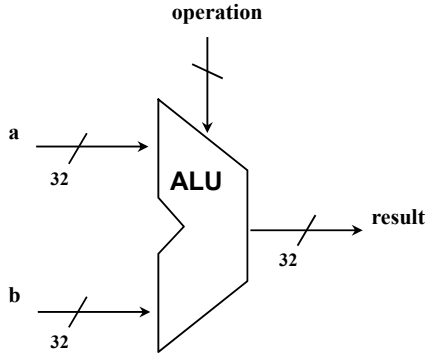
Chapter Three (1/2)

Arithmetic for Computers

- **Where we've been:**
  - **Abstractions:**
    Instruction Set Architecture
    Assembly Language and Machine Language
- **What's up ahead:**
  - **Implementing the Architecture**



Numbers

- Bits are just bits (no inherent meaning)
  — conventions define relationship between bits and numbers
- Binary numbers (base 2)
  0000 0001 0010 0011 0100 0101 0110 0111
  MSB(most significant bit) LSB(least significant bit)
  Of course it gets more complicated:
  numbers are finite (overflow)
  fractions and real numbers
  negative numbers
  e.g., no MIPS subi instruction; addi can add a negative number)
- How do we represent negative numbers?
  i.e., which bit patterns will represent which numbers?

Binary-to-Hexadecimal Conversion

- ECA8 6420
- E – 1110
  C – 1100
  A – 1010
  8 – 1000
  6 – 0110
  4 – 0100
  2 – 0010
  0 – 0000
- 1110 1100 1010 1000 0110 0100 0010 0000

Hex	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A (10)	1010
B (11)	1011
C (12)	1100
D (13)	1101
E (14)	1110
F (15)	1111

Possible Representations

Sign Magnitude	1's Complement	2's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

- Issues: balance, number of zeros, ease of operations
- Sign-magnitude
  - Where to put the sign bit? To the left? To the right?
  - Extra step to set the sign in arithmetic operations
  - Two zeros (positive zero, negative zero)???
- Which one is best?

Two's Complement Operations

- Negating a two's complement number: invert all bits and add 1
  - remember: "negate" and "invert" are quite different!

- 4 → -4
  4 = 0000 0100
  -4 = 1111 1011 + 1
  = 1111 1100
- -4 → 4
  -4 = 1111 1100
  4 = 0000 0011 + 1
  = 0000 0100

MIPS

- **32 bit signed numbers:**

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

- **The value of a 32-bit number represented in 2's complement is:**
  $(x31 * -2^{31}) + (x30 * 2^{30}) + (x29 * 2^{29}) + \ldots + (x1 * 2^{1}) + (x0 * 2^{0})$
  - **The value of a 8-bit number (1111 1000) in 2's complement?**

Signed Comparison

- **$s0 = 1111 1111 1111 1111 1111 1111 1111 1111**
  $s1 = 0000 0000 0000 0000 0000 0000 0000 0001

- **slt $t0, $s0, $s1 # signed comparison**
  sltu $t1, $s0, $s1 # unsigned comparison

- **$t0 = ?**
  $t1 = ?

- **$s0 = 4294967295 or -1**

Sign Extension

- Converting *n* bit numbers into numbers with more than *n* bits:
  - MIPS 16 bit immediate gets converted to 32 bits for arithmetic
  - copy the most significant bit (the sign bit) into the other bits

```
0010   -> 0000 0010
1010   -> 1111 1010
```

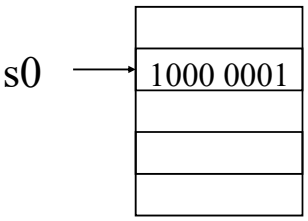
  - "sign extension" (lbu vs. lb)

```
lb      $t0, 0($s0)
```

$t0 | 1111 …1 1000 0001

```
lbu     $t0, 0($s0)
```

$t0 | 0000 …0 1000 0001



Addition & Subtraction

- Two's complement operations easy
  - subtraction using addition of negative numbers (9-6)

    x – y = x + (-y)

```
   0111          0111
 - 0110        + 1010
```

  - 6 = 0110 → -6 = 1010

- Overflow (result too large for finite computer word):
  - e.g., adding two n-bit numbers does not yield an n-bit number

```
   0111
 + 0001
   1000
```

note that overflow term is somewhat misleading, it does not mean a carry "overflowed"

Overflow

- When adding or subtracting numbers, the sum or difference can go beyond the range of representable numbers.
- This is known as overflow. For example, for two's complement numbers,

```
  5 = 0101      -5 = 1011       5 = 0101       -5 = 1011
+ 6 = 0110    + -6 = 1010     - -6 = 1010     - +6 = 0110
  --------      ---------       ---------       ---------
 -5 = 1011      5 = 0101       -5 = 1011       5 = 0101
```

- Overflow creates an incorrect result that should be detected.

