CS4100：計算機結構

Computer Arithmetic

國立清華大學資訊工程學系
一零零學年度第二學期

國立清華大學
National Tsing Hua University

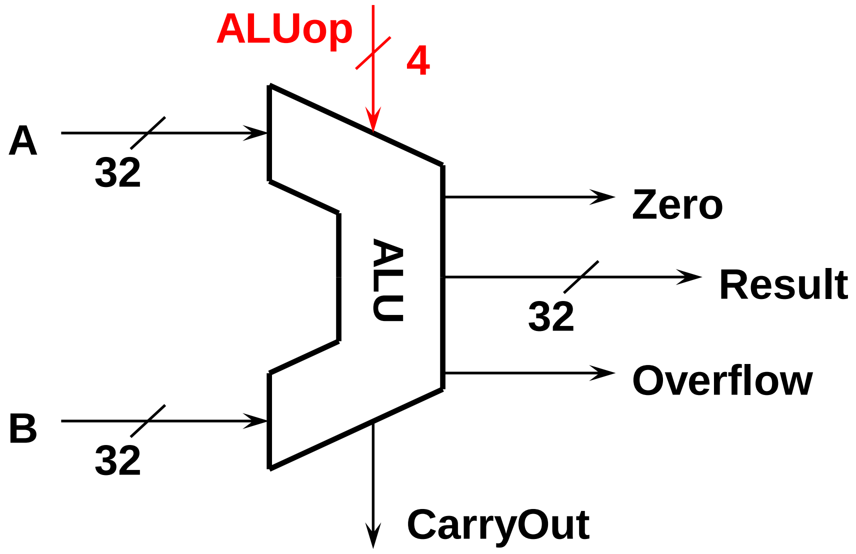
Outline

- Addition and subtraction (Sec. 3.2)
- Constructing an arithmetic logic unit (Appendix C)
- Multiplication (Sec. 3.3, Appendix C)
- Division (Sec. 3.4)
- Floating point (Sec. 3.5)

Problem: Designing MIPS ALU

- Requirements: must support the following arithmetic and logic operations
  - add, sub: two's complement adder/subtractor with overflow detection
  - and, or, nor : logical AND, logical OR, logical NOR
  - slt (set on less than): two's complement adder with inverter, check sign bit of result

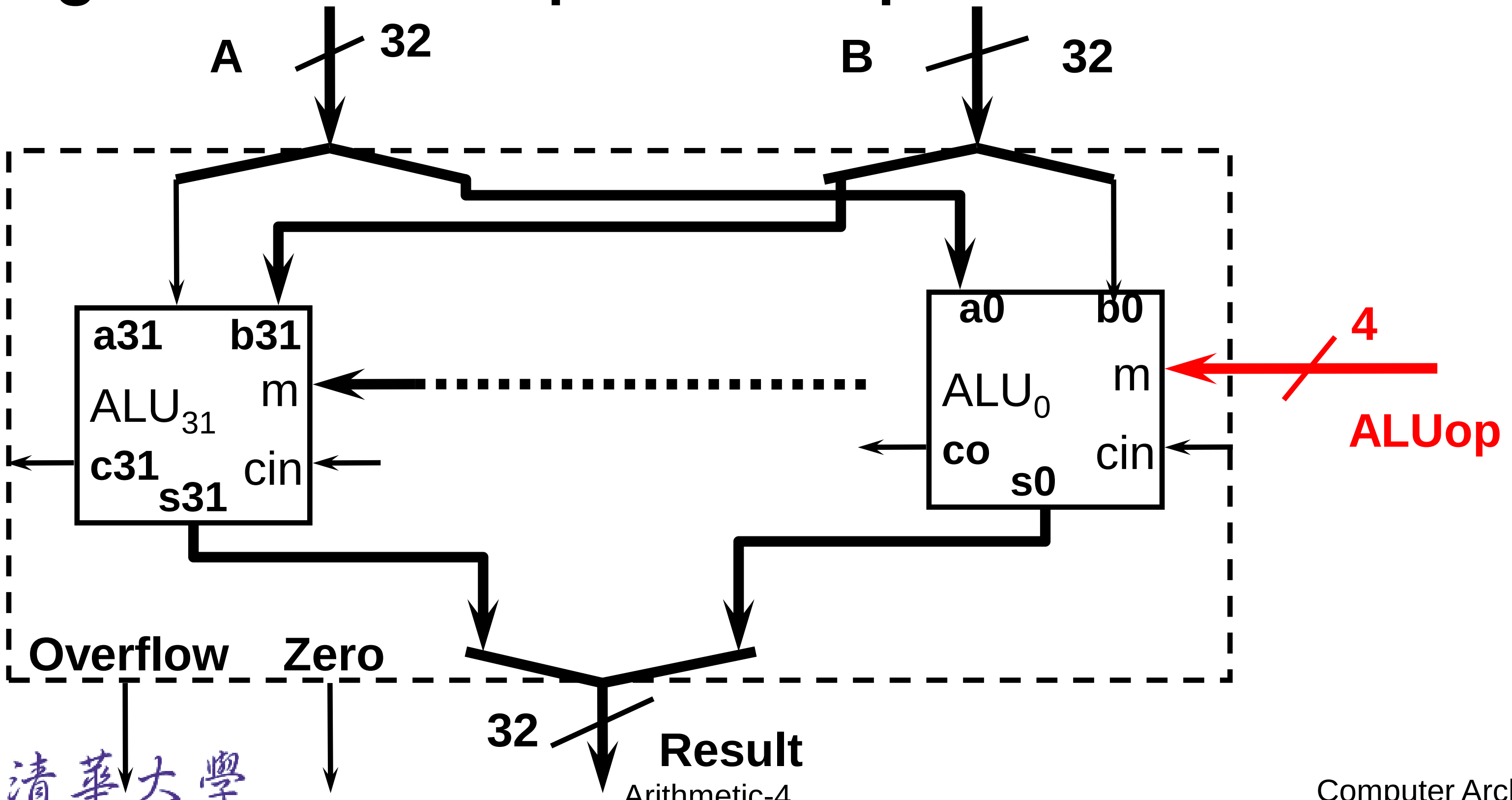
Functional Specification



ALU Control (ALUop)	Function
0000	and
0001	or
0010	add
0110	subtract
0111	set-on-less-than
1100	nor

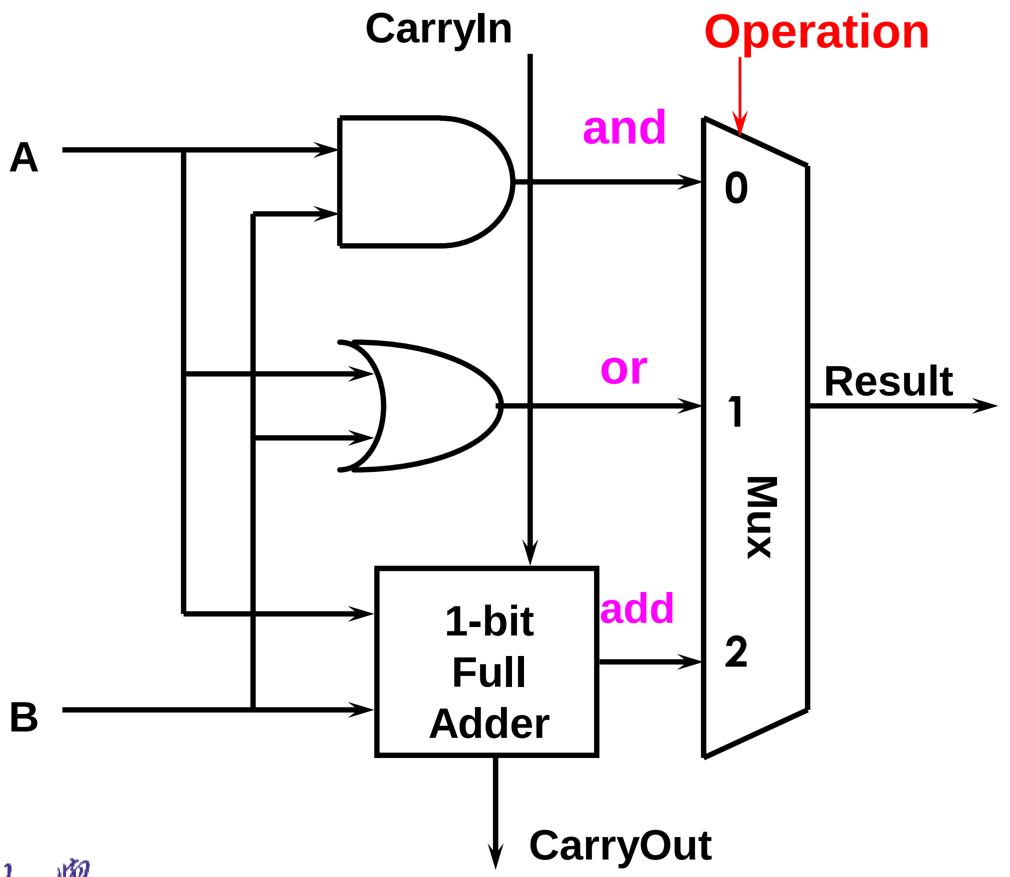
A Bit-slice ALU

- Design trick 1: divide and conquer
  - Break the problem into simpler problems, solve them and glue together the solution
- Design trick 2: solve part of the problem and extend



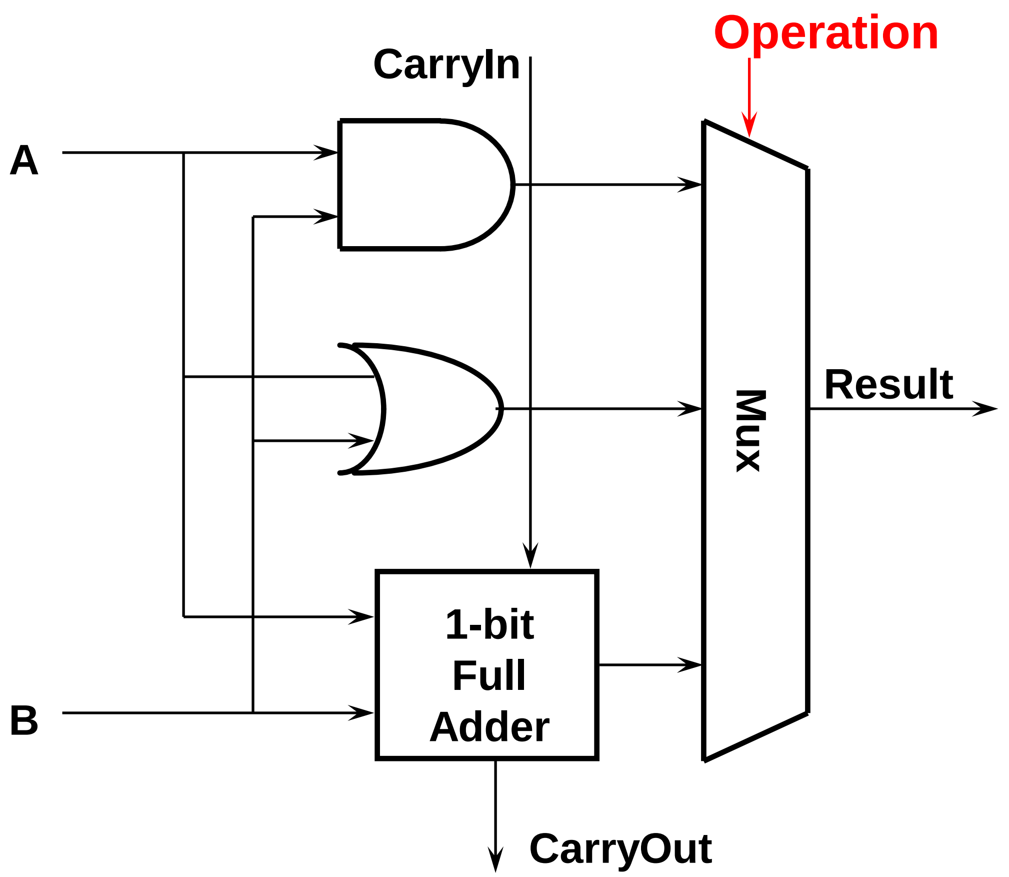
A 1-bit ALU

- Design trick 3: take pieces you know (or can imagine) and try to put them together

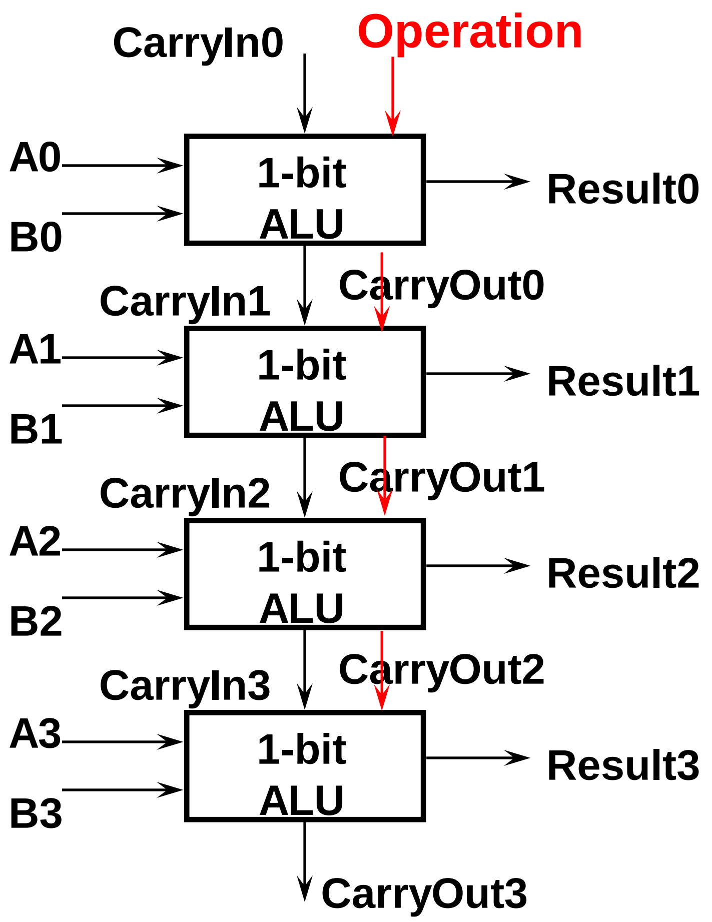


A 4-bit ALU

1-bit ALU

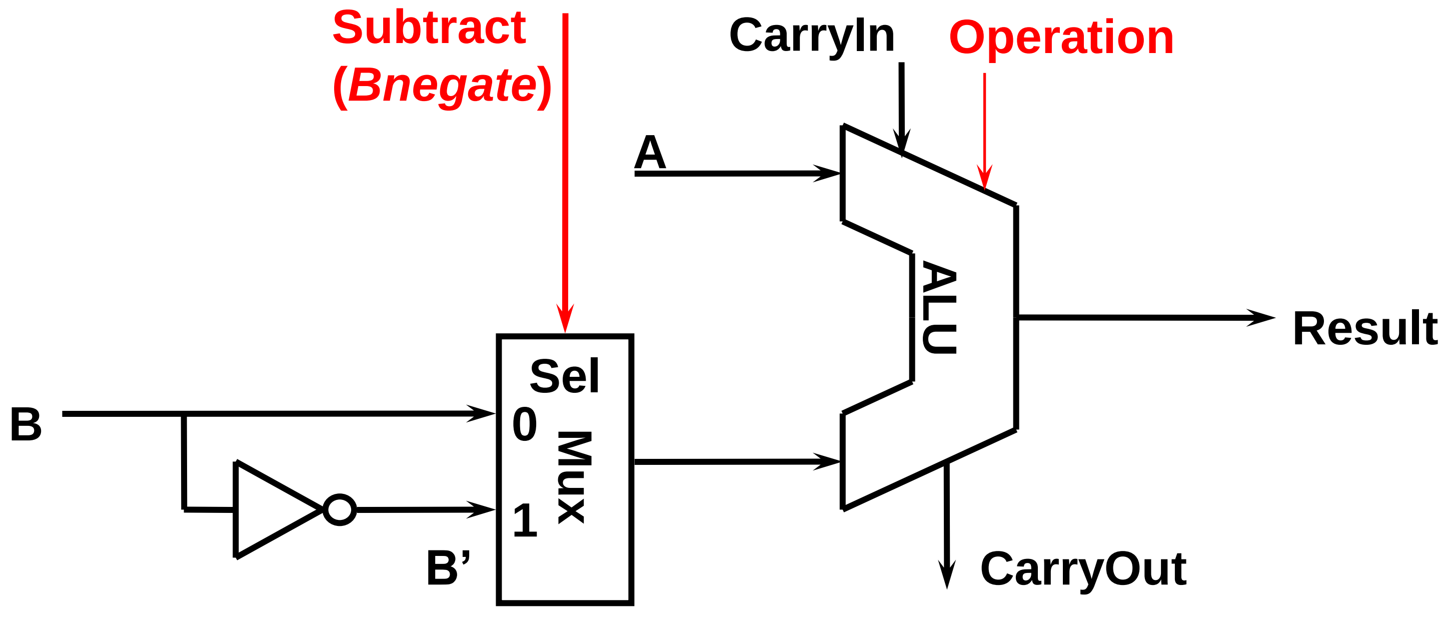


4-bit ALU



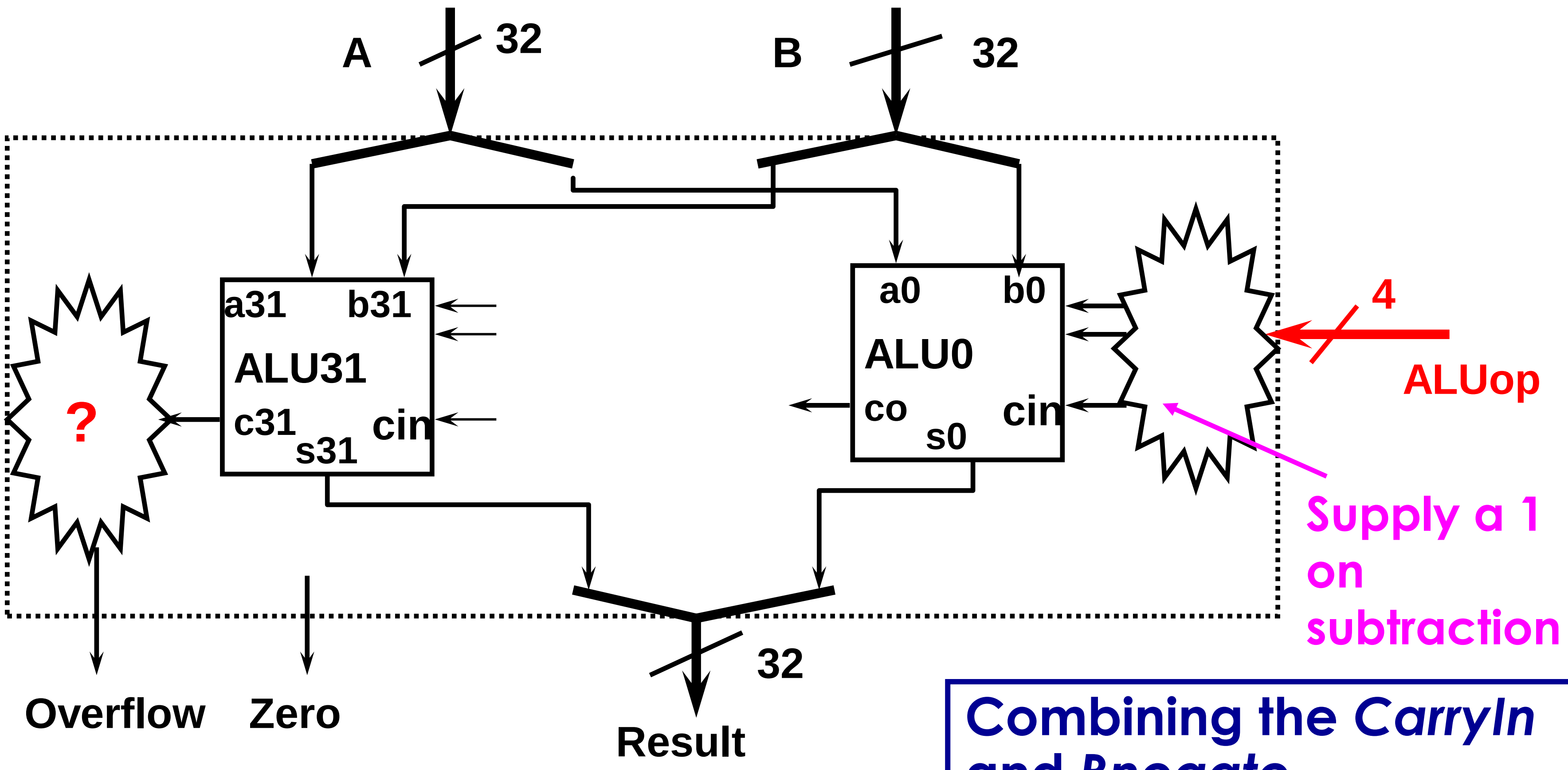
How about Subtraction?

