5 = G4 + G3*P4 + G2*P3*P4 + G1*P2*P3*P4 + G0*P1*P2*P3*P4 + C0*P0*P1*P2*P3*P4

What's the problem with this design?

A: Equations can get quite long as we progress to the higher bits



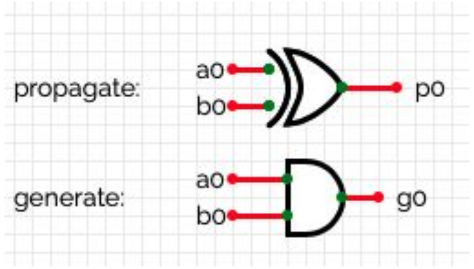
C0 = Cin
A0
B0
G
P
S
C1 = G0 + C0 • P0
A1
B1
C2 = G1 + G0 • P1 + C0 • P0 • P1
A2
B2
C3 = G2 + G1 • P2 + G0 • P1 • P2 + C0 • P0 • P1 • P2
A3
B3
G = G0•P1•P2•P3+G1•P2•P3+G2•P3+G3
P = P0•P1•P2•P3
C4 = . . .
G = A and B
P = A xor B

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

Carry Lookahead Adder(example)

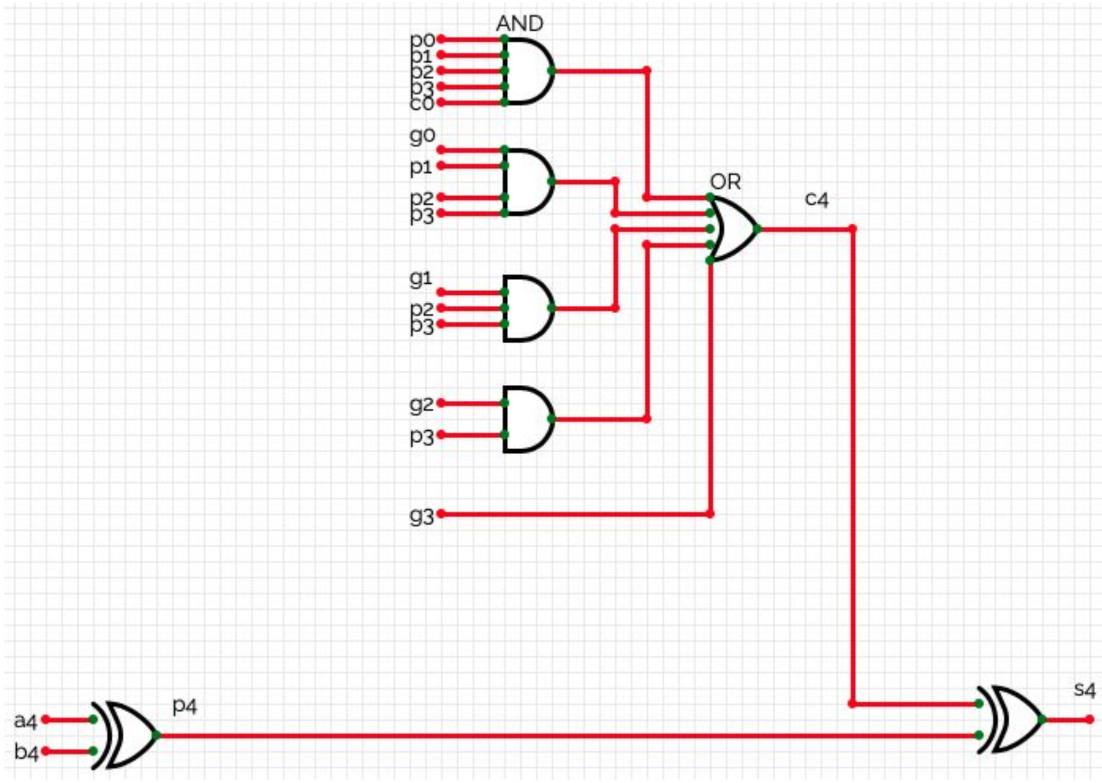
Visualization

c4 = g3 + g2p3 + g1p2p3 + g0p1p2p3 +c0p0p1p2p3

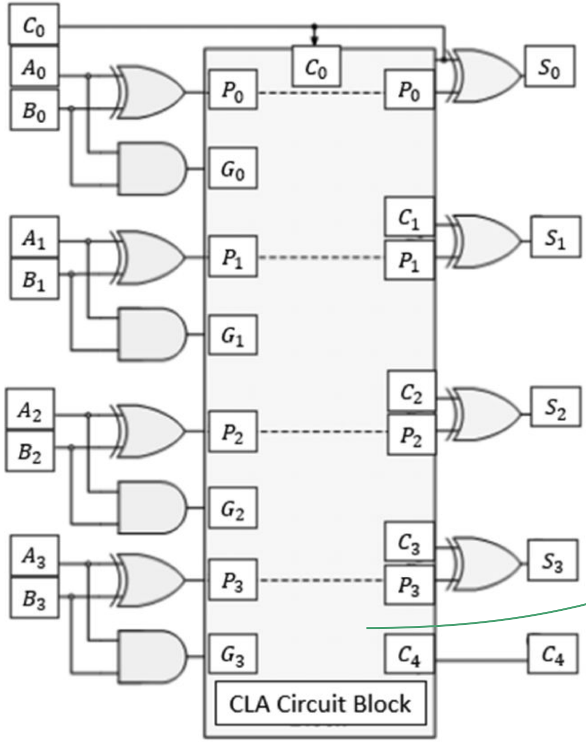


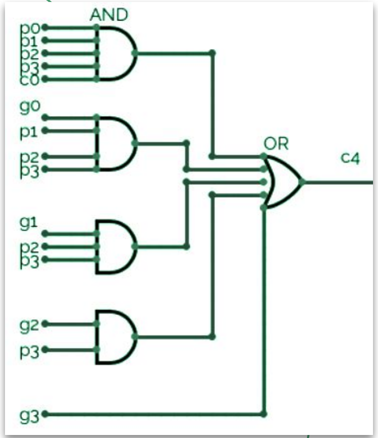
Note: S4 is not calculated within this 4-bit CLA but included for the demonstration purposes, as we only calculate s0-s3 in this 4-bit adder

Traditional 4-bit CLA S4 calculation



4-bit Carry Lookahead Adder(overview)





Img src: https://www.researchgate.net/figure/Block-diagram-of-4-bit-CLA-adder_fig1_338214910