Overflow Conditions

- No overflow when adding a positive and a negative number
- No overflow when signs are the same for subtraction
- Overflow occurs when the value affects the sign:

	A	B	Overflow if
A + B	>= 0	>= 0	Result < 0
A + B	< 0	< 0	Result >= 0
A – B	>= 0	< 0	Result < 0
A – B	< 0	>= 0	Result >= 0

- Consider the operations A + B, and A – B
  - Can overflow occur if B is 0 ?
  - Can overflow occur if A is 0 ?

2's Comp - Detecting Overflow

- When adding two's complement numbers, overflow will only occur if
  - the numbers being added have the same sign
  - the sign of the result is different
- If we perform the addition

$$\begin{array}{rl} & a_{n-1}\ a_{n-2}\ \ldots\ a_1\ a_0 \\ + & b_{n-1}\ b_{n-2}\ \ldots\ b_1\ b_0 \\ \hline = & s_{n-1}\ s_{n-2}\ \ldots\ s_1\ s_0 \end{array}$$

- Overflow can be detected as

$$V = a_{n-1} \cdot b_{n-1} \cdot \overline{s_{n-1}} + \overline{a_{n-1}} \cdot \overline{b_{n-1}} \cdot s_{n-1}$$

- Overflow can also be detected as

$$V = c_n \otimes c_{n-1}$$

where c_{n-1} and c_n are the carry in and carry out of the most significant bit.

Unsigned - Detecting Overflow

- For unsigned numbers, overflow occurs if there is carry out of the most significant bit.

$$V = c_n$$

- For example,

```
 1001 = 9
+1000 = 8
---------
=0001 = 1
```

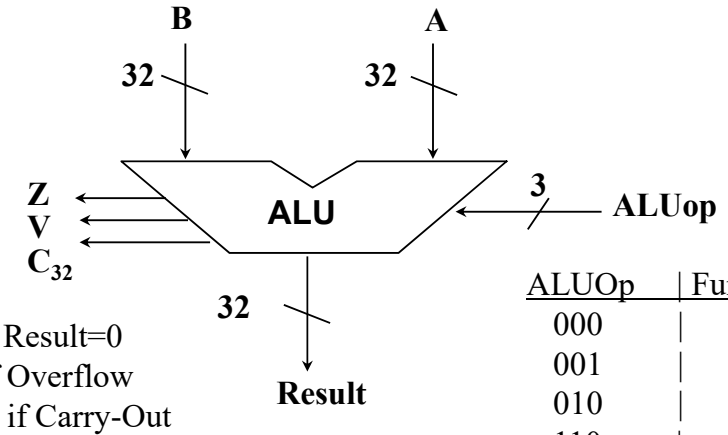
- With the MIPS architecture
  - Overflow exceptions occur for two's complement arithmetic
    - add, sub, addi
  - Overflow exceptions do not occur for unsigned arithmetic
    - addu, subu, addiu

Dealing with Overflow

- **Some languages (e.g., C) ignore overflow**
  - **Use MIPS `addu`, `addui`, `subu` instructions**
- **Other languages (e.g., Ada, Fortran) require raising an exception**
  - **Use MIPS `add`, `addi`, `sub` instructions**
  - **On overflow, invoke exception handler**
    - Save PC in exception program counter (EPC) register
    - Jump to predefined handler address
    - `mfc0` (move from coprocessor reg) instruction can retrieve EPC value, to return after corrective action

An ALU (arithmetic logic unit)

- **Let's build an ALU to support the `and`, `or`, `add`, `sub`, `beq` instructions**
  - **we'll just build a 1 bit ALU, and use 32 of them**