- 2's complement: take inverse of every bit and add 1 (at c_{in} of first stage)
  - A + B' + 1 = A + (B' + 1) = A + (-B) = A - B
  - Bit-wise inverse of B is B'



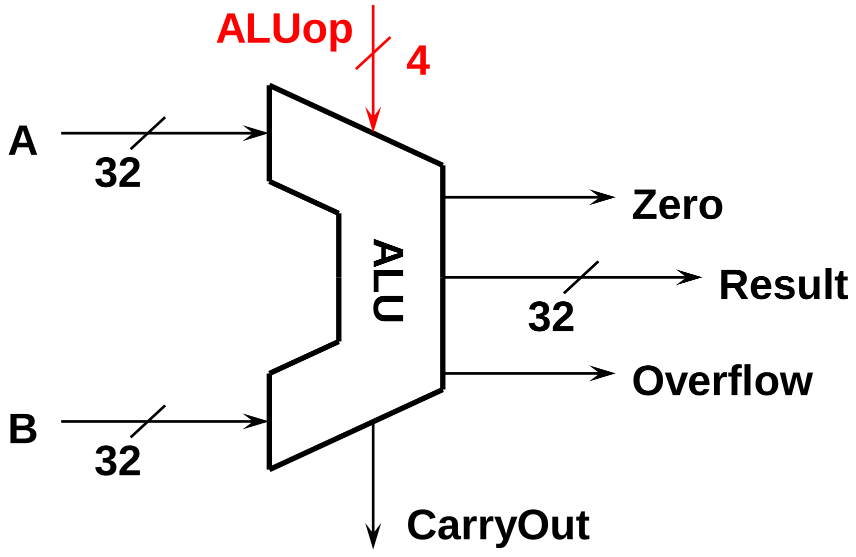
Revised Diagram

- LSB and MSB need to do a little extra



Combining the *CarryIn* and *Bnegate*

Functional Specification



ALU Control (ALUop)	Function
0000	and
0001	or
0010	add
0110	subtract
0111	set-on-less-than
1100	nor

R-Format Instructions (1/2)

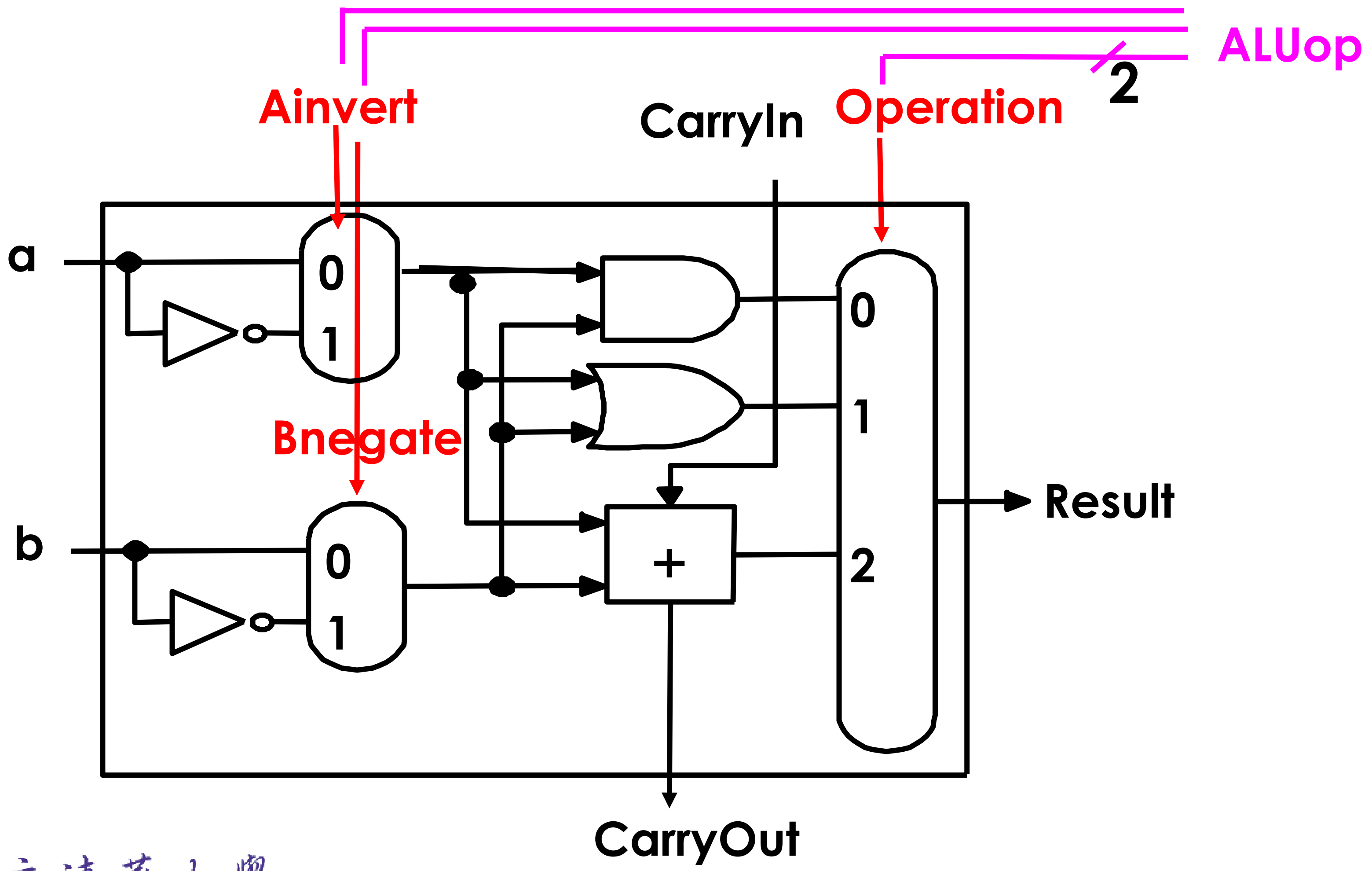
- Define the following "fields":

6	5	5	5	5	6
`opcode`	`rs`	`rt`	`rd`	`shamt`	`funct`

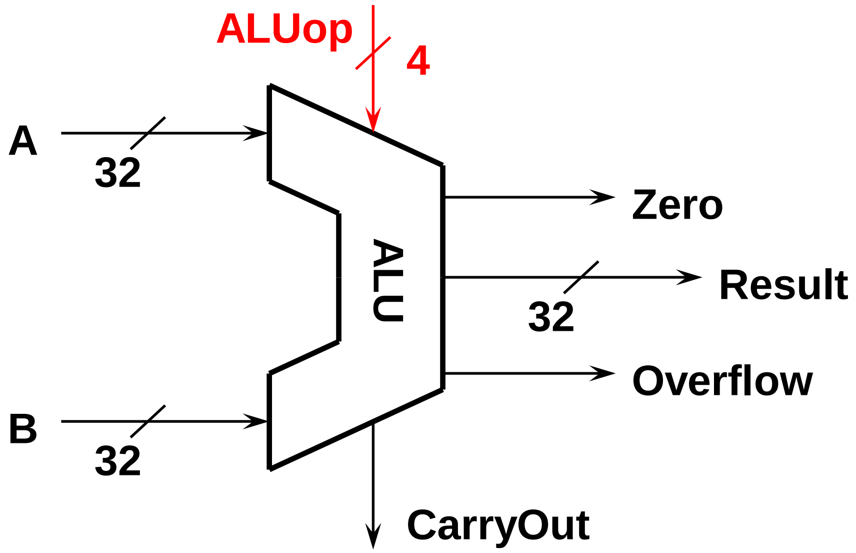
  - `opcode`: partially specifies what instruction it is (Note: 0 for all R-Format instructions)
  - `funct`: combined with `opcode` to specify the instruction
    Question: Why aren't `opcode` and `funct` a single 12-bit field?
  - `rs` (Source Register): *generally* used to specify register containing first operand
  - `rt` (Target Register): *generally* used to specify register containing second operand
  - `rd` (Destination Register): *generally* used to specify register which will receive result of computation

Nor Operation

- A nor B = (not A) and (not B)

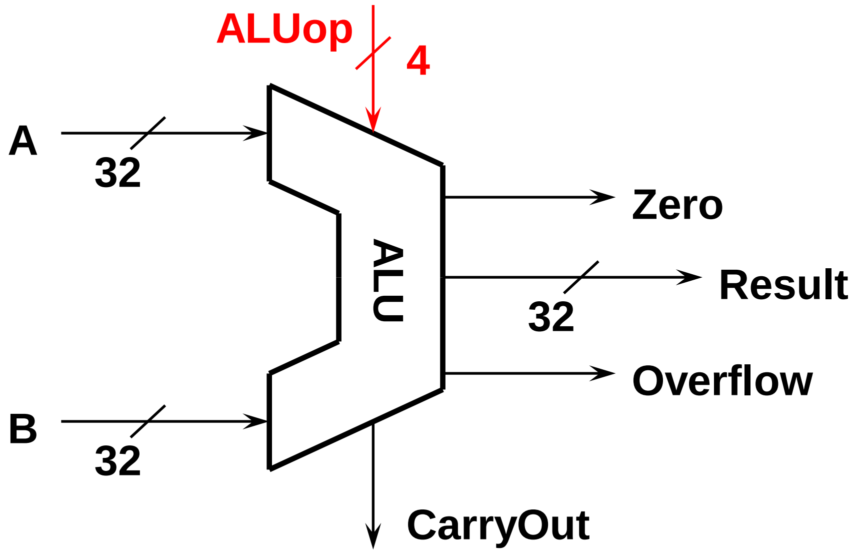


Functional Specification



ALU Control (ALUop)	Function
0000	and
0001	or
0010	add
0110	subtract
0111	set-on-less-than
1100	nor

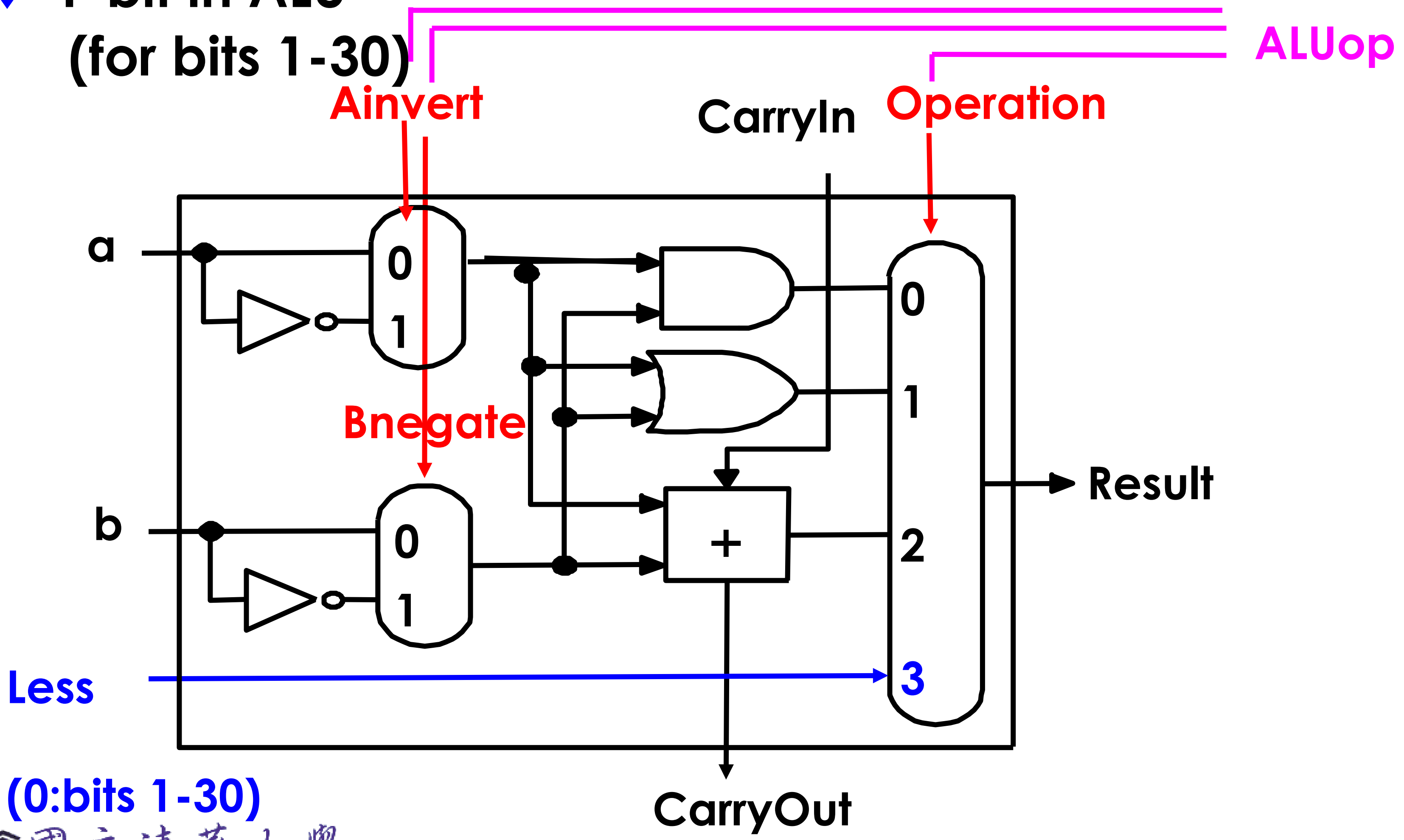
Functional Specification



ALU Control (ALUop)	Function
0000	and
0001	or
0010	add
0110	subtract
0111	set-on-less-than
1100	nor

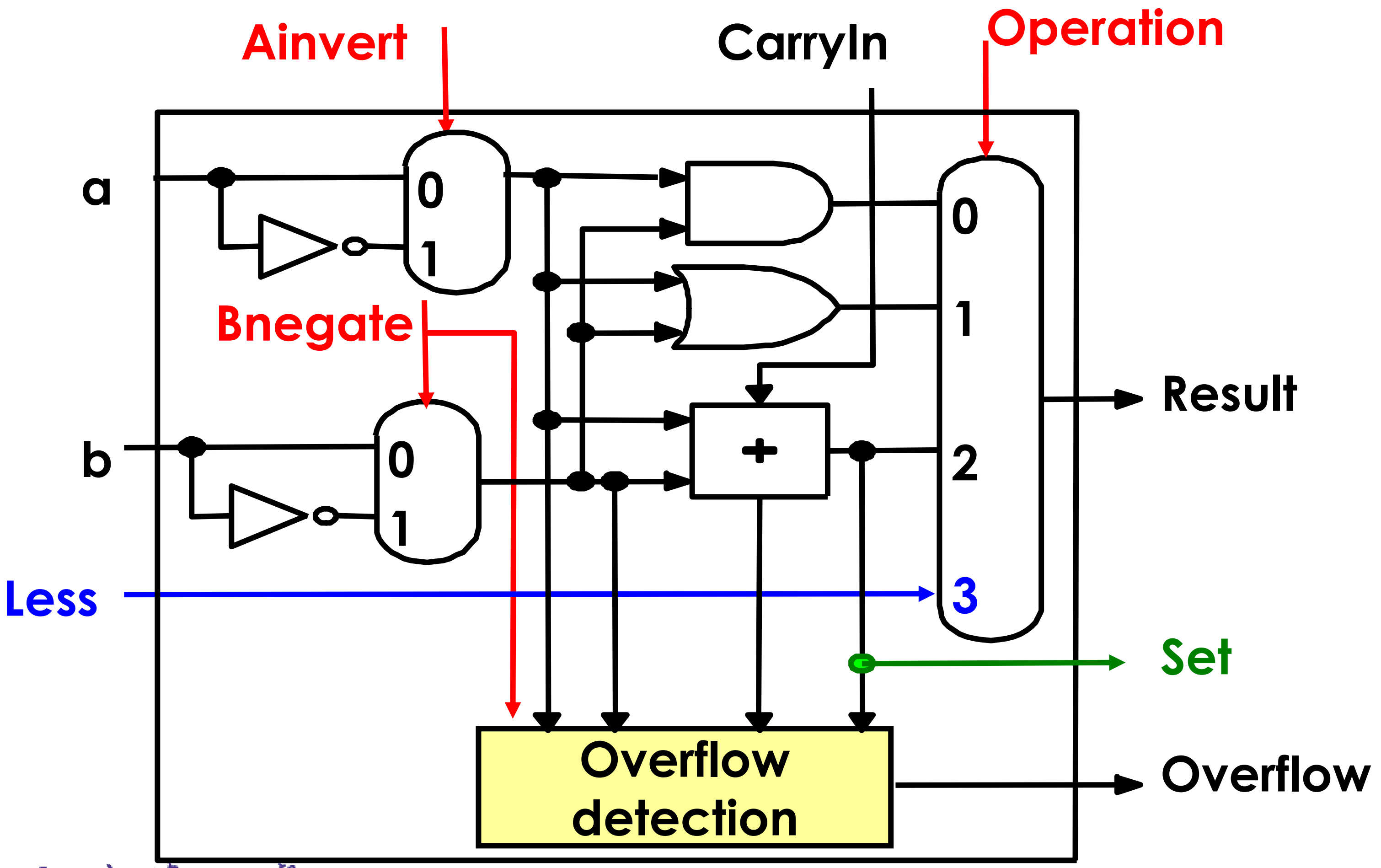
Set on Less Than (I)

- 1-bit in ALU
  (for bits 1-30)



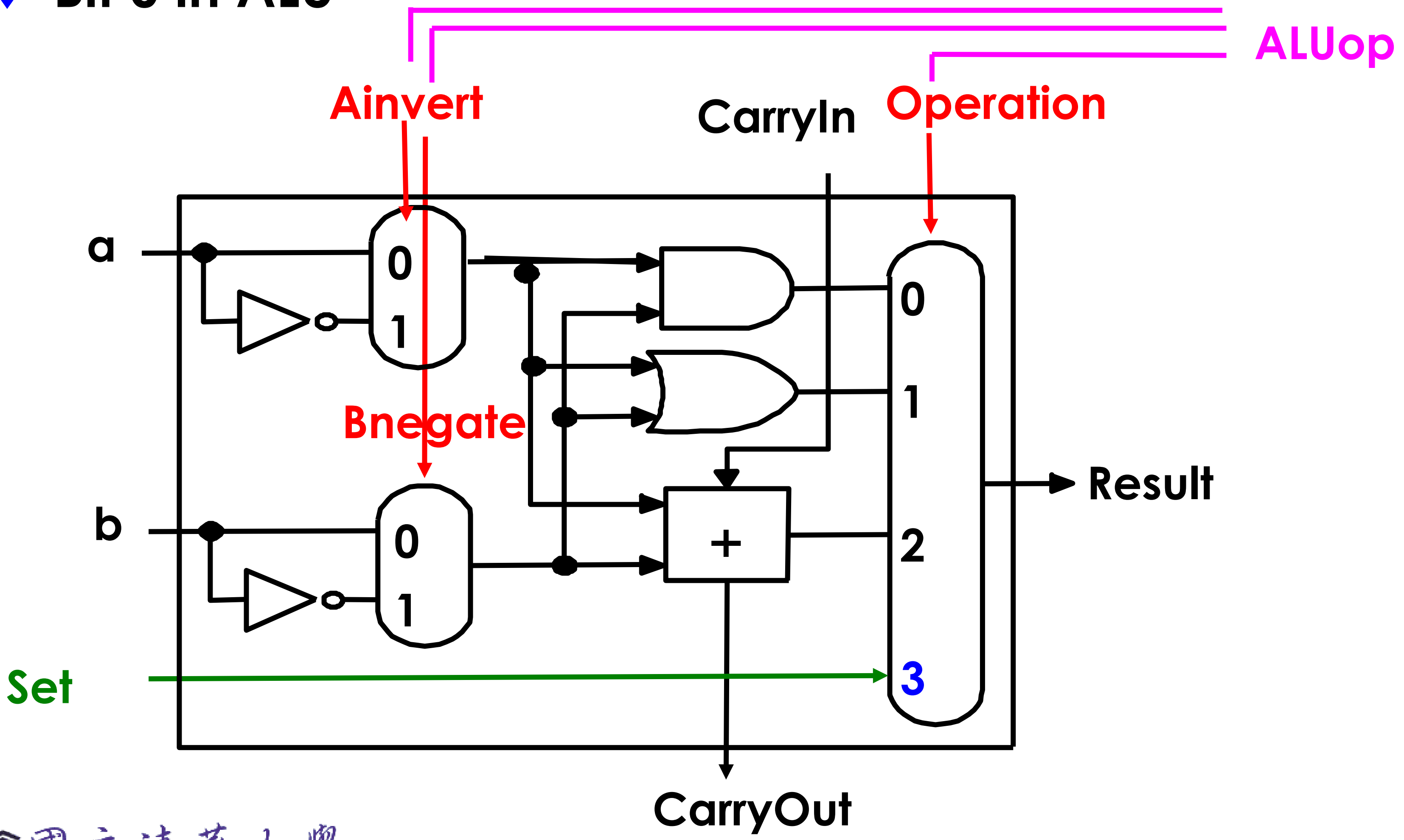
Set on Less Than (II)