Practice Questions

P1 (architectural tradoff)

Suppose we change our opcode to 4 bits to accommodate larger immediates for J and I type instructions. For I-type instructions, we get rid of some of the less useful instructions, and for R-type instructions we do not use the extra 2 bits (saved from opcode 6 bits -> 4 bits).

1. What are the potential impacts of such a change?
2. What if we increased rs, rt, rd to 6-bit fields, and took out bits from the R-type shamt and I-type Immediate fields?

P1 (architectural tradoff)

Suppose we change our opcode to 4 bits to accommodate larger immediates for J and I type instructions. For I-type instructions, we get rid of some of the less useful instructions, and for R-type instructions we do not use the extra 2 bits (saved from opcode 6 bits -> 4 bits).

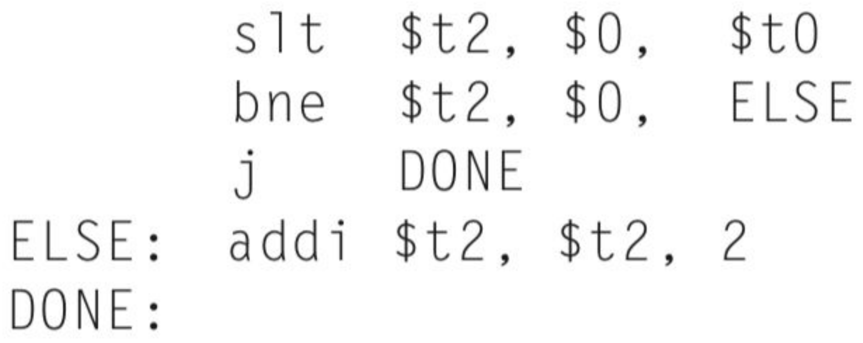
1. What are the potential impacts of such a change?
2. What if we increased rs, rt, rd to 6-bit fields, and took out bits from the R-type shamt and I-type Immediate fields?