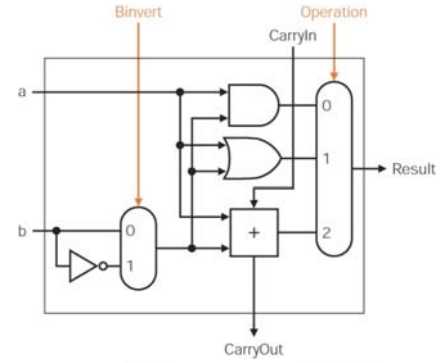


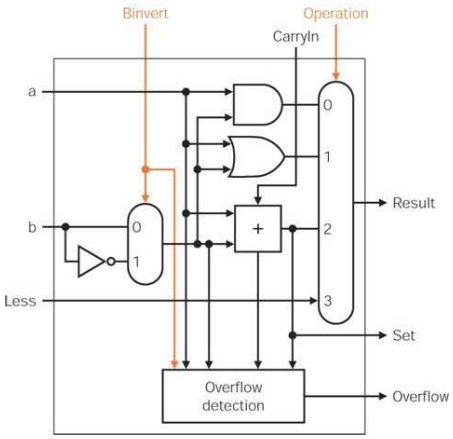
Z =1, if Result=0
V= 1, if Overflow
C_{32} = 1, if Carry-Out

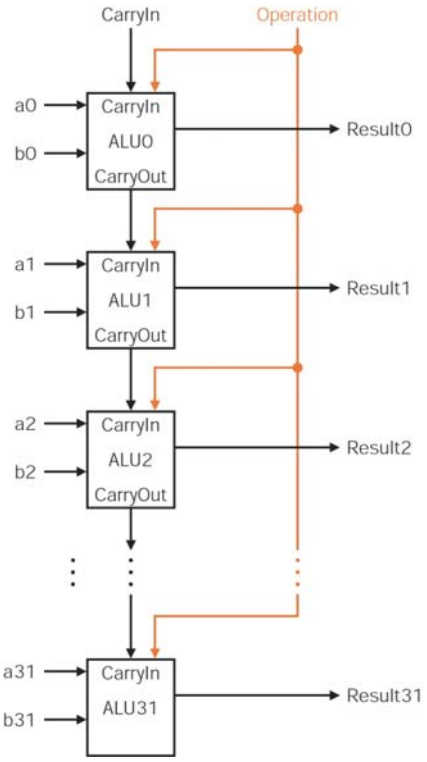
ALUOp	Function
000	and
001	or
010	add
110	subtract
111	set-on-less-than