- Sign bit in ALU

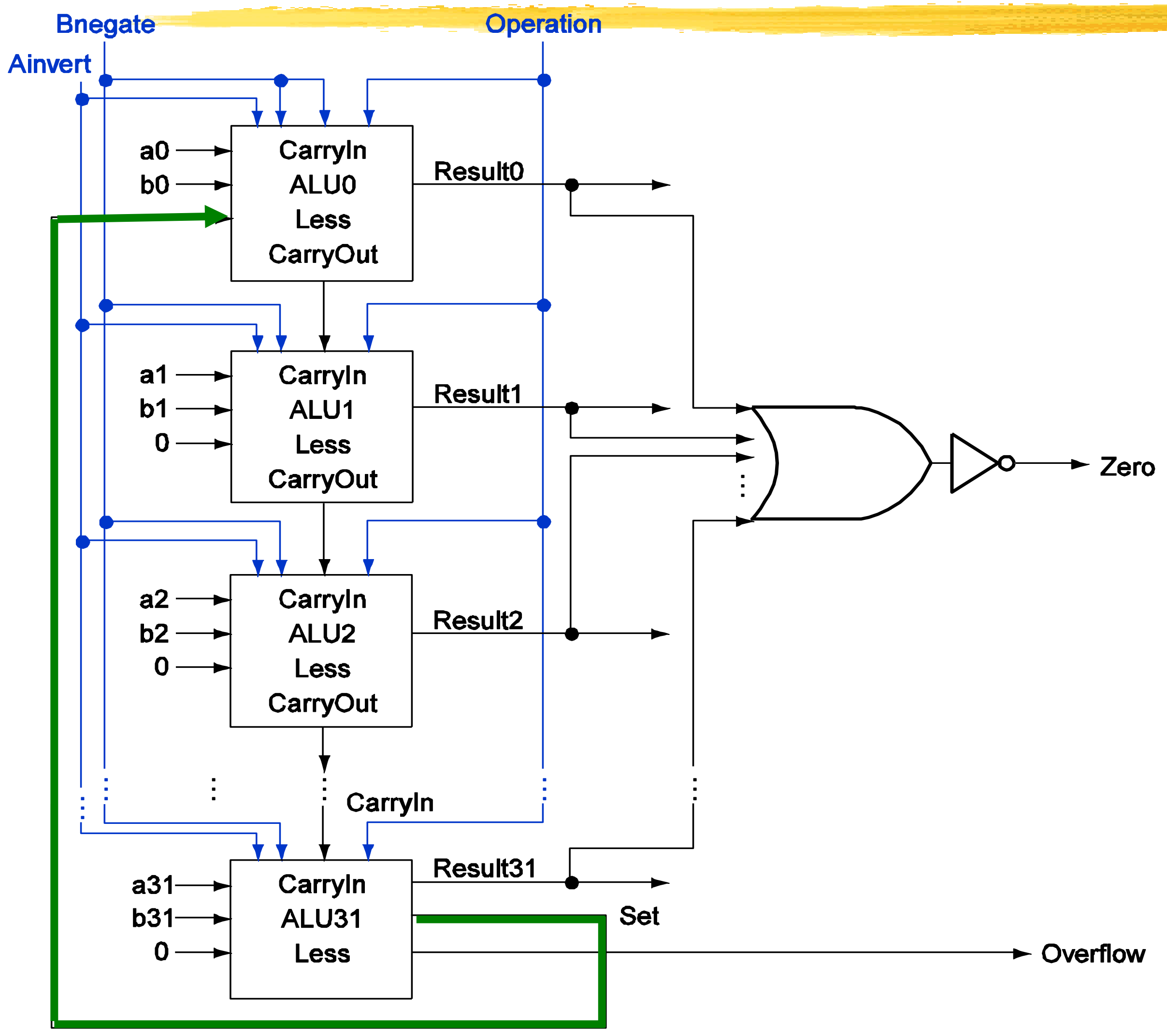


Set on Less Than (III)

- Bit 0 in ALU



A Ripple Carry Adder and Set on Less Than



ALUop	Function
0000	and
0001	or
0010	add
0110	subtract
0111	set-less-than
1100	nor

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

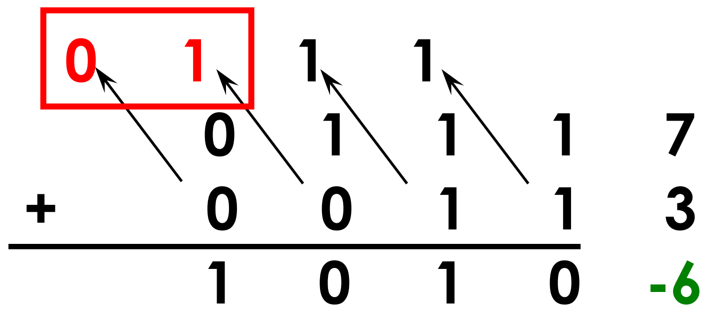
Ex: 7 + 3 = 10 but ... - 4 - 5 = - 9 but ...

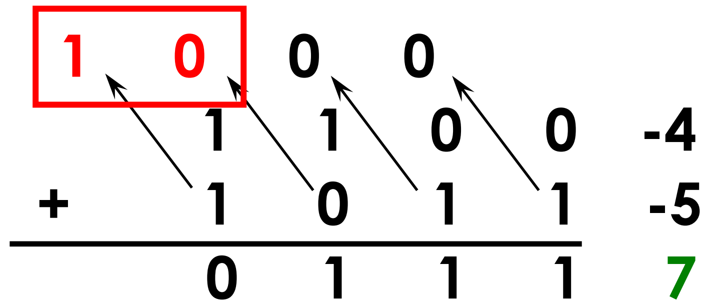
$$\begin{array}{cccccc} 0 & 1 & 1 & 1 & & \\ & 0 & 1 & 1 & 1 & 7 \\ + & 0 & 0 & 1 & 1 & 3 \\ \hline & 1 & 0 & 1 & 0 & -6 \end{array} \qquad \begin{array}{cccccc} 1 & 0 & 0 & 0 & & \\ & 1 & 1 & 0 & 0 & -4 \\ + & 1 & 0 & 1 & 1 & -5 \\ \hline & 0 & 1 & 1 & 1 & 7 \end{array}$$

Overflow Detection

- Overflow: result too big/small to represent
  - -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

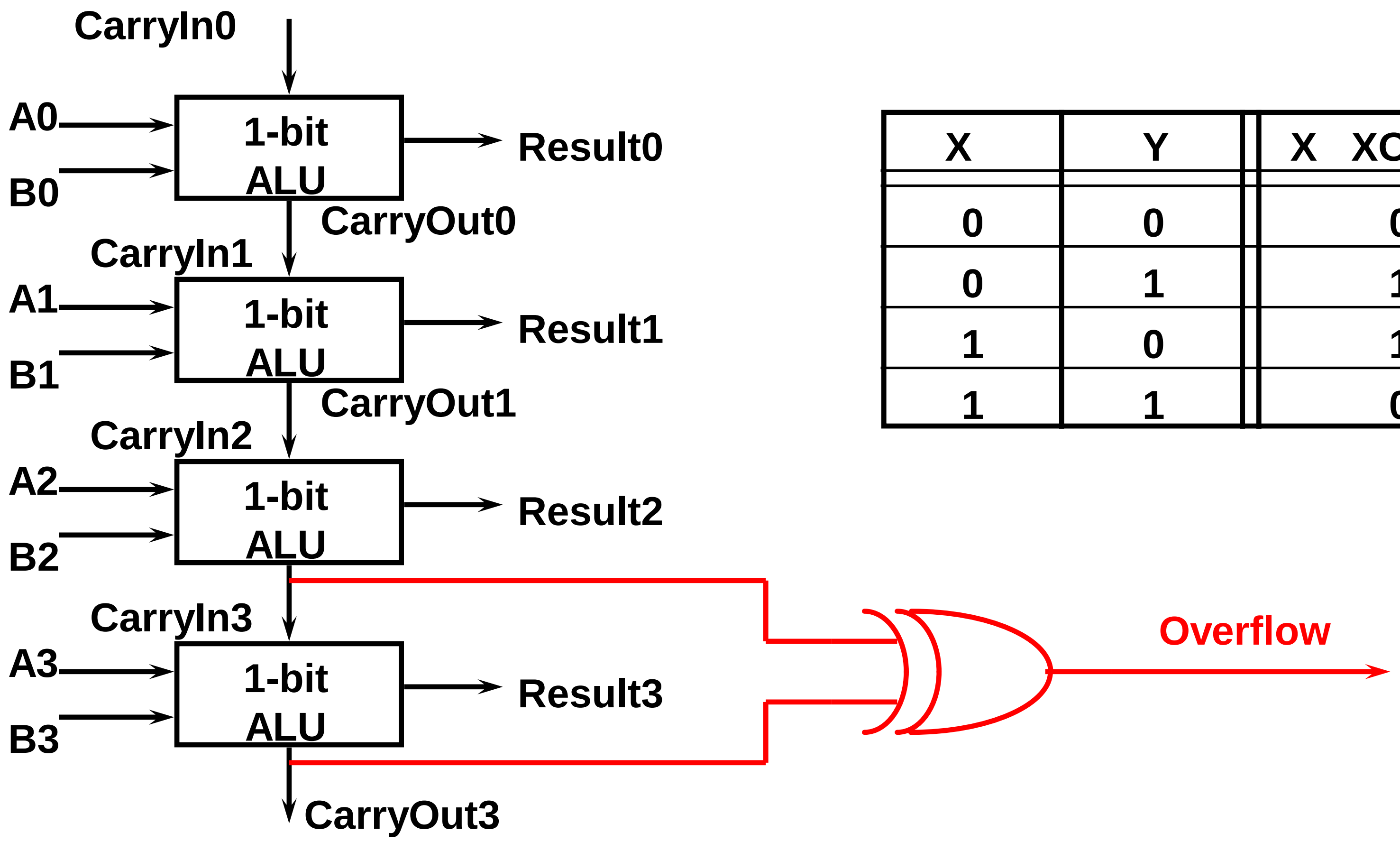
    => sign bit is set with the value of the result
  - Overflow if: <u>Carry into MSB ≠ Carry out of MSB</u>





Overflow Detection Logic

- Overflow = CarryIn[N-1] XOR CarryOut[N-1]



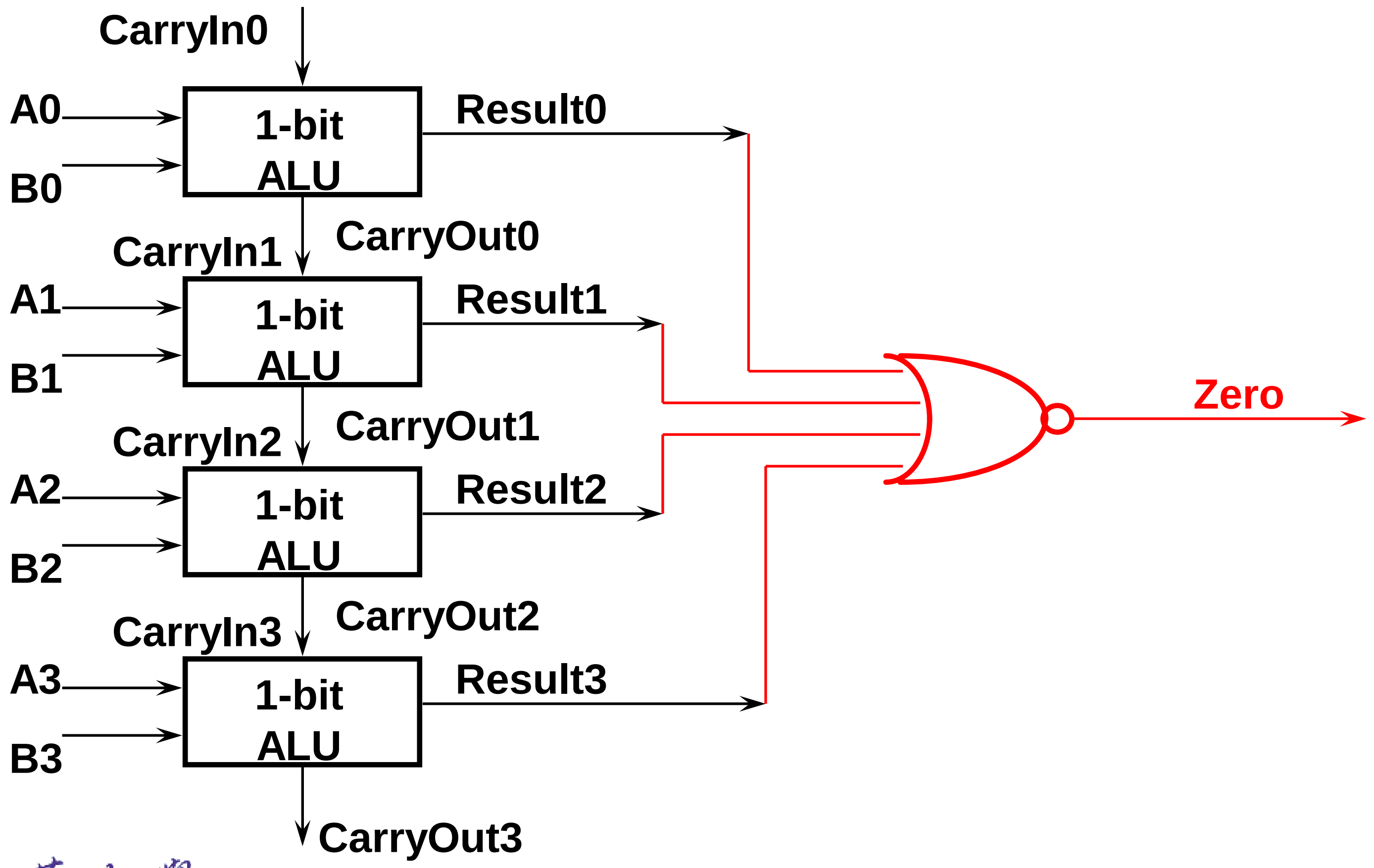
X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

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 (copy) EPC value (to a general purpose register), to return after corrective action (by jump register instruction)

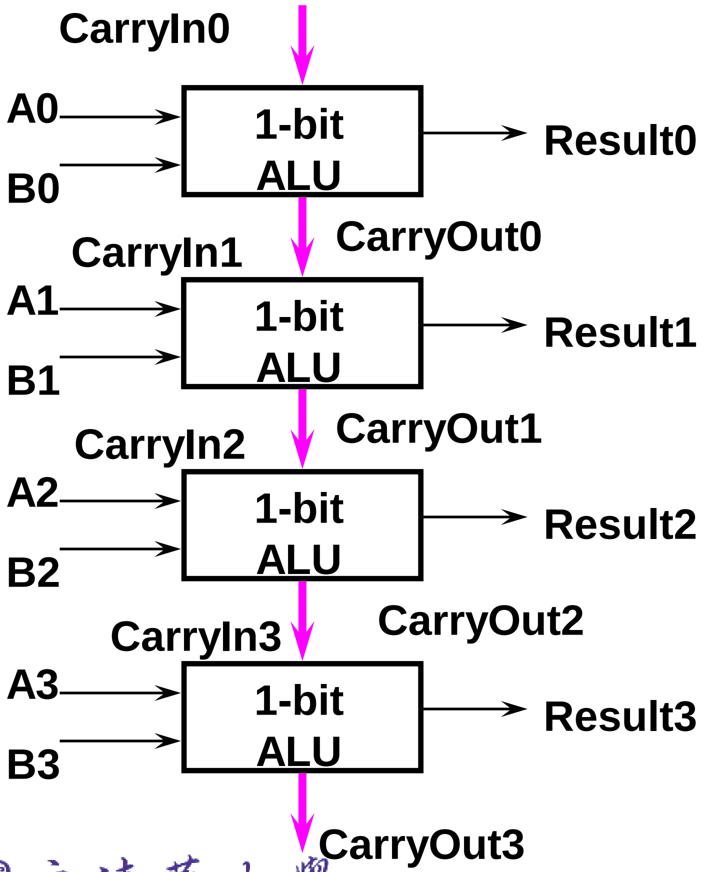
Zero Detection Logic

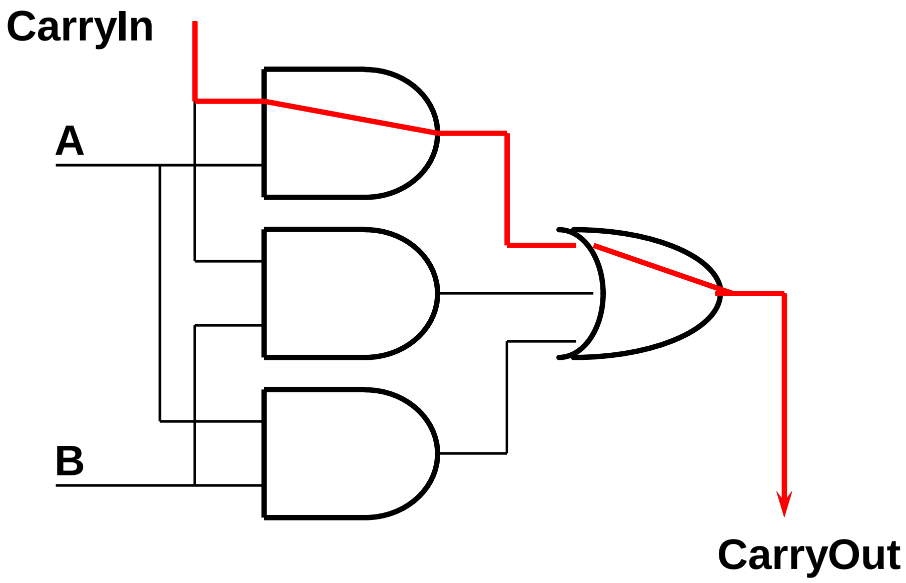
- Zero Detection Logic is a one BIG NOR gate (support conditional jump)



Problems with Ripple Carry Adder

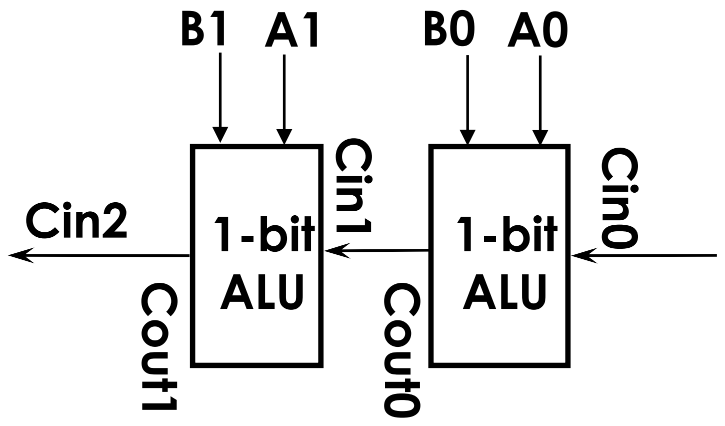
- Carry bit may have to propagate from LSB to MSB => worst case delay: N-stage delay





Design Trick: look for parallelism and throw hardware at it

Carry Lookahead: Theory (I) (Appendix C)

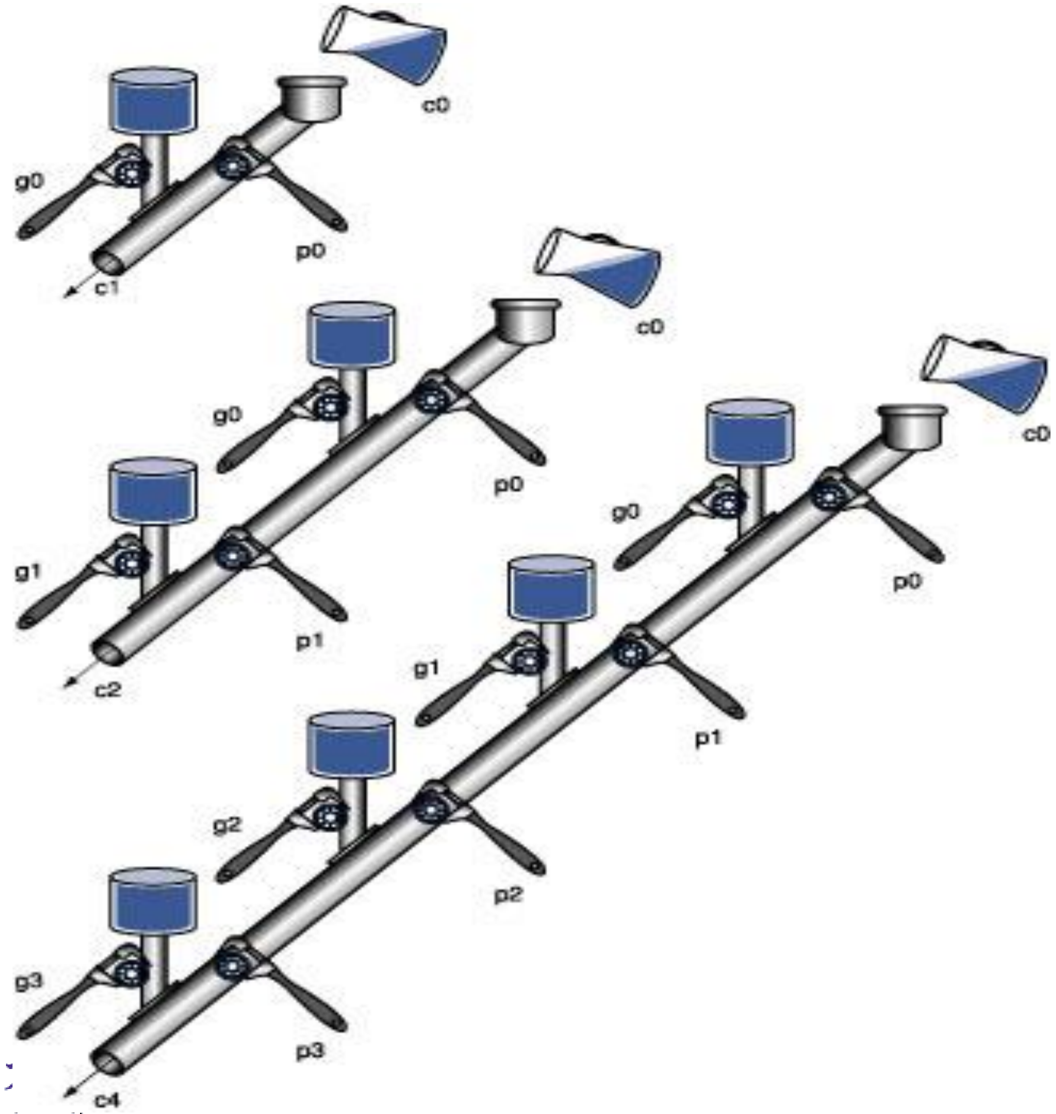


- CarryOut=(B*CarryIn)+(A*CarryIn)+(A*B)
  - Cin2=Cout1= (B1 * Cin1)+(A1 * Cin1)+ (A1 * B1)
  - Cin1=Cout0= (B0 * Cin0)+(A0 * Cin0)+ (A0 * B0)
- Substituting Cin1 into Cin2:
  - Cin2=(A1*A0*B0)+(A1*A0*Cin0)+(A1*B0*Cin0)
    +(B1*A0*B0)+(B1*A0*Cin0)+(B1*B0*Cin0)
    +(A1*B1)