Ans:
R is not affected. Because opcode for R is 0, and R variety only depends on func code
Less spilling so less ic for load store -> proportion of load/store changed so overall cpi affected
Increased instruction count bc simpler instruction set

Increased ic for shifting and immediates
Decreased number of i type instructions bc larger immediate

P2 (SLT)

Assume $t0 holds the value 0x0010 1000. What is the value of $t2 after the following instructions?



```
        slt  $t2, $0,  $t0
        bne  $t2, $0,  ELSE
        j    DONE
ELSE:   addi $t2, $t2, 2
DONE:
```

A: 3

P3 (propagate and generate)

Determine gi, pi value of these two 16 bit numbers **for i=0-7**
Assuming a0, b0 is least significant bit (right most)

```
a:        0001  1010  0011  0011two
b:        1110  0101  1110  1011two
```

Also, what is CarryOut15

P3 (propagate and generate)

Determine gi, pi value of these two 16 bit numbers **for i=0-7**
Assuming a0, b0 is least significant bit (right most)

a: $0001\ 1010\ 0011\ 0011_{two}$
b: $1110\ 0101\ 1110\ 1011_{two}$

Also, what is CarryOut15

pi = Ai XOR Bi
gi = Ai AND Bi

Ans:

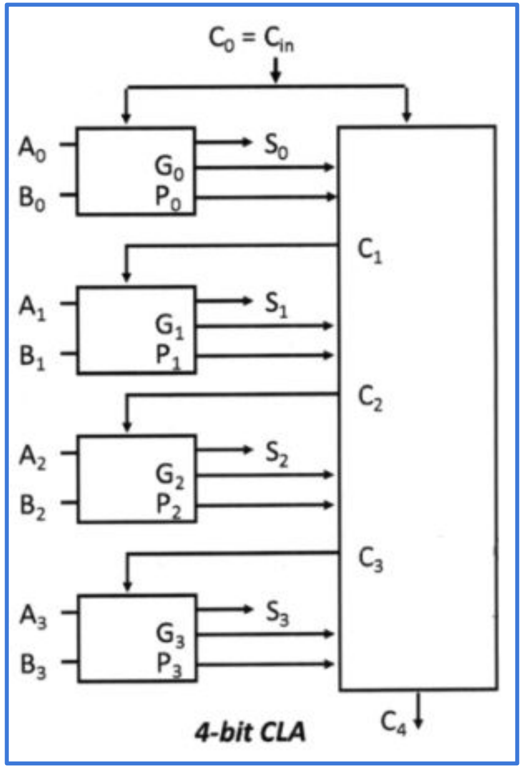
i	7	6	5	4	3	2	1	0
g	0	0	1	0	0	0	1	1
p	1	1	0	1	1	0	0	0

carryout : 1

P4 (4-bit CLA)

Unspoken assumption: Fastest delay

Assume the logic gates have the following delays, and fill in the result table for 4-bit CLA