1-Bit and 32-Bit ALU

For AND, OR, & Addition





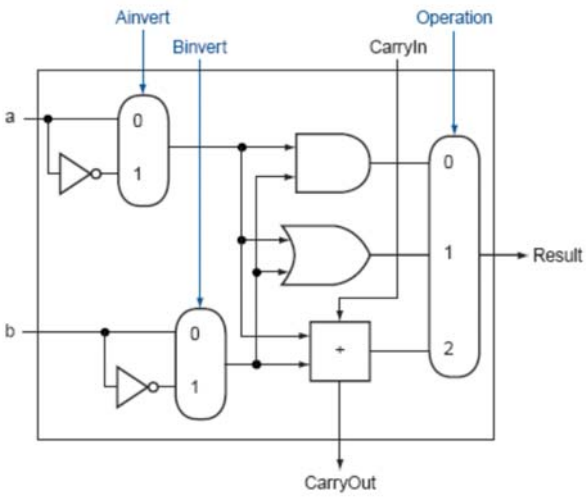


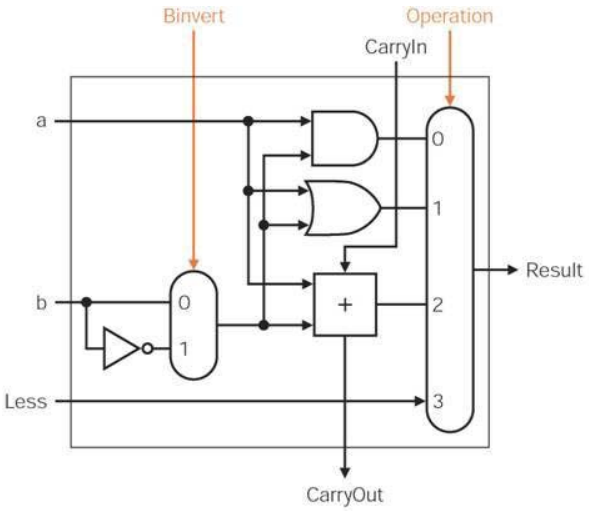
$$set = (A < B)$$

$$V = c_n \otimes c_{n-1}$$

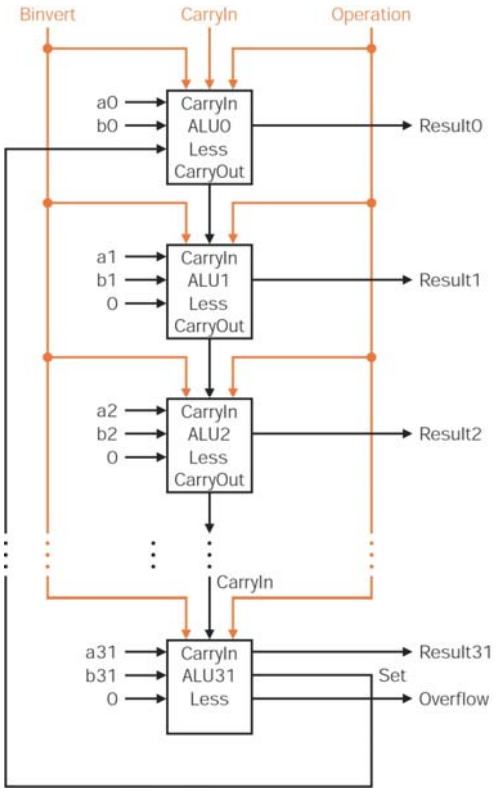
Adding a NOR & SLT

- **Choose to invert a to get "a NOR b"**
- **Need to support the set-on-less-than instruction (slt)**





Supporting SLT



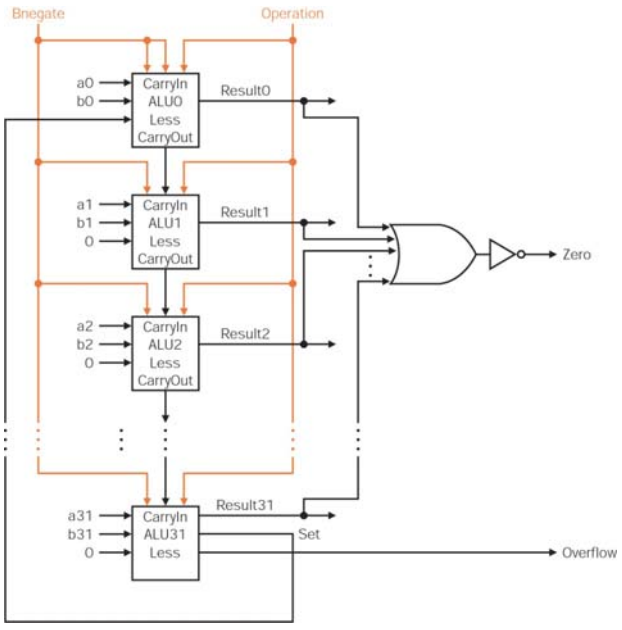
Test for equality

- Need to support test for equality (beq $t5, $t6, $t7)
  - Use comparison:

    xor all corresponding bits
  - Or use subtraction:

    (a-b) = 0 implies a = b
- Checking if zero costs a lot.
- Notice control lines:

```
000 = and
001 = or
010 = add
110 = subtract
111 = slt
```



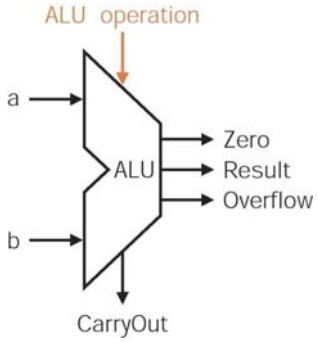
•*Note: zero is a 1 when the result is zero!*

ALU Summary

- We can build an ALU to support MIPS addition
- Our focus is on comprehension, not performance
- Real processors use more sophisticated techniques for arithmetic
- Where performance is not critical, hardware description languages allow designers to completely automate the creation of hardware!

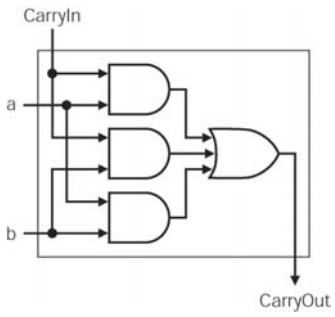
```
module MIPSALU (ALUctl, A, B, ALUOut, Zero);
   input [3:0] ALUctl;
   input [31:0] A,B;
   output reg [31:0] ALUOut;
   output Zero;
   assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0; goes anywhere
   always @(ALUctl, A, B) //reevaluate if these change
      case (ALUctl)
         0: ALUOut <= A & B;
         1: ALUOut <= A | B;
         2: ALUOut <= A + B;
         6: ALUOut <= A - B;
         7: ALUOut <= A < B ? 1:0;
         12: ALUOut <= ~(A | B); // result is nor
         default: ALUOut <= 0; //default to 0, should not happen;
      endcase
endmodule
```

FIGURE B.4.3 A Verilog behavioral definition of a MIPS ALU. This could be synthesized using a module library containing basic arithmetic and logical operations.