Carry Lookahead: Theory (II)

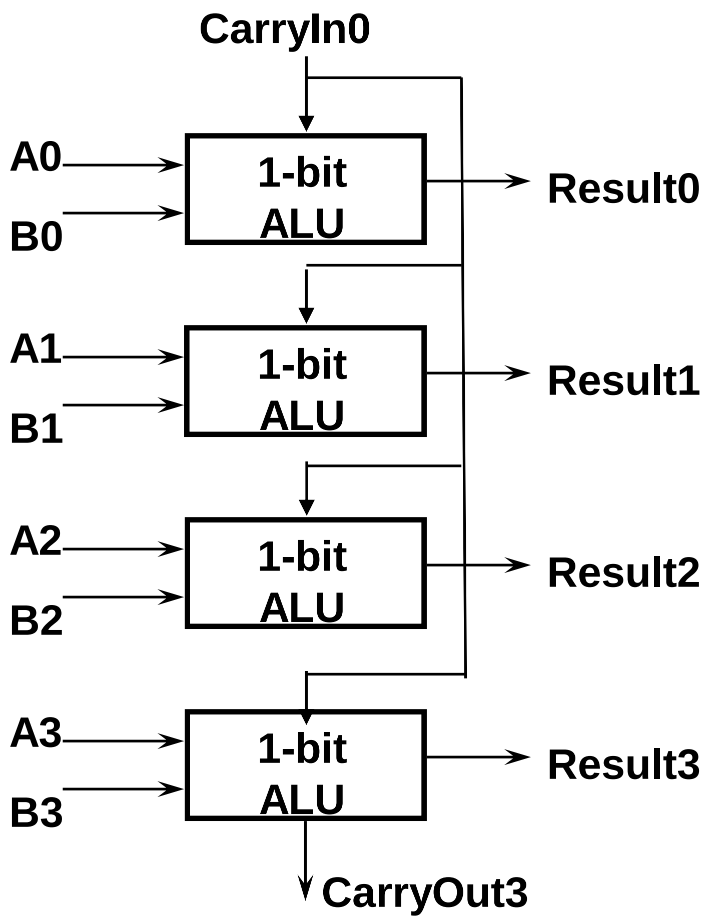
- Now define two new terms:
  - Generate Carry at Bit i: gi = Ai * Bi
  - Propagate Carry via Bit i: pi = Ai xor Bi
- We can rewrite:
  - Cin1=g0+(p0*Cin0)
  - Cin2=g1+(p1*g0)+(p1*p0*Cin0)
  - Cin3=g2+(p2*g1)+(p2*p1*g0)+(p2*p1*p0*Cin0)
- Carry going into bit 3 is 1 if
  - We generate a carry at bit 2 (g2)
  - Or we generate a carry at bit 1 (g1) and bit 2 allows it to propagate (p2 * g1)
  - Or we generate a carry at bit 0 (g0) and bit 1 as well as bit 2 allows it to propagate .....

A Plumbing Analogy for Carry Lookahead (1, 2, 4 bits)



Carry Lookahead Adder

- No Carry bit propagation from LSB to MSB

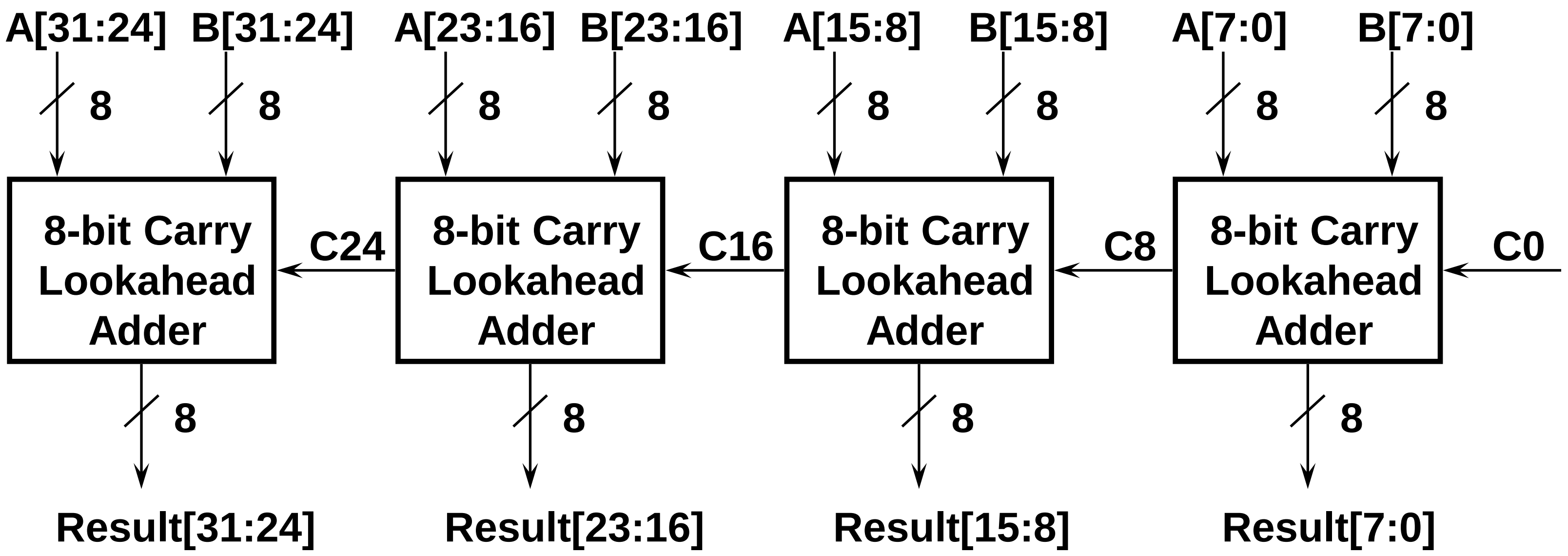


Common Carry Lookahead Adder

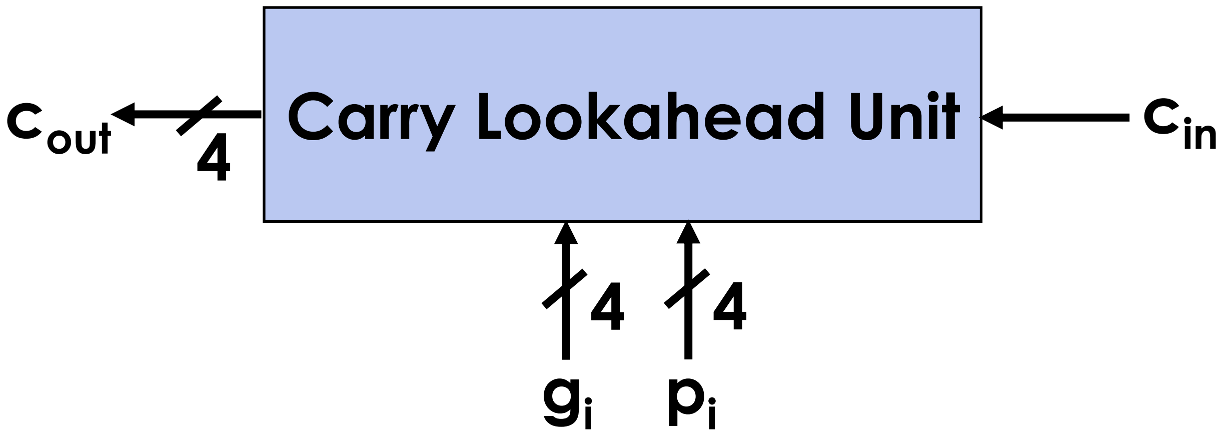
- Expensive to build a "full" carry lookahead adder
  - Just imagine length of the equation for Cin31
- Common practices:
  - Cascaded carry look-ahead adder
  - Multiple level carry look-ahead adder

Cascaded Carry Lookahead

- Connects several N-bit lookahead adders to form a big one

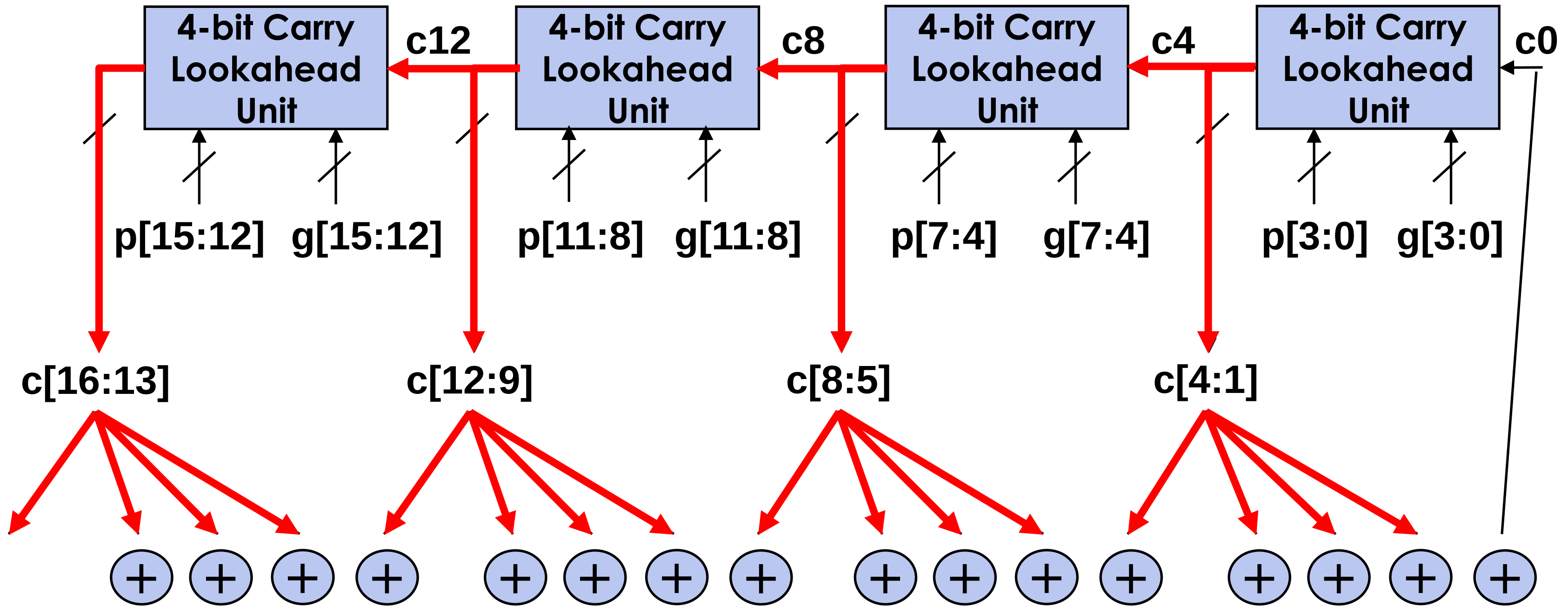


Example: Carry Lookahead Unit



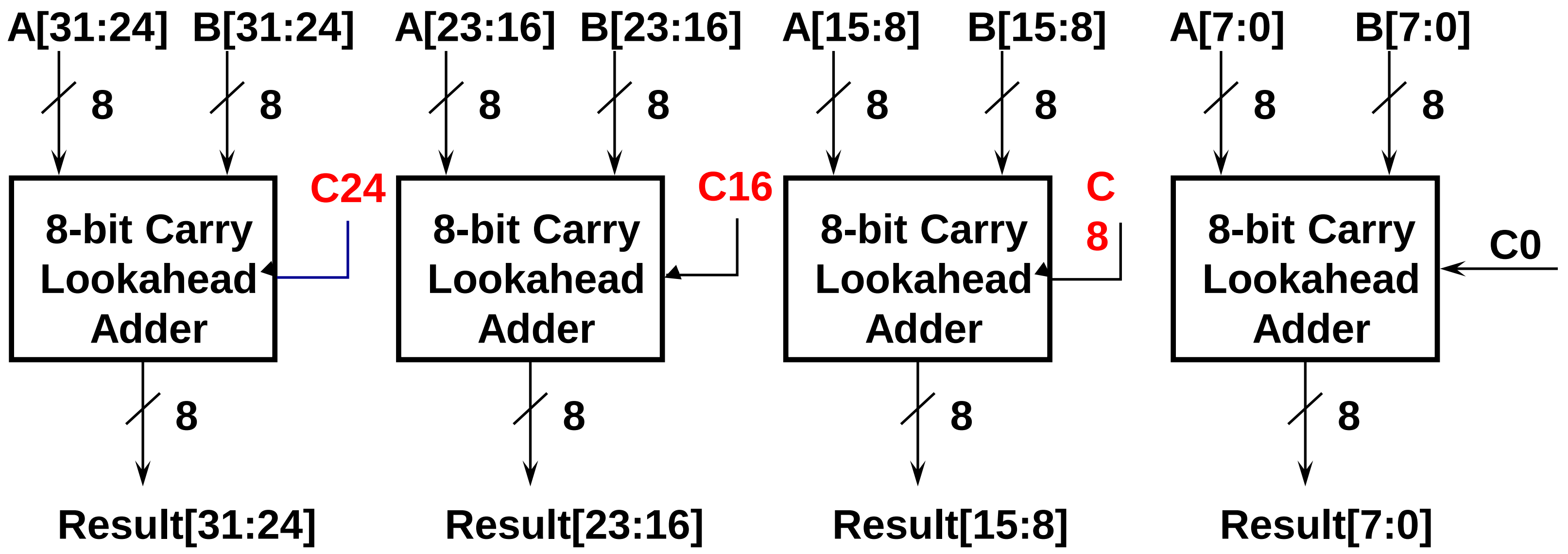
Example: Cascaded Carry Lookahead

- Connects several N-bit lookahead adders to form a big one


4-bit Carry Lookahead Unit
c12
4-bit Carry Lookahead Unit
c8
4-bit Carry Lookahead Unit
c4
4-bit Carry Lookahead Unit
c0
p[15:12]
g[15:12]
p[11:8]
g[11:8]
p[7:4]
g[7:4]
p[3:0]
g[3:0]
c[16:13]
c[12:9]
c[8:5]
c[4:1]
+

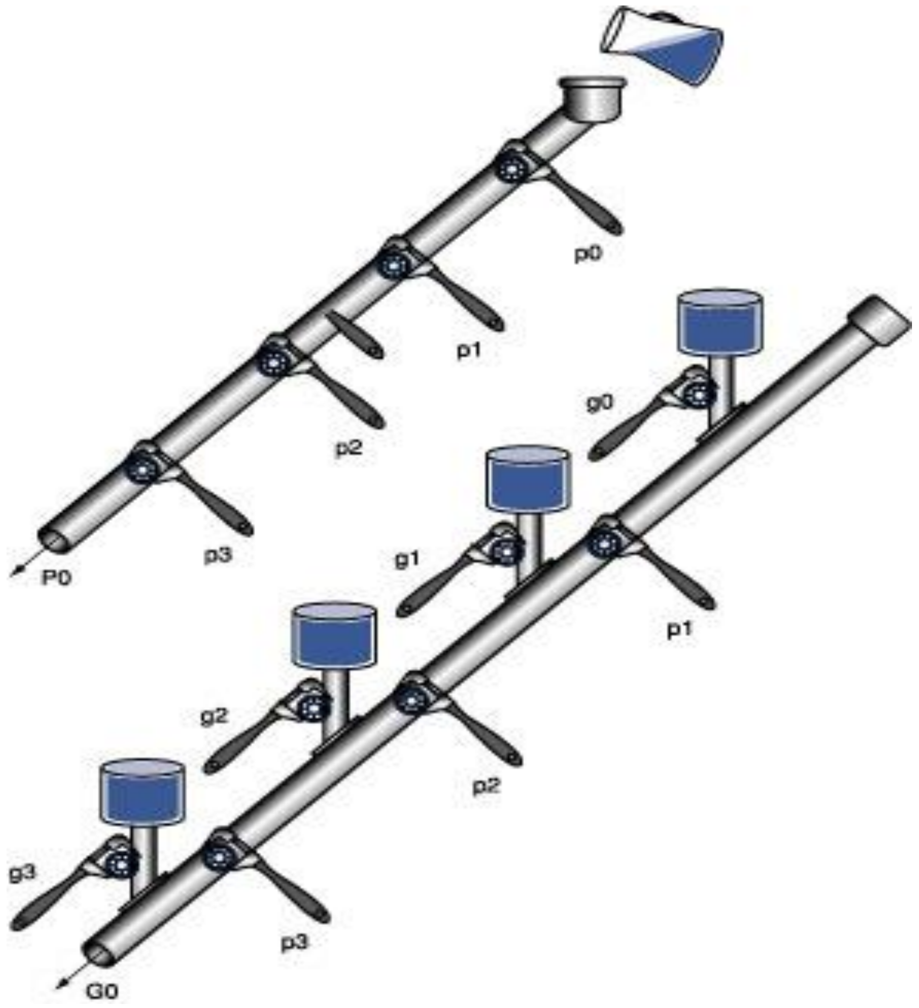
Multiple Level Carry Lookahead

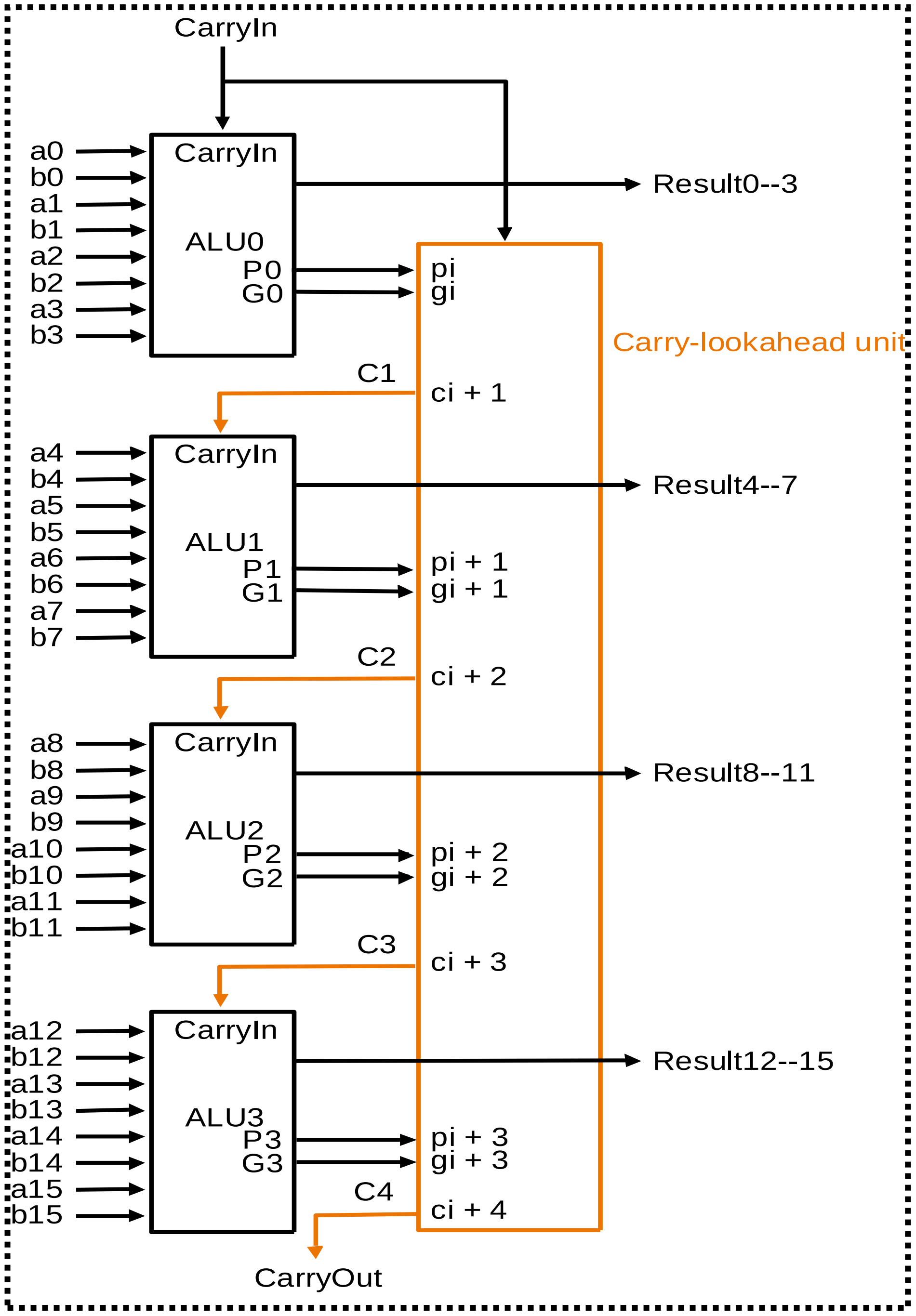
- View an N-bit lookahead adder as a block
- Where to get Cin of the block ?