Fan in	Delays
2	4T
3	6T
4	9T
5	13T
6	17T

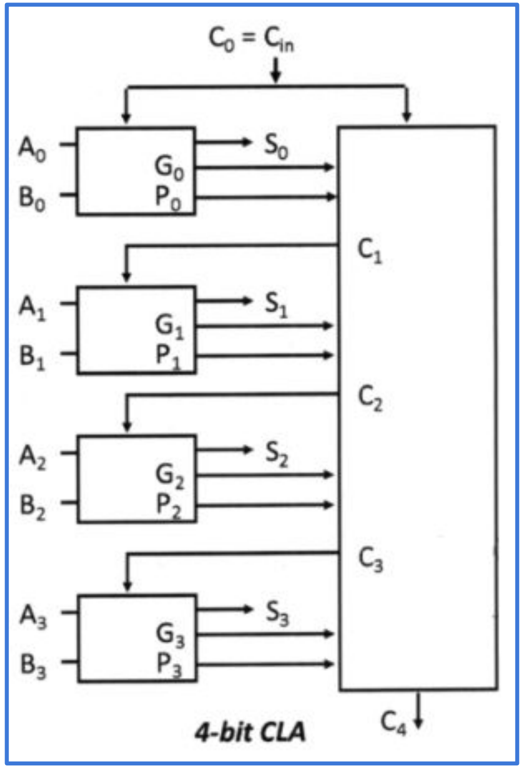
result table

output	delay
G0	4T
P0	4T
G3	4T
P3	4T
C1	12T
S0	16T

This question was collected from a previous midterm exam

P4 (4-bit CLA)

Assume the logic gates have the following delays, and fill in the result table for 4-bit CLA



c1 = g0 + c0p0
4T(getting g,p)+4T(2inputand)+4T(2inputor) = 12T

s1 = (a1 xor b1) xor c1 //a0 xor b0 is computed while c1 is being computed
12T+4T(2inputxor) = 16T

Fan in	Delays
2	4T
3	6T
4	9T
5	13T
6	17T

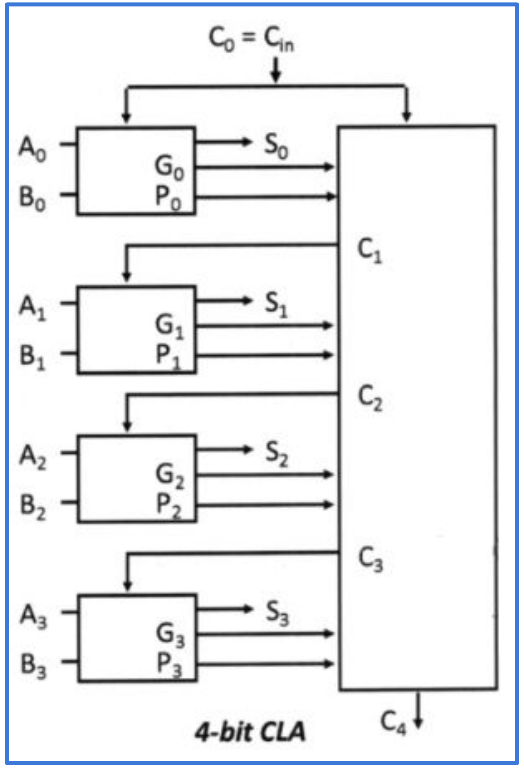
result table

output	delay
G0	4T
P0	4T
G3	4T
P3	4T
C1	12T
S1	16T

This question was collected from a previous midterm exam

P5 (4-bit CLA)

Assume the logic gates have the following delays, and fill in the result table for 4-bit CLA



Fan in	Delays
2	4T
3	6T
4	9T
5	13T
6	17T

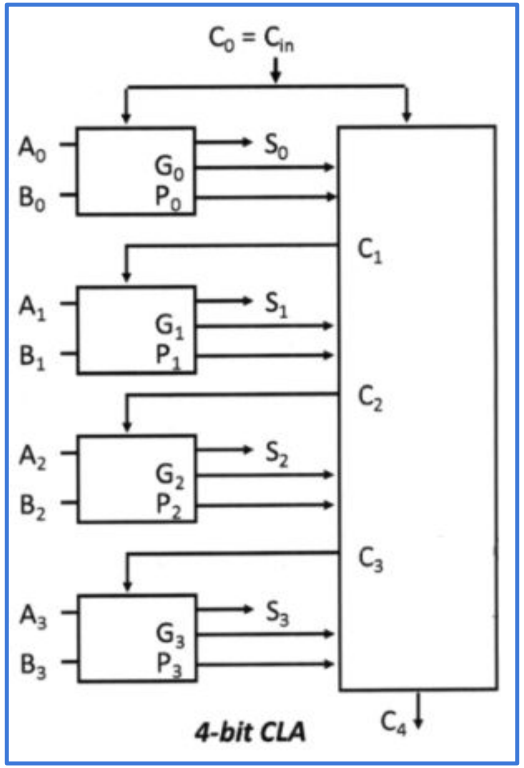
result table

output	delay
G0	4T
P0	4T
G3	4T
P3	4T
C3	22T
S3	26T
C4	30T

This question was collected from a previous midterm exam

P5 (4-bit CLA)