Problem: ripple carry adder is slow

- Is a 32-bit ALU as fast as a 1-bit ALU?
- Is there more than one way to do addition?
  - two extremes: ripple carry & sum-of-products



Can you see the ripple? How could you get rid of it?

$c_1 = b_0c_0 + a_0c_0 + a_0b_0$

$c_2 = b_1c_1 + a_1c_1 + a_1b_1$

$c_2 = b_1(b_0c_0 + a_0c_0 + a_0b_0) + a_1(b_0c_0 + a_0c_0 + a_0b_0) + a_1b_1$

$c_3 = b_2c_2 + a_2c_2 + a_2b_2$

$c_3 = ???$

$c_4 = b_3c_3 + a_3c_3 + a_3b_3$

$c_4 = ?????$

Not feasible! Why?

Carry Logic Equation

- The carry logic equation is

$$c_{i+1} = a_i b_i + (a_i + b_i) c_i$$

- We define a propagate signal

$$p_i = a_i + b_i$$

and a generate signal

$$g_i = a_i b_i$$

- This allows the carry logic equation to be rewritten as

$$c_{i+1} = g_i + p_i c_i$$

Carry-lookahead adder*

- An approach in-between our two extremes
- Goal: To produce sum and carryout for n-th bit without waiting for a carry over from (n-1)th bit.
- How???
  - For each bit position, we look into
    - whether a carry will be generated by itself? ($g_i = a_i b_i$)
    - whether a carryin could be propagated to the next bit position? ($p_i = a_i + b_i$)
  - At each bit, CarryOut is now computed based on g_i and p_i, not on c_i