- Generate "super" Pi and Gi of the block
- Use next level carry lookahead structure to generate block Cin

A Plumbing Analogy for Carry Lookahead (Next Level P0 and G0)





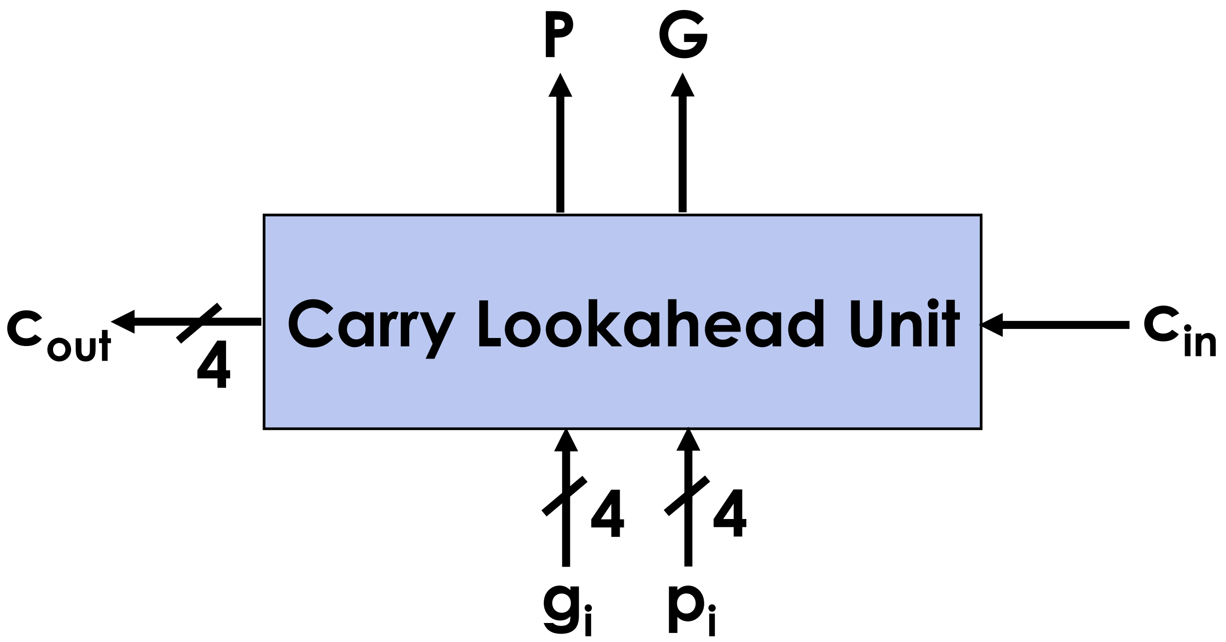
A Carry Lookahead Adder

A	B	Cout	
0	0	0	kill
0	1	Cin	propagate
1	0	Cin	propagate
1	1	1	generate

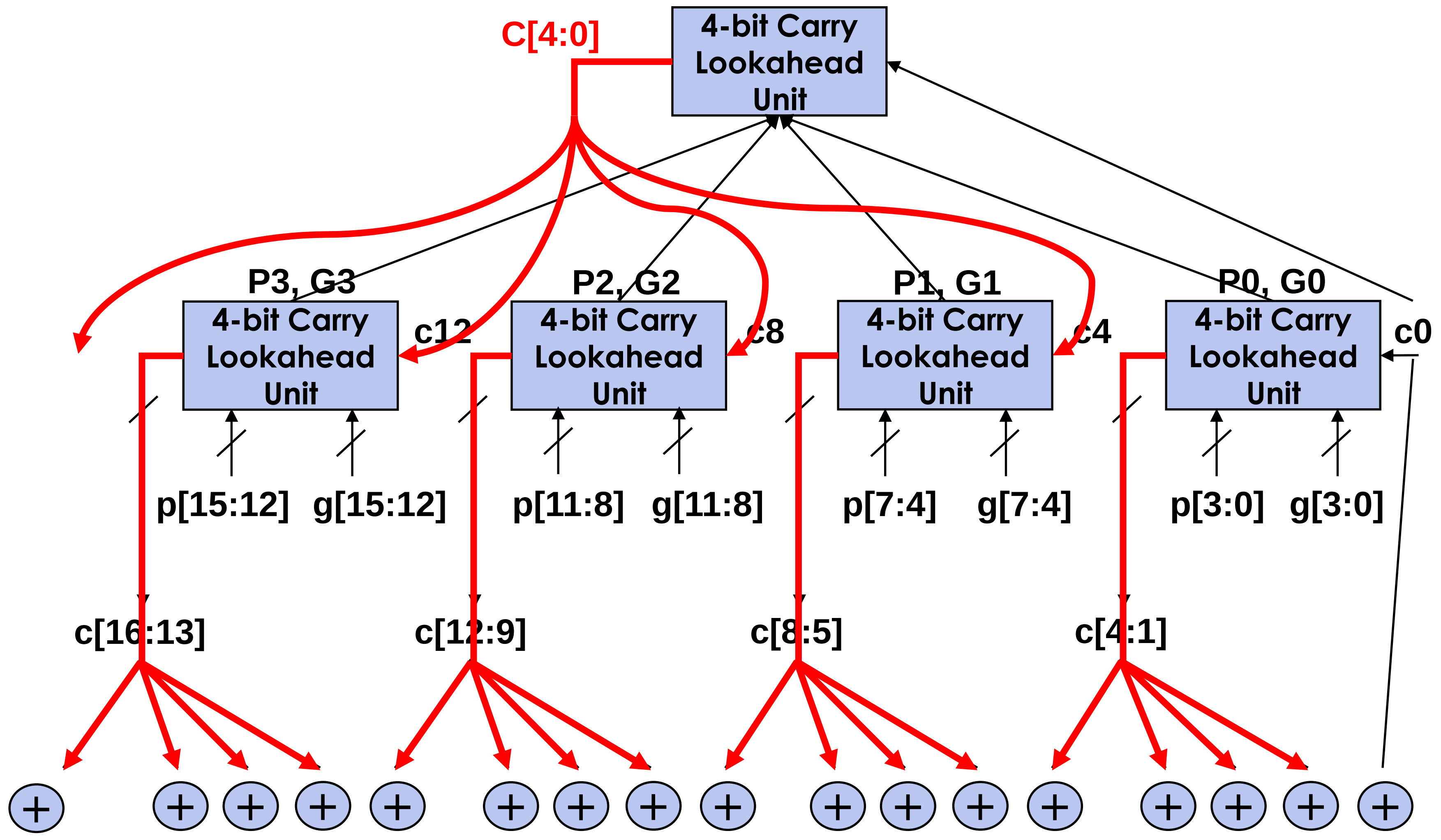
$G = A * B$

$P = A + B$

Example: Carry Lookahead Unit



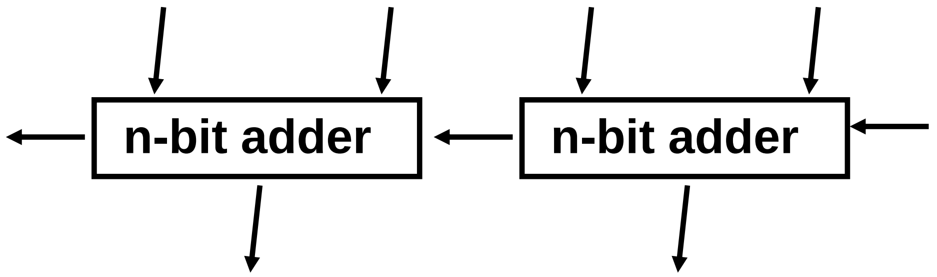
Example: Multiple Level Carry Lookahead



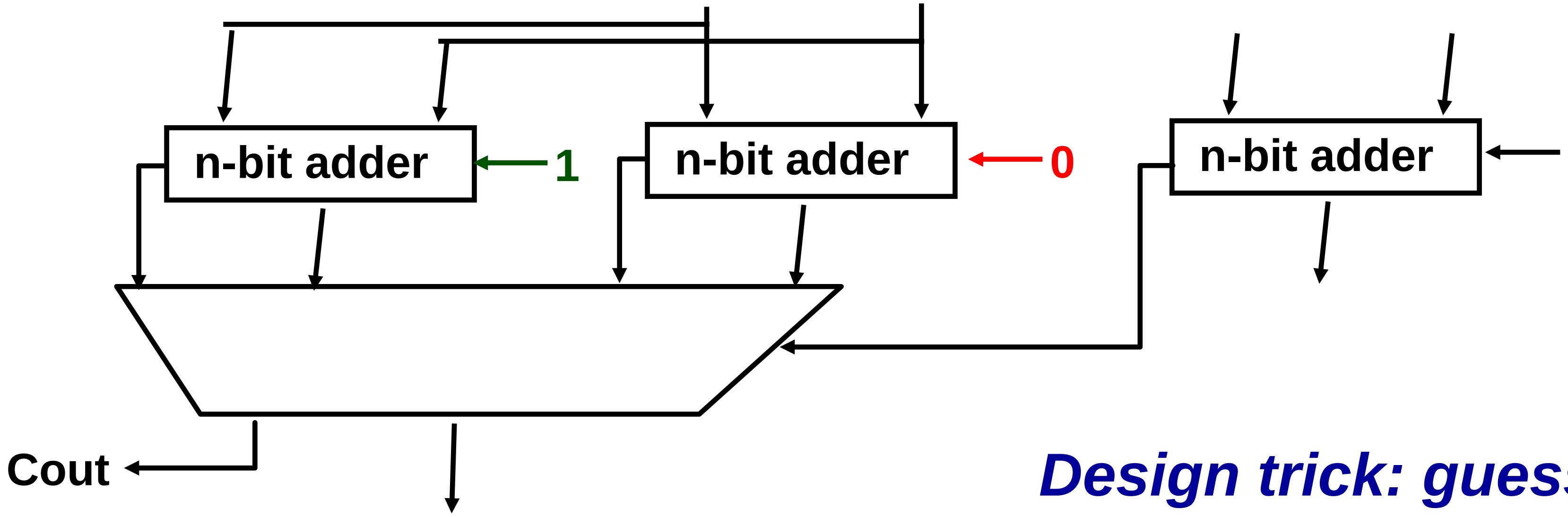
C[4:0]
4-bit Carry Lookahead Unit
P3, G3
P2, G2
P1, G1
P0, G0
4-bit Carry Lookahead Unit
4-bit Carry Lookahead Unit
4-bit Carry Lookahead Unit
4-bit Carry Lookahead Unit
c12
c8
c4
c0
p[15:12]
g[15:12]
p[11:8]
g[11:8]
p[7:4]
g[7:4]
p[3:0]
g[3:0]
c[16:13]
c[12:9]
c[8:5]
c[4:1]

Carry-select Adder

CP(2n) = 2*CP(n)



CP(2n) = CP(n) + CP(mux)



Design trick: guess

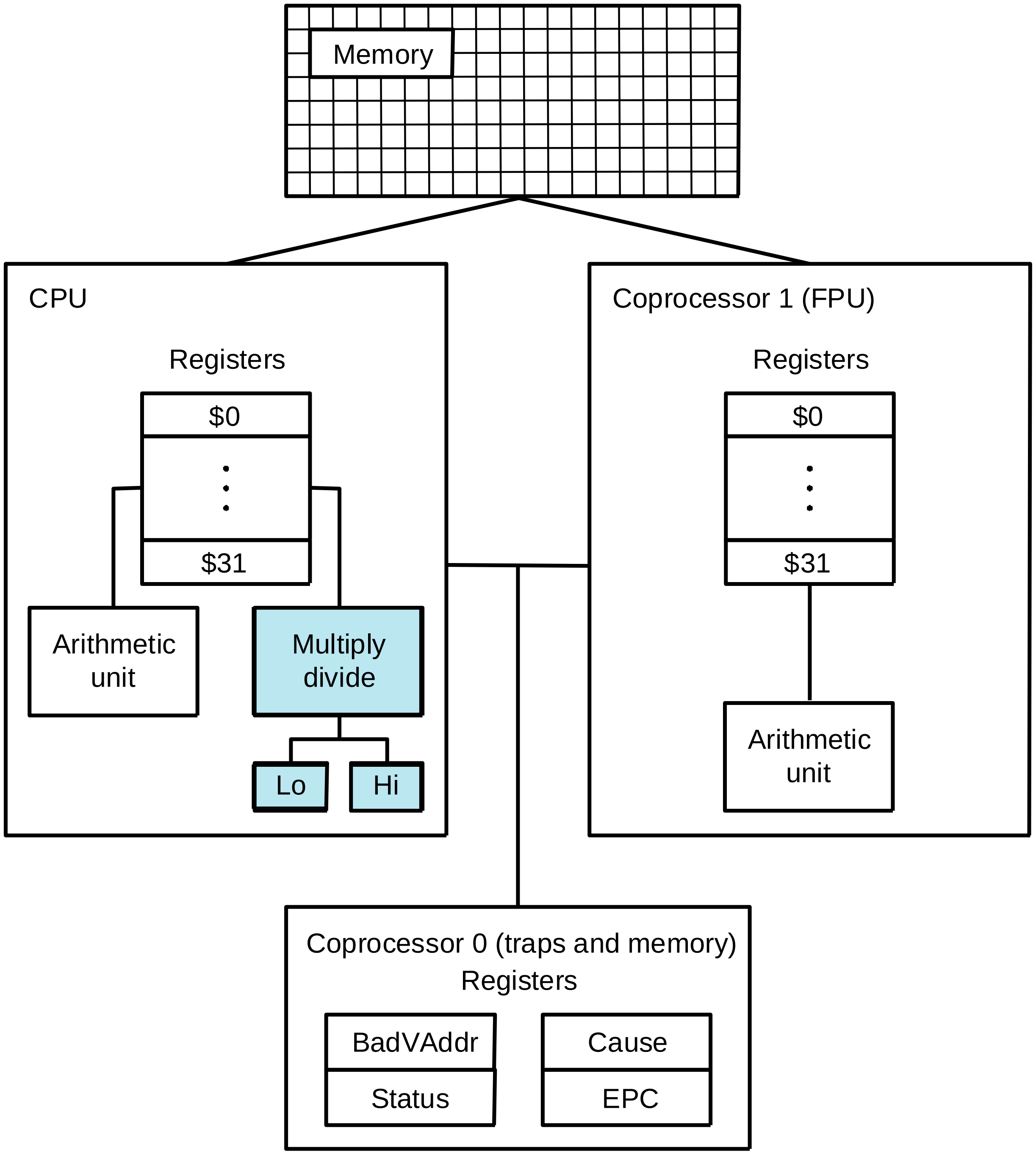
Arithmetic for Multimedia

- 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

Outline

- Addition and subtraction (Sec. 3.2)
- Constructing an arithmetic logic unit (Appendix C)
- Multiplication (Sec. 3.3, Appendix C)
- Division (Sec. 3.4)
- Floating point (Sec. 3.5)

MIPS R2000 Organization



Multiplication in MIPS

```
mult $t1, $t2          # t1 * t2
```

- No destination register: product could be ~2^{64}; need two special registers to hold it
- 3-step process:

```
     $t1   01111111111111111111111111111111
   X $t2   01000000000000000000000000000000
--------------------------------------------------------------------
00011111111111111111111111111111   11|000000000000000000000000000000
              Hi                                Lo

mfhi $t3        $t3   00011111111111111111111111111111

mflo $t4        $t4   11000000000000000000000000000000
```

MIPS Multiplication

- Two 32-bit registers for product
  - HI: most-significant 32 bits
  - LO: least-significant 32-bits
- Instructions
  - mult rs, rt / multu rs, rt
    - 64-bit product in HI/LO
  - mfhi rd / mflo rd
    - Move from HI/LO to rd
    - Can test HI value to see if product overflows 32 bits
  - mul rd, rs, rt
    - Least-significant 32 bits of product –> rd

Unsigned Multiply

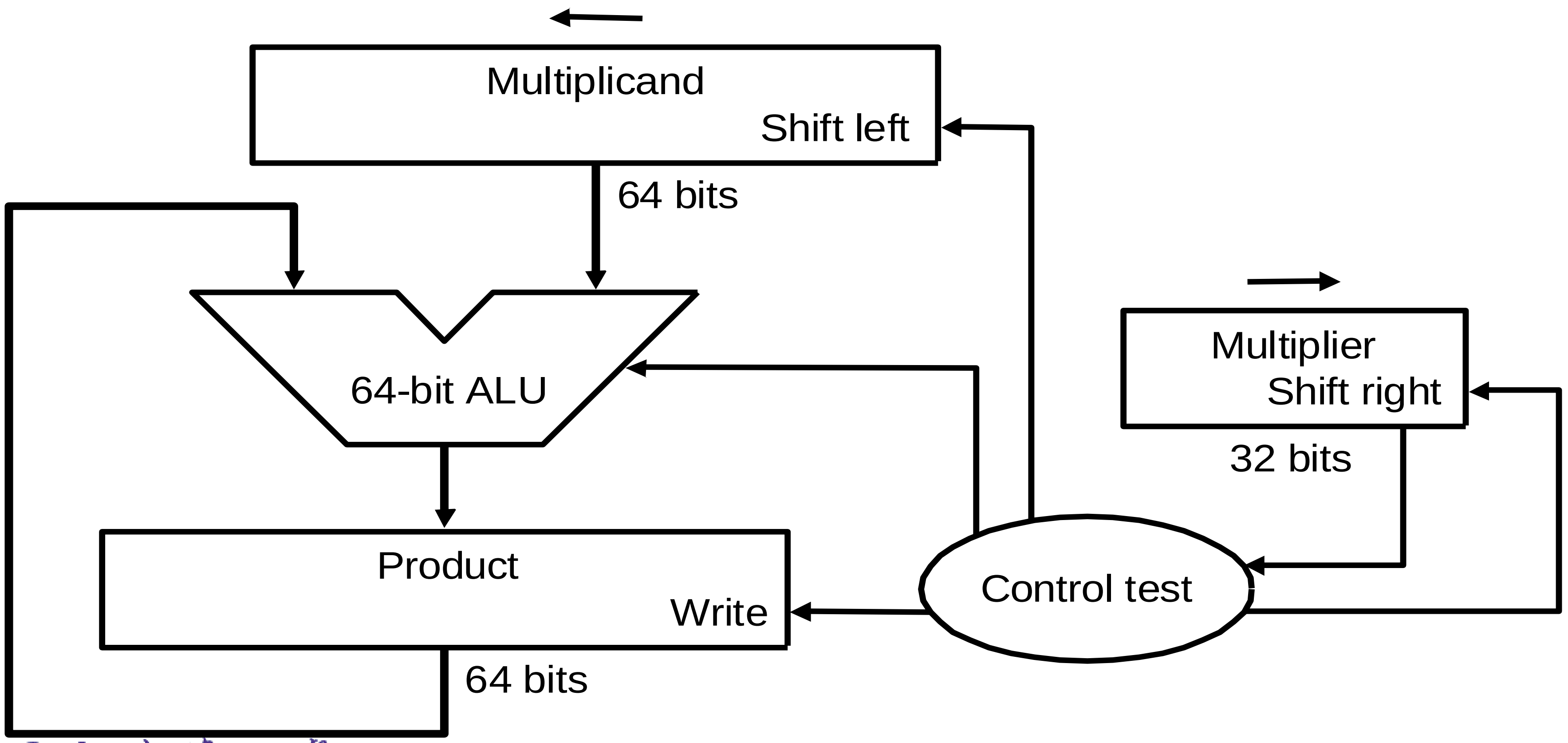
- Paper and pencil example (unsigned):

$$
\begin{array}{lr}
\text{Multiplicand} & 1000_{ten} \\
\text{Multiplier} & \text{X} \quad 1001_{ten} \\
\hline
 & 1000 \\
 & 0000 \\
 & 0000 \\
 & 1000 \\
\hline
\text{Product} & 01001000_{ten}
\end{array}
$$