Assume the logic gates have the following delays, and fill in the result table for 4-bit CLA



Fan in	Delays
2	4T
3	6T
4	9T
5	13T
6	17T

result table

output	delay
C3	22T
S3	26T
C4	30T

c3 = g2 + p2g1 + p2p1g0 + p2p1p0c0
4T(getting g,p)+9T(4inputand)+9T(4inputor) = 22T

s3 = (a2 xor b2) xor c3 //a2 and b2 can be precomputed
c3 = 22T
s3 <- 22T+4T(2inputxor) = 26T

c4 = g3 + g2p3 + g1p2p3 + g0p1p2p2 +c0p0p1p2p3
4T(getting g,p)+13T(5inputand)+13T(5inputor) = 30T

This question was collected from a previous midterm exam

P6 (review, if time permits)

Assume a program requires the execution of 50×10^6 FP instructions, 110×10^6 INT instructions, 80×10^6 L/S instructions, and 16×10^6 branch instructions. The CPI for each type of instruction is 1, 1, 4, and 2, respectively. Assume that the processor has a 2 GHz clock rate.

1) By how much must we improve the CPI of FP instructions if we want the program to run two times faster?
2) By how much is the execution time of the program improved if the CPI of INT and FP instructions is reduced by 40% and the CPI of L/S and Branch is reduced by 30%?

P6 (review, if time permits)

Assume a program requires the execution of 50×10^6 FP instructions, 110×10^6 INT instructions, 80×10^6 L/S instructions, and 16×10^6 branch instructions. The CPI for each type of instruction is 1, 1, 4, and 2, respectively. Assume that the processor has a 2 GHz clock rate.

1) By how much must we improve the CPI of FP instructions if we want the program to run two times faster? ans: It is not possible. The total cycles needed for FP instructions is less than half of total cycles required for the whole program so even if we made FP have a CPI of 0, the program could not be run in less than half the time.
2) By how much is the execution time of the program improved if the CPI of INT and FP instructions is reduced by 40% and the CPI of L/S and Branch is reduced by 30%?

CPI for FP: 0.6
CPI for INT: 0.6
CPI L/S : 4 * 0.7 = 2.8
CPI for branch: 1.4

Original T = # instructions * # cycles per instruction * seconds per cycle = (50E6 * 1 + 110E6 * 1 + 80E6 * 4 + 16E6 * 2) / 2E9 = 0.256 s

New T = # instructions * # cycles per instruction * seconds per cycle = (50E6 * 0.6 + 110E6 * 0.6 + 80E6 * 2.8 + 16E6 * 1.4) / 2E9 = 0.171 s

Original T / New T ~ 1.50 times speedup.

P7 (review, if time permits)

If the current value of the PC is 0x0000 0008, can you use a single jump instruction to get to the PC addresses below?

1) 0010 0000 0000 0001 0100 1001 0010 0100 ?

   No, we can't, as the upper four bits of our current PC is 0000, but the target's upper four bits are 0010.

2) 0x0000 0600 ?

   Yes, this is possible. The most significant four bits of the target address is: 0000, which is the same as our PC's upper four bits, the lowest two bits of the target is 00, thus we can jump to this address in a single jump.

References:

Patterson, David, and John Hennessy. *Computer Organization and Design MIPS Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. 5th ed., Morgan Kaufmann, 2013.

Credit to Prof. (Glenn) Reinman (some practice questions were extracted from previous midterm exams)

Q&A

Questions? Feedback?

Thanks!