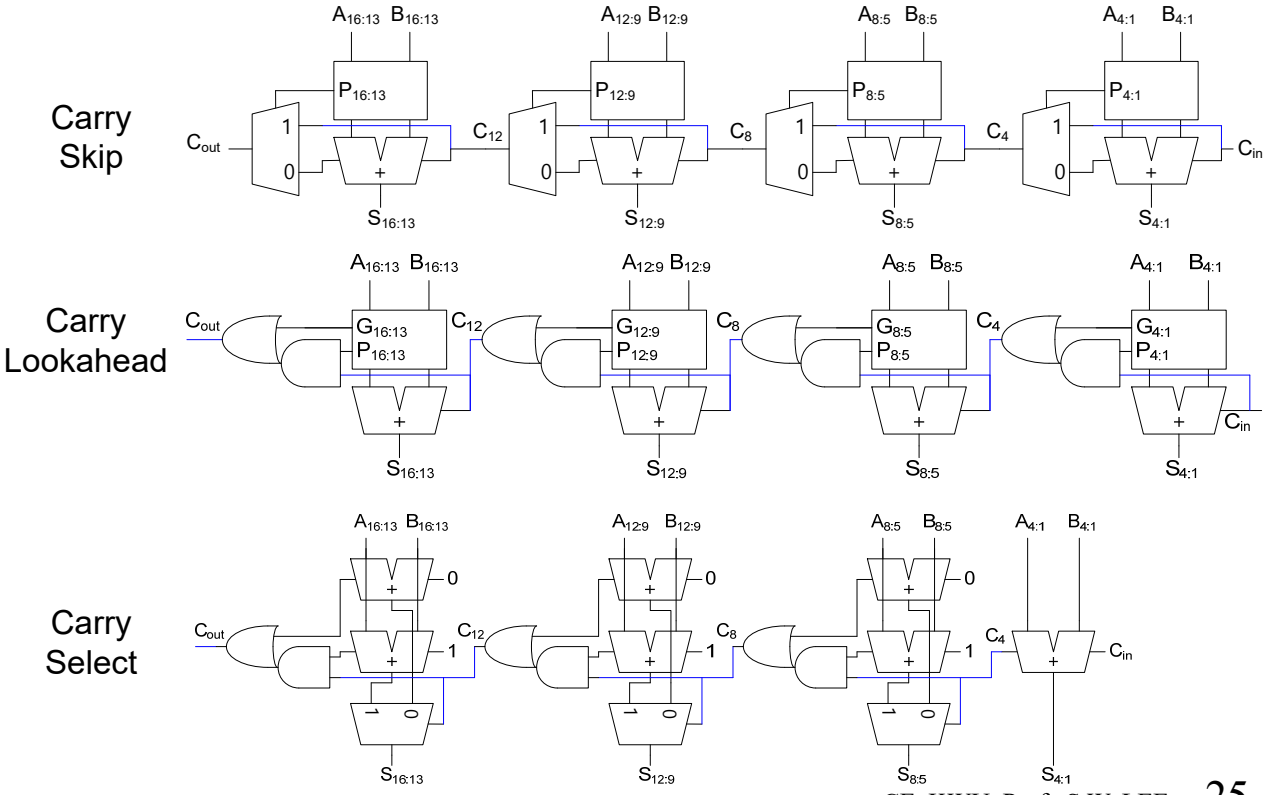
$c_{i+1} = g_i + p_i c_i$

$c_1 = g_0 + p_0 c_0$
$c_2 = g_1 + p_1 c_1$
$c_3 = g_2 + p_2 c_2$
$c_4 = g_3 + p_3 c_3$

$c_1 = g_0 + p_0 c_0$
$c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$
$c_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$
$c_4 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0$

Feasible! Why?

Various Adders

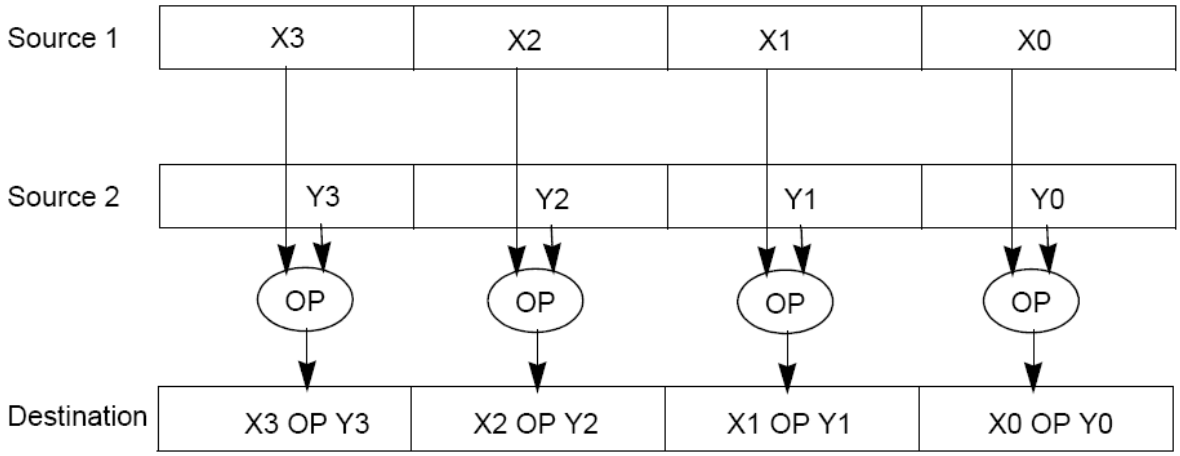


Arithmetic for Multimedia

- **Graphics and media processing operates on vectors of 8-bit and 16-bit data**
  - **Use 64-bit adder, with partitioned carry chain**
    - Operate on 8×8-bit, 4×16-bit, or 2×32-bit vectors
  - **SIMD (single-instruction, multiple-data)**
- **Saturating operations**
  - **On overflow, result is largest representable value**
    - c.f. 2s-complement modulo arithmetic
  - **E.g., clipping in audio, saturation in video**

SIMD

- **SIMD (single instruction multiple data) architecture performs the same operation on multiple data elements in parallel**
- **PADDW MM0, MM1**



Conclusion

- **We can build an ALU to support the MIPS instruction set**
  - **key idea: use multiplexor to select the output we want**
  - **we can efficiently perform subtraction using two's complement**
  - **An *n*-bit ALU can be designed by concatenating *n* 1-bit ALUs**
- **Important points about hardware**
  - **all of the gates are always working**
  - **the speed of a gate is affected by the number of inputs to the gate**
  - **the speed of a circuit is affected by the number of gates in series (on the "critical path" or the "deepest level of logic")**
  - **Carry lookahead logic can be used to improve the speed of the computation.**
  - **A variety of design options exist for implementing the ALU.**
- **The best design depends on area, delay, and power requirements, which vary based on the underlying technology.**