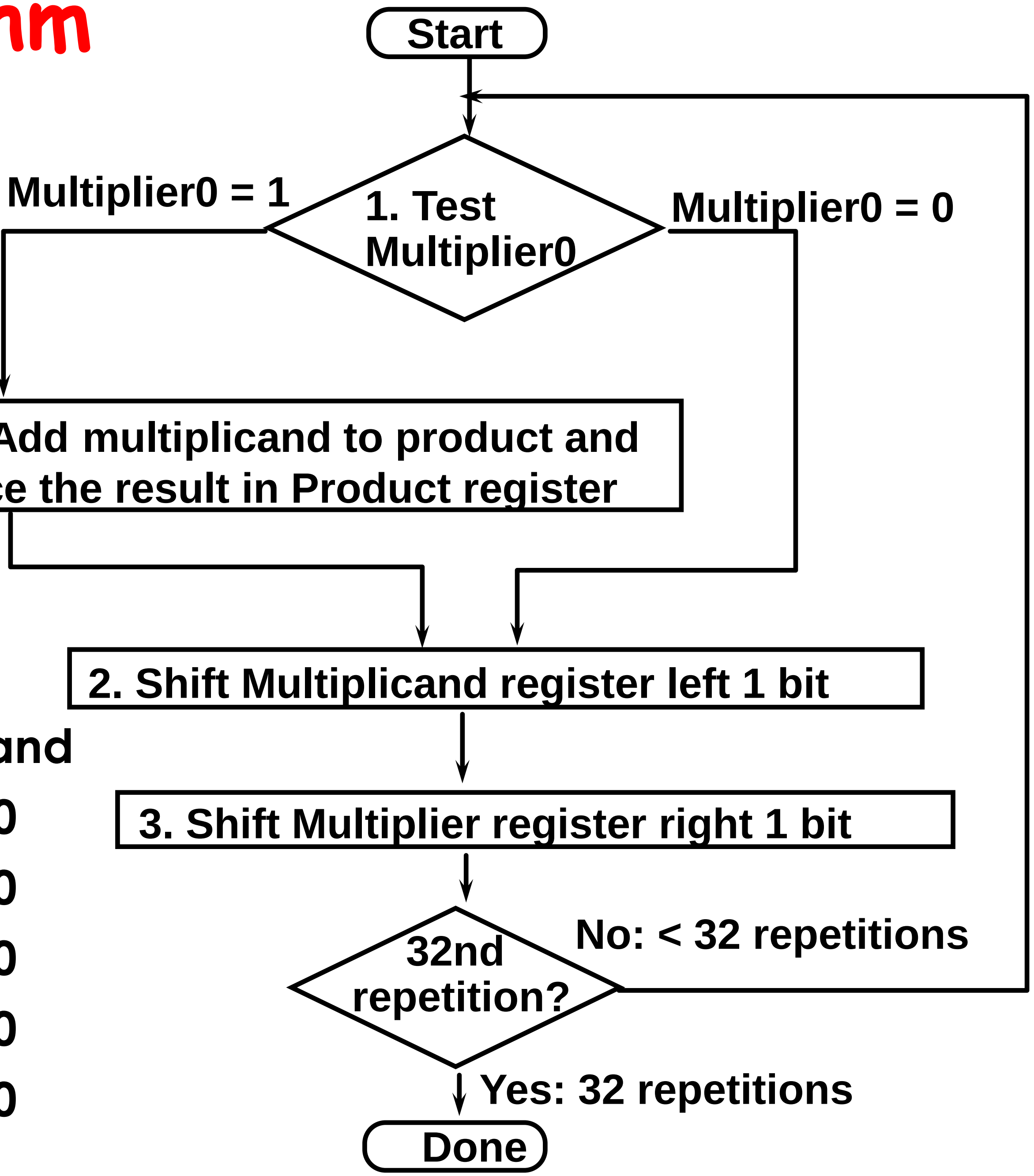
- m bits x n bits = m+n bit product
- Binary makes it easy:
  - 0 => place 0 (0 x multiplicand)
  - 1 => place a copy (1 x multiplicand)
- 2 versions of multiply hardware and algorithm

Unsigned Multiplier (Ver. 1)

- 64-bit *multiplicand register* (with 32-bit multiplicand at right half), 64-bit ALU, 64-bit *product register*, 32-bit *multiplier register*



Multiply Algorithm (Ver. 1)



Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift Multiplicand register left 1 bit
3. Shift Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

0010 x 0011

Product	Multiplier	Multiplicand
0000 0000	0011	0000 0010
0000 0010	0001	0000 0100
0000 0110	0000	0000 1000
0000 0110	0000	0001 0000
0000 0110	0000	0010 0000

Observations: Multiply Ver. 1

- 1 clock per cycle => ~100 clocks per multiply
  - Ratio of multiply to add 5:1 to 100:1
- Half of the bits in multiplicand always 0
  => 64-bit adder is wasted
- 0's inserted in right of multiplicand as shifted
  => least significant bits of product never changed once formed
- Instead of shifting multiplicand to left, shift product to right?
- Product register wastes space => combine Multiplier and Product register

Unsigned Multiply

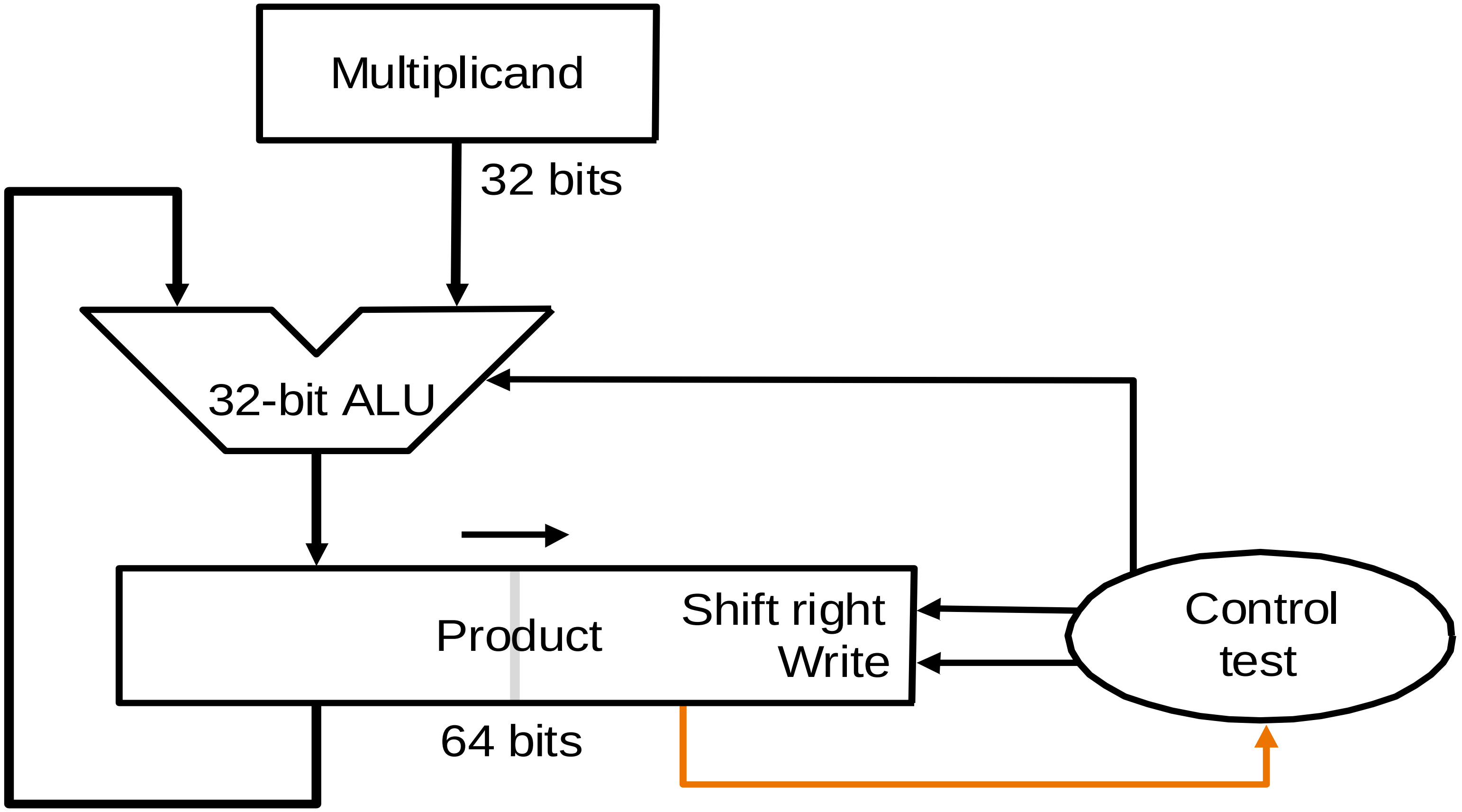
- Paper and pencil example (unsigned):

$$\begin{array}{l|cccccccc|l}
\text{Multiplicand} & & & & & 1 & 0 & 0 & 0 & _{ten} \\
\text{Multiplier} & & & & \times & 1 & 0 & 0 & 1 & _{ten} \\
\hline
 & & & & & 1 & 0 & 0 & 0 & \\
 & & & & 0 & 0 & 0 & 0 & & \\
 & & & 0 & 0 & 0 & 0 & & & \\
 & & 1 & 0 & 0 & 0 & & & & \\
\hline
\text{Product} & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & _{ten}
\end{array}$$

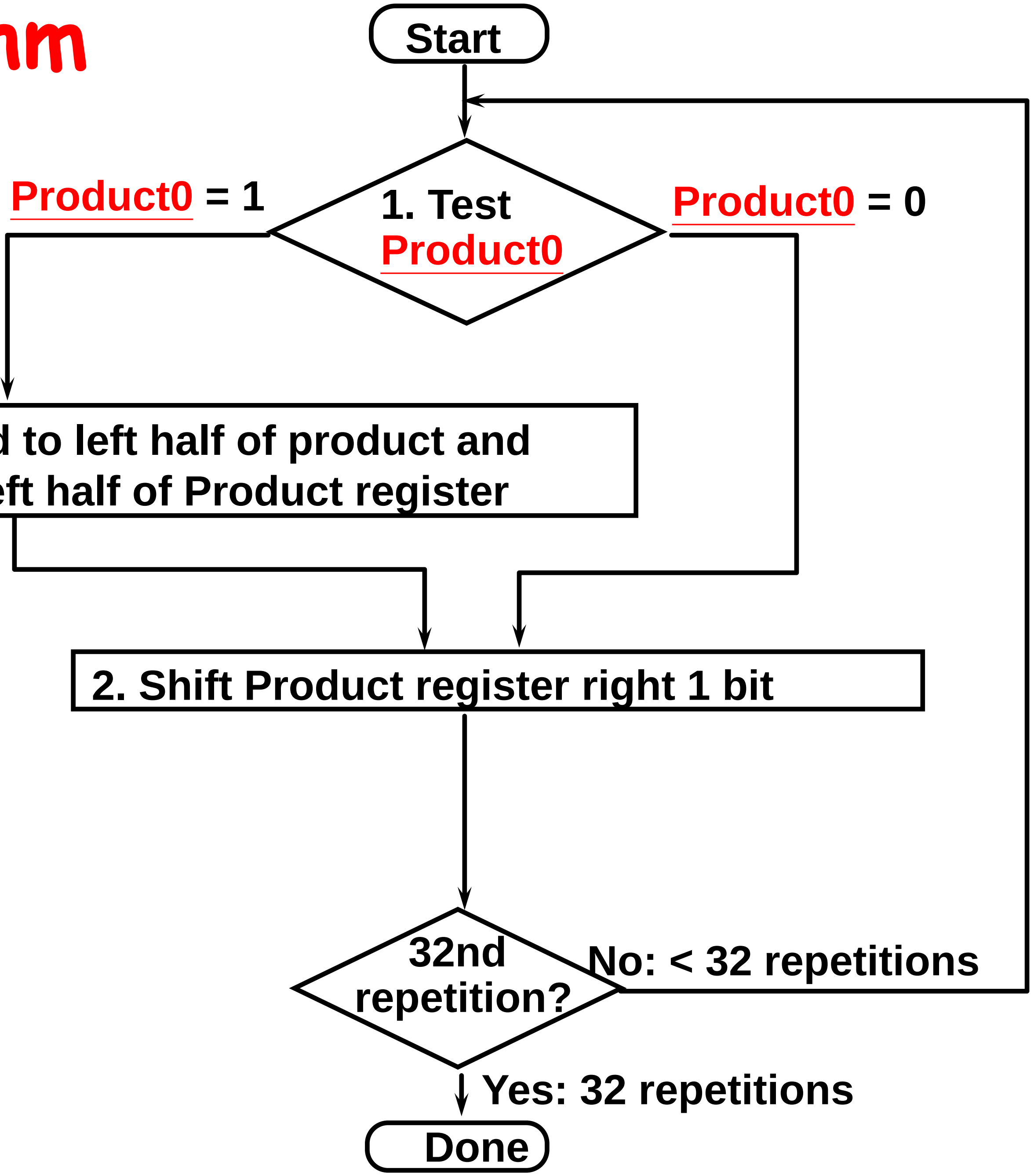
- m bits x n bits = m+n bit product
- Binary makes it easy:
  - 0 => place 0 (0 x multiplicand)
  - 1 => place a copy (1 x multiplicand)
- 2 versions of multiply hardware and algorithm

Unisigned Multiplier (Ver. 2)

- 32-bit Multiplicand register, 32 -bit ALU, 64-bit Product register (HI & LO in MIPS), (0-bit Multiplier register)



Multiply Algorithm (Ver. 2)


Start
1. Test Product0
Product0 = 1
Product0 = 0
1a. Add multiplicand to left half of product and place the result in left half of Product register
2. Shift Product register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

Multiplicand	Product
0010	0000 0011
	0010 0011
0010	0001 0001
	0011 0001
0010	0001 1000
0010	0000 1100
0010	0000 0110

Observations: Multiply Ver. 2

- 2 steps per bit because multiplier and product registers combined
- MIPS registers Hi and Lo are left and right half of Product register
  => this gives the MIPS instruction MultU
- What about signed multiplication?
  - The easiest solution is to make both positive and remember whether to complement product when done (leave out sign bit, run for 31 steps)
  - Apply definition of 2's complement
    - sign-extend partial products and subtract at end
  - Booth's Algorithm is an elegant way to multiply signed numbers using same hardware as before and save cycles

Signed Multiply

- Paper and pencil example (signed):

```
Multiplicand              1001 (-7)
Multiplier             X  1001 (-7)
                       11111001
            +          0000000
            +          000000
            -          11001
Product                00110001 (49)
```

- Rule 1: Multiplicand sign extended
- Rule 2: Sign bit (s) of Multiplier
  - 0 => 0 x multiplicand
  - 1 => -1 x multiplicand
- Why rule 2 ?
  - $X = s\ x_{n-2}\ x_{n-3....}\ x_1\ x_0$ (2's complement)
  - $\text{Value}(X) = -1 \times s \times 2^{n-1} + x_{n-2} \times 2^{n-2} + \ldots\ldots + x_0 \times 2^0$

```
      00100000
-     00000001
-------------------
      00011111
```

Booth's Algorithm: Motivation

- Example: 2 x 6 = 0010 x 0110:

```
              0010two
x             0110two
+             0000   shift (0 in multiplier)
+            0010    add (1 in multiplier)
+           0010     add (1 in multiplier)
+          0000      shift (0 in multiplier)
           0001100two
```

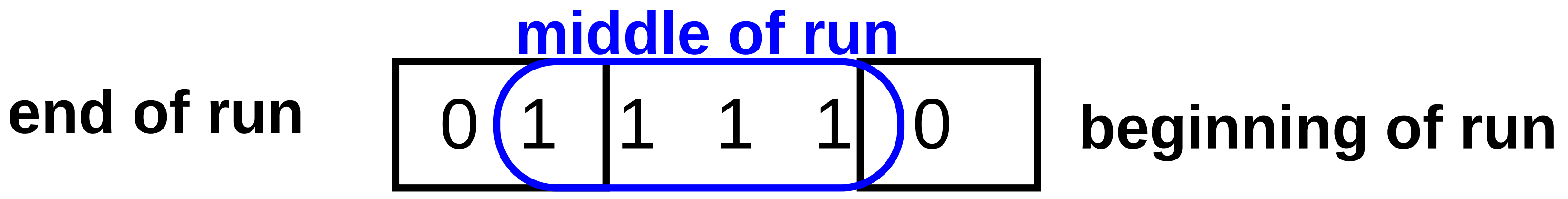
- Can get same result in more than one way:

```
    6 = -2 + 8        0110 = -00010 + 01000
```

- Basic idea: replace a string of 1s with an initial subtract on seeing a one and add after last one

```
              0010two
x             0110two
              0000   shift (0 in multiplier)
-            0010    sub (first 1 in multiplier)
            0000     shift (mid string of 1s)
+          0010      add (prior step had last 1)
          00001100two
```

Booth's Algorithm: Rationale



Current bit	Bit to right	Explanation	Example	Op
1	0	Begins run of 1s	0000111**10**00	sub
1	1	Middle run of 1s	00001**11**1000	none
0	1	End of run of 1s	000**01**111000	add
0	0	Middle run of 0s	0**00**01111000	none

Originally for speed (when shift was faster than add)

- Why it works?

$$\begin{array}{r} -1 \\ +\ 10000 \\ \hline 01111 \end{array}$$

Booth's Algorithm

1. Depending on the current and previous bits, do one of the following:
   00: Middle of a string of 0s, no arithmetic op.
   01: End of a string of 1s, so add multiplicand to the left half of the product
   10: Beginning of a string of 1s, so subtract multiplicand from the left half of the product
   11: Middle of a string of 1s, so no arithmetic op.
2. As in the previous algorithm, shift the Product register right (arithmetically) 1 bit

Booths Example (2 x 7)

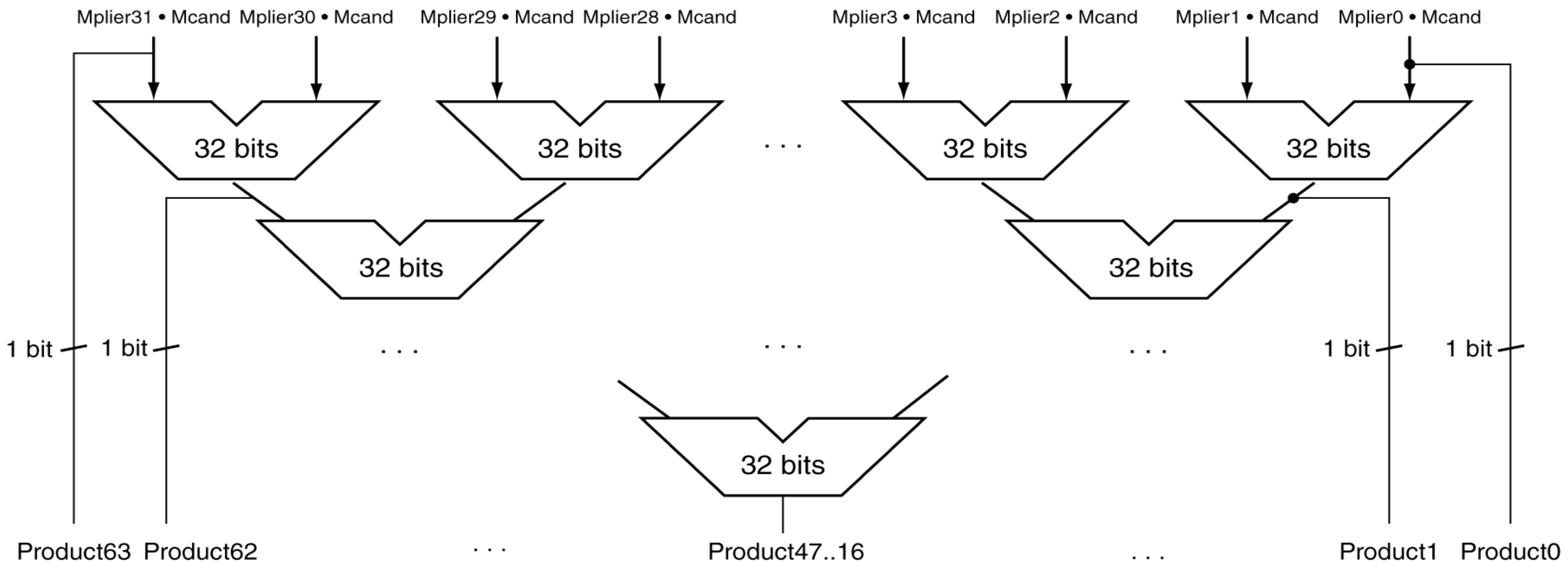
Operation	Multiplicand	Product	next?
0. initial value	0010	0000 0111 0	10 -> sub
1a. P = P - m	1110	+1110	
		1110 0111 0	shift P (sign ext)
1b.	0010	$\underline{1}$111 0011 1	11 -> nop, shift
2.	0010	1111 1001 1	11 -> nop, shift
3.	0010	1111 1100 1	01 -> add
4a.	0010	+0010	
		0001 1100 1	shift
4b.	0010	0000 1110 0	done

Booths Example (2 x -3)

Operation	Multiplicand	Product	next?
0. initial value	0010	0000 1101 0	10 -> sub
1a. P = P - m	1110	+1110	
		1110 1101 0	shift P (sign ext)
1b.	0010	$\underline{1}$111 0110 1	01 -> add
	0010	+0010	
2a.		0001 0110 1	shift P
2b.	0010	0000 1011 0	10 -> sub
	1110	+1110	
3a.	0010	1110 1011 0	shift
3b.	0010	1111 0101 1	11 -> nop
4a		1111 0101 1	shift
4b.	0010	1111 1010 1	done

Faster Multiplier

- A combinational multiplier
- Use multiple adders
  - Cost/performance tradeoff



- Can be pipelined
  - Several multiplication performed in parallel

Wallace Tree Multiplier

- Use carry save adders: three inputs and two outputs

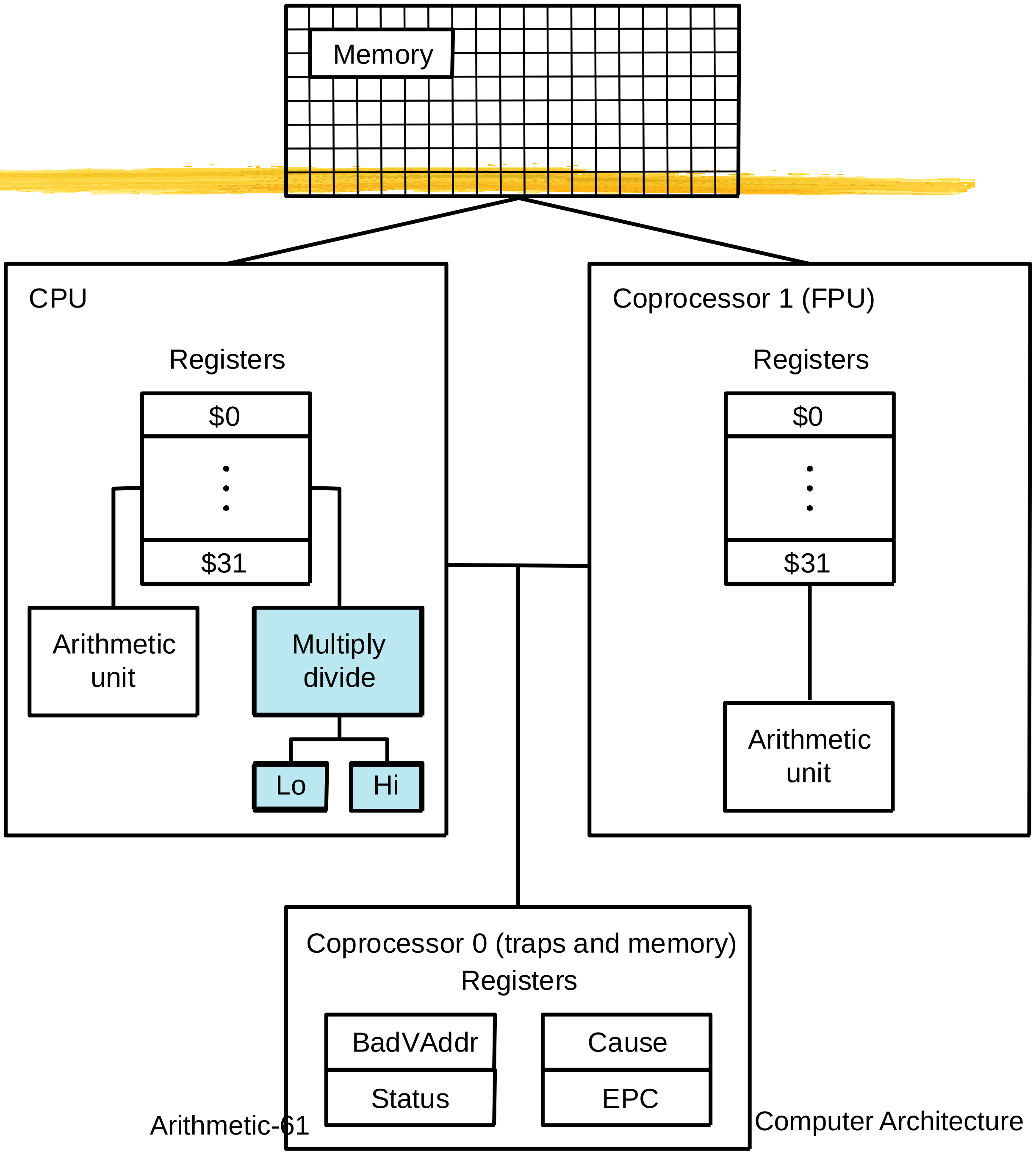
```
   1 0 1 0 1 1 1 0
   0 0 1 0 0 0 1 1
   1 0 0 0 0 1 1 1
   ----------------
   0 0 0 0 1 0 1 0 (sum)
 1 0 1 0 0 1 1 1    (carry)
```

- 8 full adders
- One full adder delay (no carry propagation)
- The last stage is performed by regular adder
- What is the minimum delay for 16 x 16 multiplier ?

Outline

- Addition and subtraction (Sec. 3.2)
- Constructing an arithmetic logic unit (Appendix C)
- Multiplication (Sec. 3.3, Appendix C)
- Division (Sec. 3.4)
- Floating point (Sec. 3.5)

MIPS R2000 Organization



Division in MIPS

```
div $t1, $t2      # t1 / t2
```

- Quotient stored in Lo, remainder in Hi

```
mflo $t3      #copy quotient to t3
mfhi $t4      #copy remainder to t4
```

- 3-step process

- Unsigned division:

```
divu  $t1, $t2      # t1 / t2
```

  - Just like div, except now interpret t1, t2 as unsigned integers instead of signed
  - Answers are also unsigned, use mfhi, mflo to access
- No overflow or divide-by-0 checking
  - Software must perform checks if required

Divide: Paper & Pencil

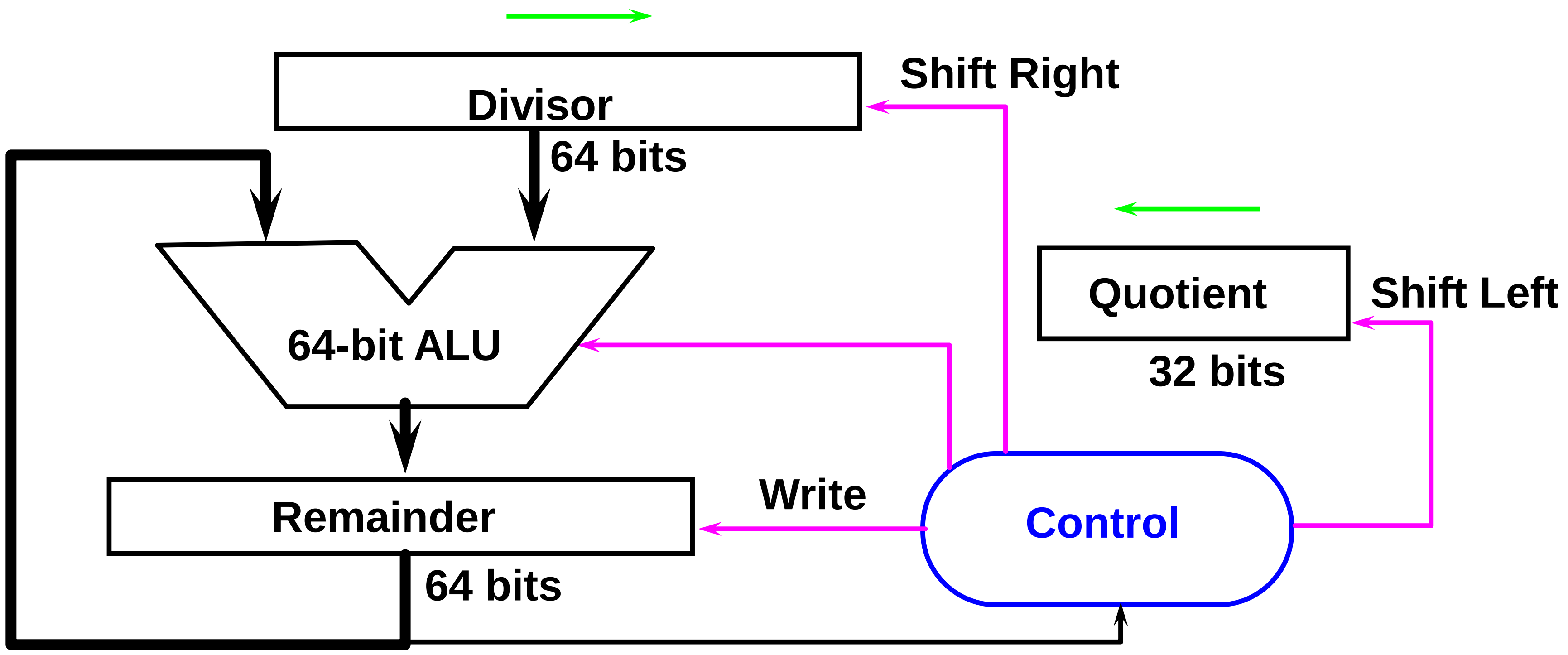
```
                           1001ten          Quotient
Divisor 1000ten     |1001010ten          Dividend
                   -1000
                     0010
                      0101
                       1010
                      -1000
                         10ten          Remainder
```

- See how big a number can be subtracted, creating quotient bit on each step

  Binary => 1 * divisor or 0 * divisor
- Two versions of divide, successive refinement
- Both dividend and divisor are 32-bit positive integers

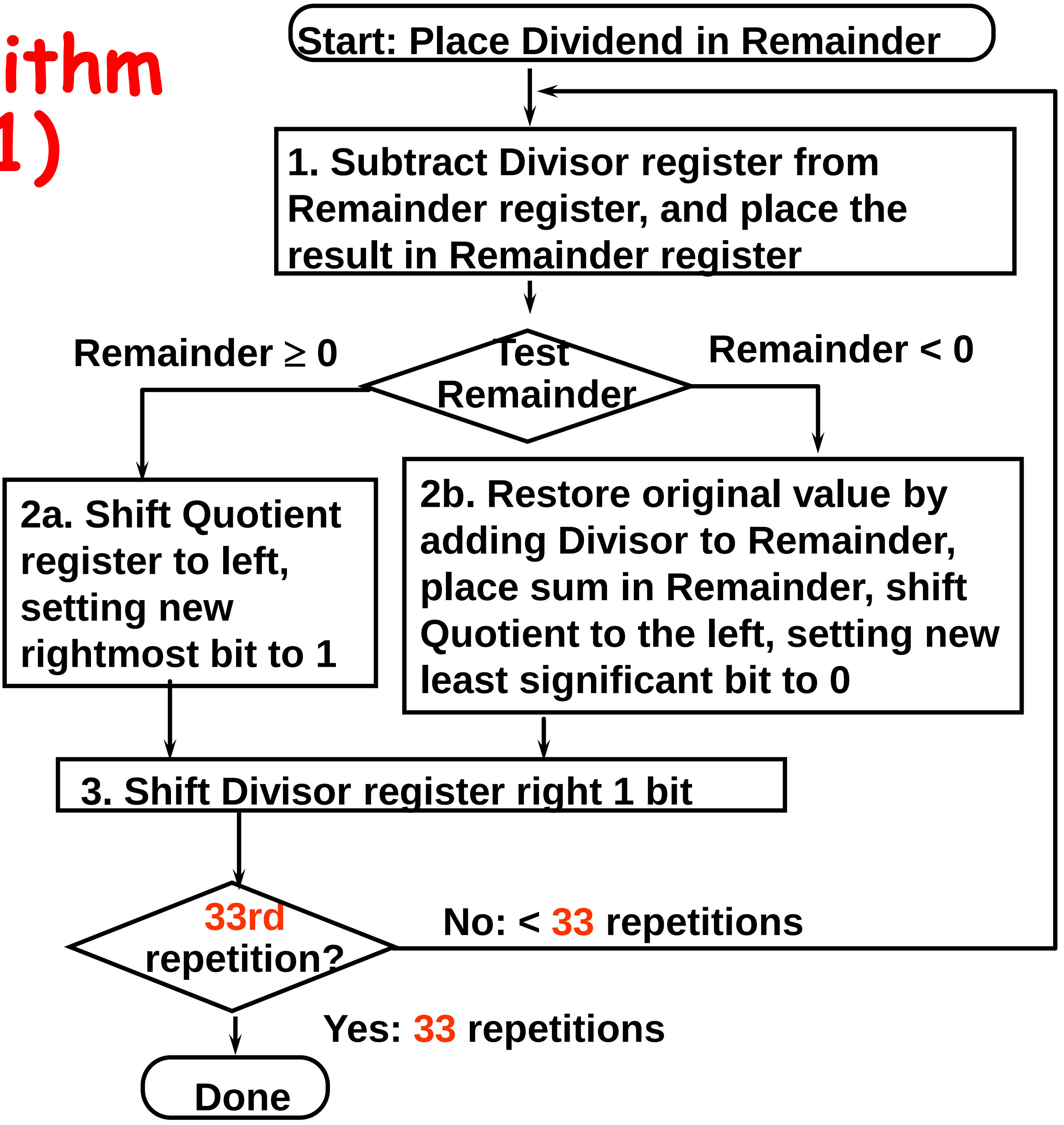
Divide Hardware (Version 1)

- 64-bit *Divisor register* (initialized with 32-bit divisor in left half), 64-bit ALU, 64-bit *Remainder register* (initialized with 64-bit dividend), 32-bit *Quotient register*



Divide Algorithm (Version 1)

Quot.	Divisor	Rem.
0000	00100000	00000111
		11100111
		00000111
0000	00010000	00000111
		11110111
		00000111
0000	00001000	00000111
		11111111
		00000111
0000	00000100	00000111
		00000011
0001		00000011
0001	00000010	00000011
		00000001
0011		00000001
0011	00000001	00000001



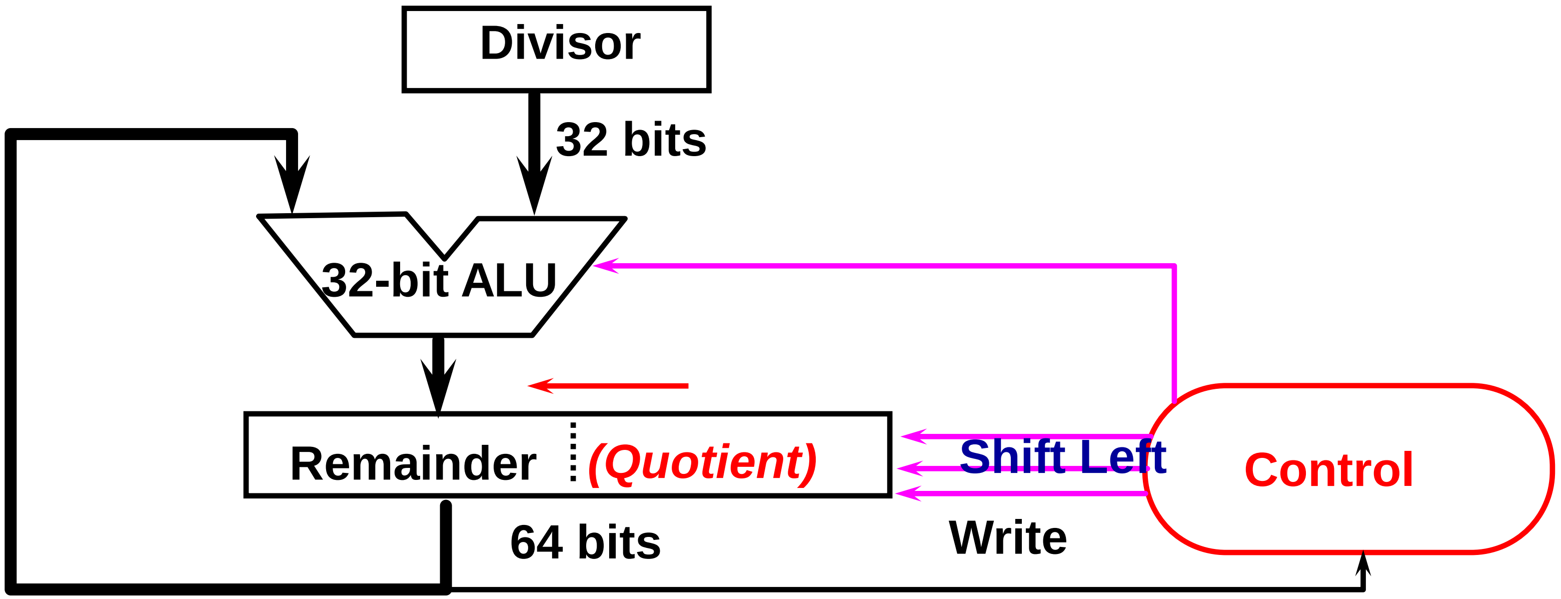
Start: Place Dividend in Remainder
1. Subtract Divisor register from Remainder register, and place the result in Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift Quotient register to left, setting new rightmost bit to 1
2b. Restore original value by adding Divisor to Remainder, place sum in Remainder, shift Quotient to the left, setting new least significant bit to 0
3. Shift Divisor register right 1 bit
33rd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done

Observations: Divide Version 1

- Half of the bits in divisor register always 0
  => 1/2 of 64-bit adder is wasted
  => 1/2 of divisor is wasted
- Instead of shifting divisor to right,
  shift remainder to left?
- 1st step cannot produce a 1 in quotient bit
  (otherwise quotient is too big for the register)
  => switch order to shift first and then subtract
  => save 1 iteration
- Eliminate Quotient register by combining with
  Remainder register as shifted left

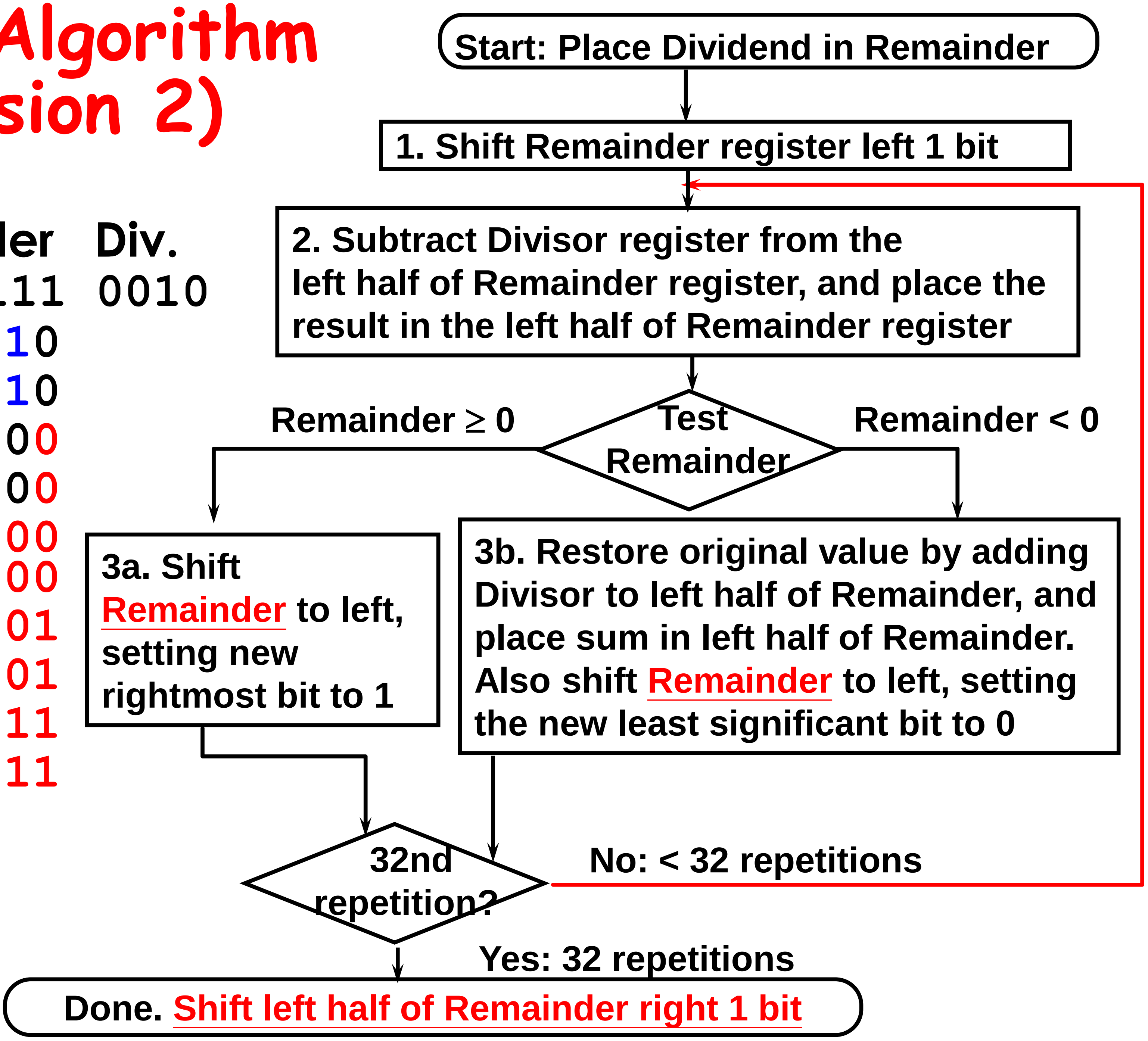
Divide Hardware (Version 2)

- 32-bit Divisor register, 32 -bit ALU, 64-bit Remainder register, (0-bit Quotient register)



Divide Algorithm (Version 2)

Step	Remainder		Div.
0	0000	0111	0010
1.1	0000	1110	
1.2	1110	1110	
1.3b	0001	1100	
2.2	1111	1100	
2.3b	0011	1000	
3.2	0001	1000	
3.3a	0011	0001	
4.2	0001	0001	
4.3a	0010	0011	
	0001	0011	


Start: Place Dividend in Remainder
1. Shift Remainder register left 1 bit
2. Subtract Divisor register from the left half of Remainder register, and place the result in the left half of Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
3a. Shift Remainder to left, setting new rightmost bit to 1
3b. Restore original value by adding Divisor to left half of Remainder, and place sum in left half of Remainder. Also shift Remainder to left, setting the new least significant bit to 0
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done. Shift left half of Remainder right 1 bit

Divide

- Signed Divides:
  - Remember signs, make positive, complement quotient and remainder if necessary
  - Let Dividend and Remainder have same sign and negate Quotient if Divisor sign & Dividend sign disagree,
  - e.g., -7÷ 2 = -3, remainder = -1

    -7÷- 2 = 3, remainder = -1
  - Satisfy Dividend =Quotient x Divisor + Remainder
- Possible for quotient to be too large:

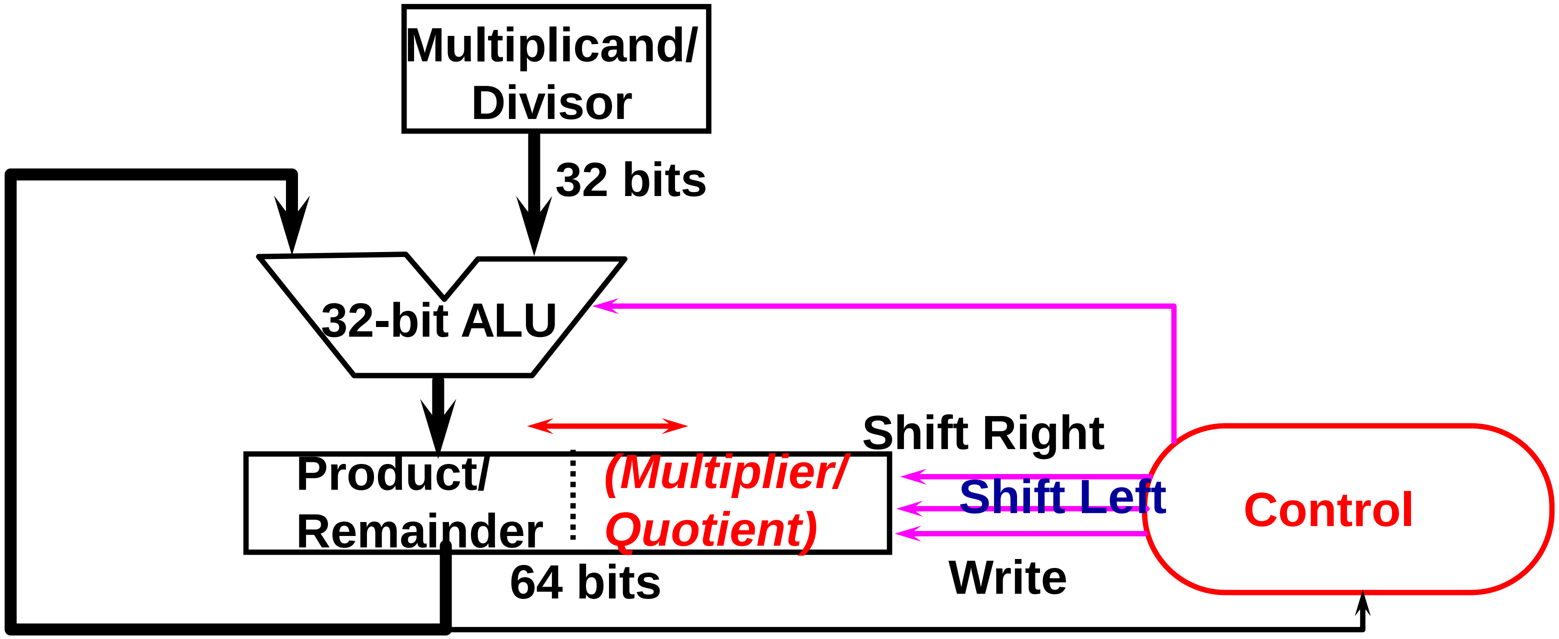
  if divide 64-bit integer by 1, quotient is 64 bits

Observations: Multiply and Divide

- Same hardware as multiply: just need ALU to add or subtract, and 64-bit register to shift left or shift right
- Hi and Lo registers in MIPS combine to act as 64-bit register for multiply and divide

Multiply/Divide Hardware

- 32-bit Multiplicand/Divisor register, 32 -bit ALU, 64-bit Product/Remainder register, (0-bit Multiplier/Quotient register)



Outline

- Addition and subtraction (Sec. 3.2)
- Constructing an arithmetic logic unit (Appendix C)
- Multiplication (Sec. 3.3, Appendix C)
- Division (Sec. 3.4)
- Floating point (Sec. 3.5)

Floating-Point: Motivation

- What can be represented in N bits?

Unsigned	0	to	$2^n - 1$
2's Complement	-2^{n-1}	to	$2^{n-1} - 1$
1's Complement	$-2^{n-1}+1$	to	2^{n-1}
Excess M	$-M$	to	$2^n - M - 1$

- But, what about ...
  - very large numbers?
    9,349,398,989,787,762,244,859,087,678
  - very small number?
    0.0000000000000000000000045691
  - rationals 2/3
  - irrationals $\sqrt{2}$
  - transcendentals e, π

Scientific Notation: Binary

Significand (Mantissa) ***exponent***

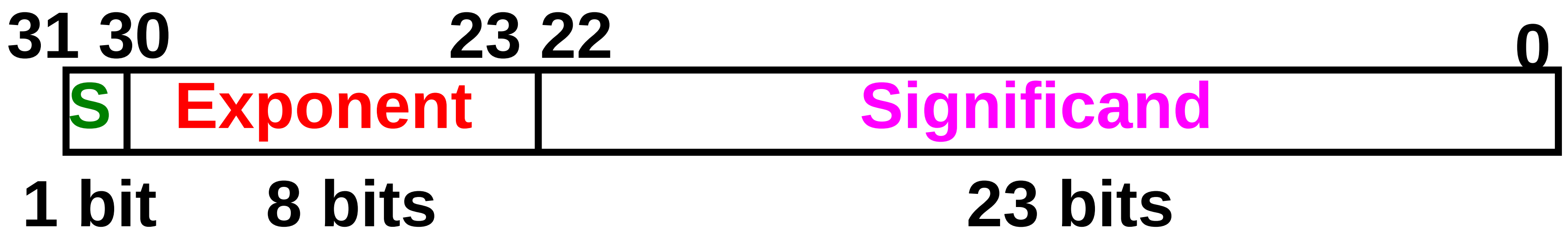
$$1.0_{two} \times 2^{-1}$$

“binary point” ***radix (base)***

- Computer arithmetic that supports it is called floating point, because the binary point is not fixed, as it is for integers
- Normalized form: no leading 0s
  (exactly one digit to left of decimal point)
- Alternatives to represent 1/1,000,000,000
  - Normalized: 1.0×10^{-9}
  - Not normalized: 0.1×10^{-8}, 10.0×10^{-10}

FP Representation

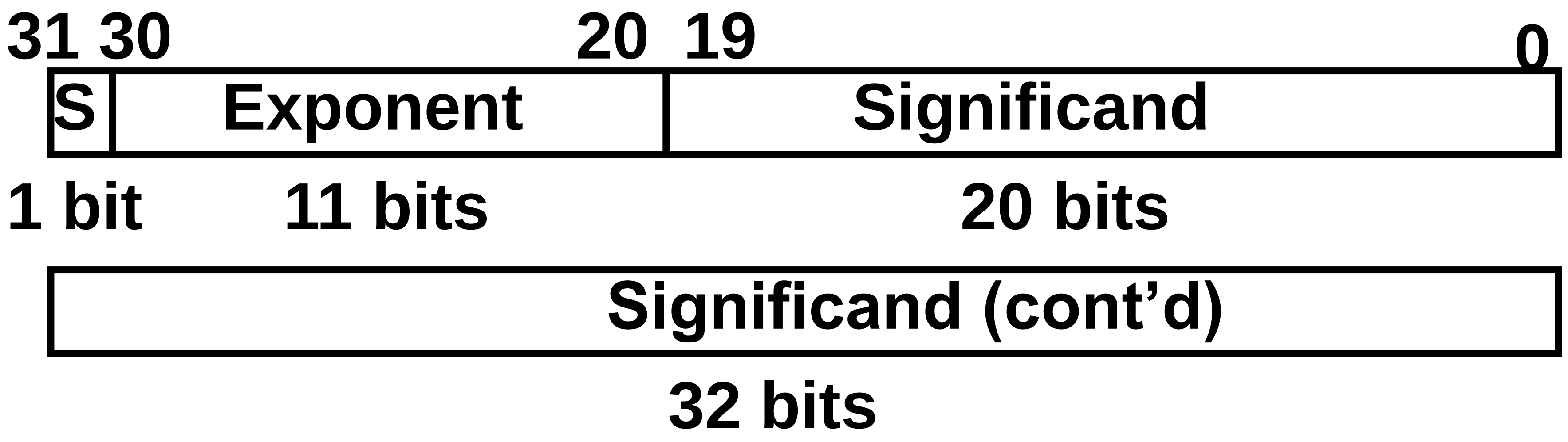
- Normal format: 1.xxxxxxxxxx$_{two}$ × 2^{yyyy}$_{two}$
- Want to put it into multiple words: 32 bits for *single-precision* and 64 bits for *double-precision*
- A simple single-precision representation:



S represents sign
Exponent represents y's
Significand represents x's

Double Precision Representation

- Next multiple of word size (64 bits)



- Double precision (vs. single precision)
  - But primary advantage is greater accuracy due to larger significand

IEEE 754 Standard (1/4)

- Regarding single precision, DP similar
- Sign bit:

  1 means negative

  0 means positive
- Significand:
  - To pack more bits, leading 1 implicit for normalized numbers
  - 1 + 23 bits single, 1 + 52 bits double
  - always true: 0 < Significand < 1

    (for normalized numbers)
- Note: 0 has no leading 1, so reserve exponent value 0 just for number 0

IEEE 754 Standard (2/4)

- Exponent:
  - Need to represent positive and negative exponents
  - Also want to compare FP numbers as if they were <u>integers</u>, to help in value comparisons
  - If use 2's complement to represent?
    e.g., 1.0 x 2^{-1} versus 1.0 x2^{+1} (1/2 versus 2)

1/2	0	1111 1111	000 0000 0000 0000 0000 0000
2	0	0000 0001	000 0000 0000 0000 0000 0000

If we use integer comparison for these two words, we will conclude that 1/2 > 2!!!

Biased (Excess) Notation

- Biased 7

0000	-7
0001	-6
0010	-5
0011	-4
0100	-3
0101	-2
0110	-1
0111	0
1000	1
1001	2
1010	3
1011	4
1100	5
1101	6
1110	7
1111	8

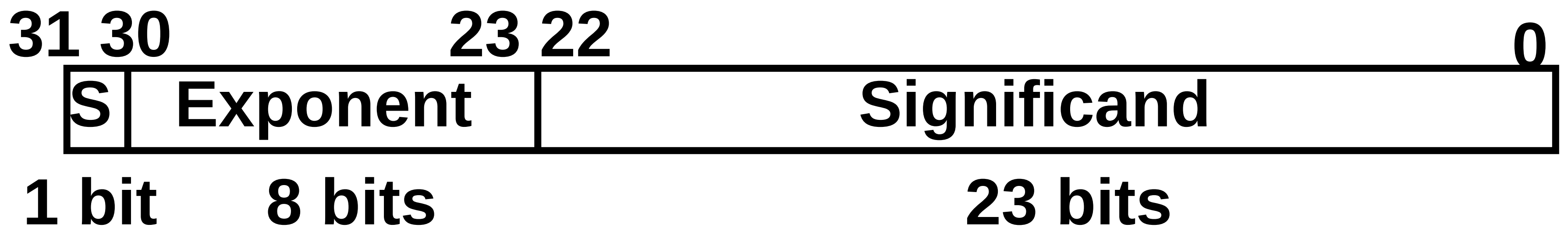
IEEE 754 Standard (3/4)

- Instead, let notation 0000 0000 be most negative, and 1111 1111 most positive
- Called biased notation, where bias is the number subtracted to get the real number
  - IEEE 754 uses bias of 127 for single precision: Subtract 127 from Exponent field to get actual value for exponent
  - 1023 is bias for double precision

1/2	0	0111 1110	000 0000 0000 0000 0000 0000
2	0	1000 0000	000 0000 0000 0000 0000 0000

IEEE 754 Standard (4/4)

- Summary (single precision):



$(-1)^S \times (1.\text{Significand}) \times 2^{(\text{Exponent}-127)}$

- Double precision identical, except with exponent bias of 1023

Example: FP to Decimal

0	0110 1000	101 0101 0100 0011 0100 0010

- Sign: 0 => positive
- Exponent:
  - $0110\ 1000_{two} = 104_{ten}$
  - Bias adjustment: $104 - 127 = -23$
- Significand:
  - $1+2^{-1}+2^{-3}+2^{-5}+2^{-7}+2^{-9}+2^{-14}+2^{-15}+2^{-17}+2^{-22}$ $= 1.0 + 0.666115$
- Represents: $1.666115_{ten}\times2^{-23} \approx 1.986 \times 10^{-7}$

Example 1: Decimal to FP

- Number = - 0.75
  = - $0.11_{two} \times 2^0$ (scientific notation)
  = - $1.1_{two} \times 2^{-1}$ (normalized scientific notation)

- Sign: negative => 1
- Exponent:
  - Bias adjustment: -1 +127 = 126
  - 126_{ten} = $0111\ 1110_{two}$

1	0111 1110	100 0000 0000 0000 0000 0000

Example 2: Decimal to FP

- A more difficult case: representing 1/3?

  $= 0.33333\ldots_{10} = 0.0101010101\ldots_{2} \times 2^{0}$

  $= 1.0101010101\ldots_{2} \times 2^{-2}$

  - Sign: 0
  - Exponent = -2 + 127 = 125_{10}=01111101_{2}
  - Significand = 0101010101…

0	0111 1101	0101 0101 0101 0101 0101 010

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    $\Rightarrow$ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    $\Rightarrow$ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 $\Rightarrow$ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Precision

- Relative precision
  - all fraction bits are significant
  - Single: approx 2^{-23}
    - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
  - Double: approx 2^{-52}
    - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

Zero and Special Numbers

- What have we defined so far? (single precision)

Exponent	Significand	Object
0	0	???
0	nonzero	???
1-254	anything	+/- floating-point
255	0	???
255	nonzero	???

Representation for 0

- Represent 0?
  - exponent all zeroes
  - significand all zeroes too
  - What about sign?
  - `+0:  0  00000000  00000000000000000000000`
  - `-0:  1  00000000  00000000000000000000000`
- Why two zeroes?
  - Helps in some limit comparisons

Special Numbers

- What have we defined so far? (single precision)

Exponent	Significand	Object
0	0	0
0	nonzero	???
1-254	anything	+/- floating-point
255	0	???
255	nonzero	???

- Range:

  $1.0 \times 2^{-126} \approx 1.8 \times 10^{-38}$

  What if result too small? (>0, $< 1.8 \times 10^{-38}$ => Underflow!)

  $(2 - 2^{-23}) \times 2^{127} \approx 3.4 \times 10^{38}$

  What if result too large? ($> 3.4 \times 10^{38}$ => Overflow!)

Gradual Underflow

- Represent denormalized numbers (denorms)
  - Exponent : all zeroes
  - Significand : non-zeroes
  - Allow a number to degrade in significance until it become 0 (gradual underflow)

  - The smallest normalized number
    - $1.0000\ 0000\ 0000\ 0000\ 0000\ 0000 \times 2^{-126}$
  - The smallest de-normalized number
    - $0.0000\ 0000\ 0000\ 0000\ 0000\ 0001 \times 2^{-126}$

Special Numbers

- What have we defined so far? (single precision)

Exponent	Significand	Object
0	0	0
0	nonzero	denorm
1-254	anything	+/- floating-point
255	0	???
255	nonzero	???

Representation for +/- Infinity

- In FP, divide by zero should produce +/- infinity, not overflow
- Why?
  - OK to do further computations with infinity, e.g., X/0 > Y may be a valid comparison
- IEEE 754 represents +/- infinity
  - Most positive exponent reserved for infinity
  - Significands all zeroes

S	1111 1111	0000 0000 0000 0000 0000 000

Special Numbers (cont'd)

- What have we defined so far? (single-precision)

Exponent	Significand	Object
0	0	0
0	nonzero	denom
1-254	anything	+/- fl. pt. #
255	0	+/- infinity
255	nonzero	???

Representation for Not a Number

- What do I get if I calculate sqrt(-4.0) or 0/0?
  - If infinity is not an error, these should not be either
  - They are called *Not a Number* (NaN)
  - Exponent = 255, Significand nonzero
- Why is this useful?
  - Hope NaNs help with debugging?
  - They contaminate: op(NaN,X) = NaN
  - OK if calculate but don’t use it

Special Numbers (cont'd)

- What have we defined so far? (single-precision)

Exponent	Significand	Object
0	0	0
0	nonzero	denom
1-254	anything	+/- fl. pt. #
255	0	+/- infinity
255	nonzero	NaN

Floating-Point Addition

Basic addition algorithm:

(1) Align binary point :compute Ye – Xe

- ♦ right shift the smaller number, say Xm, that many positions to form Xm $\times$ 2^{Xe-Ye}

(2) Add mantissa: compute Xm $\times$ 2^{Xe-Ye} + Ym

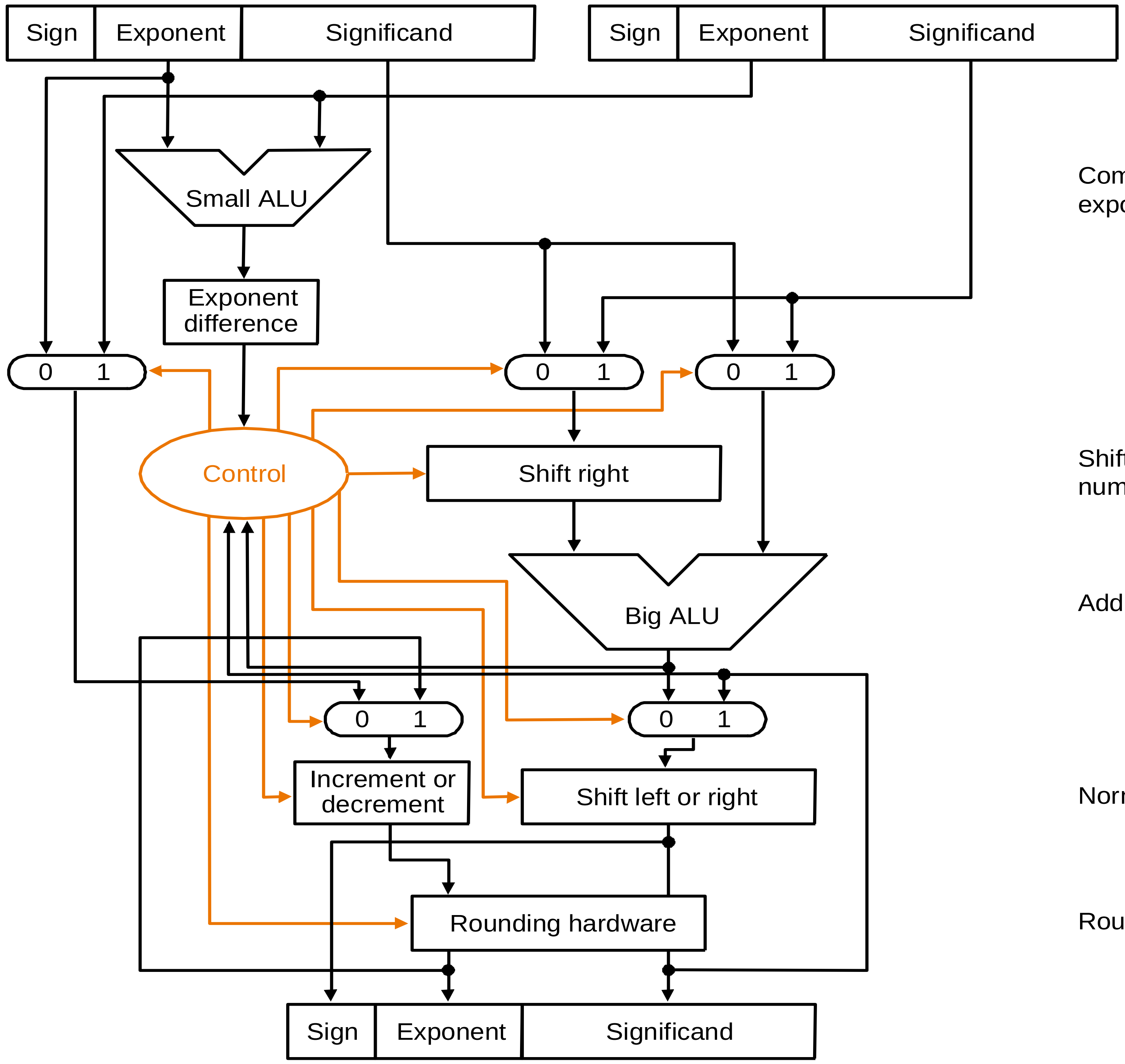
(3) Normalization & check for over/underflow if necessary:

- left shift result, decrement result exponent
- right shift result, increment result exponent
- check overflow or underflow during the shift

(4) Round the mantissa and renormalize if necessary

Floating-Point Addition Example

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)
- 1. Align binary points
  - Shift number with smaller exponent
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. Add mantissa
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-}1 = 0.001_2 \times 2^{-1}$
- 3. Normalize result & check for over/underflow
  - $1.000_2 \times 2^{-4}$, with no over/underflow
- 4. Round and renormalize if necessary
  - $1.000_2 \times 2^{-4}$ (no change) = 0.0625



Compare exponents

Shift smaller number right

Step 1

Add

Step 2

Normalize

Step 3

Round

Step 4

FP Adder Hardware

- Much more complex than integer adder
- Doing it in one clock cycle would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP adder usually takes several cycles
  - Can be pipelined

Floating-Point Multiplication

Basic multiplication algorithm

(1) Add exponents of operands to get exponent of product

doubly biased exponent must be corrected:

```
Xe = 7
Ye = -3       Xe = 1111      = 15       =  7 + 8
Excess 8      Ye = 0101      =   5      = -3 + 8
                   10100       20          4 + 8 + 8
```

need extra subtraction step of the bias amount

(2) Multiplication of operand mantissa

(3) Normalize the product & check overflow or underflow during the shift

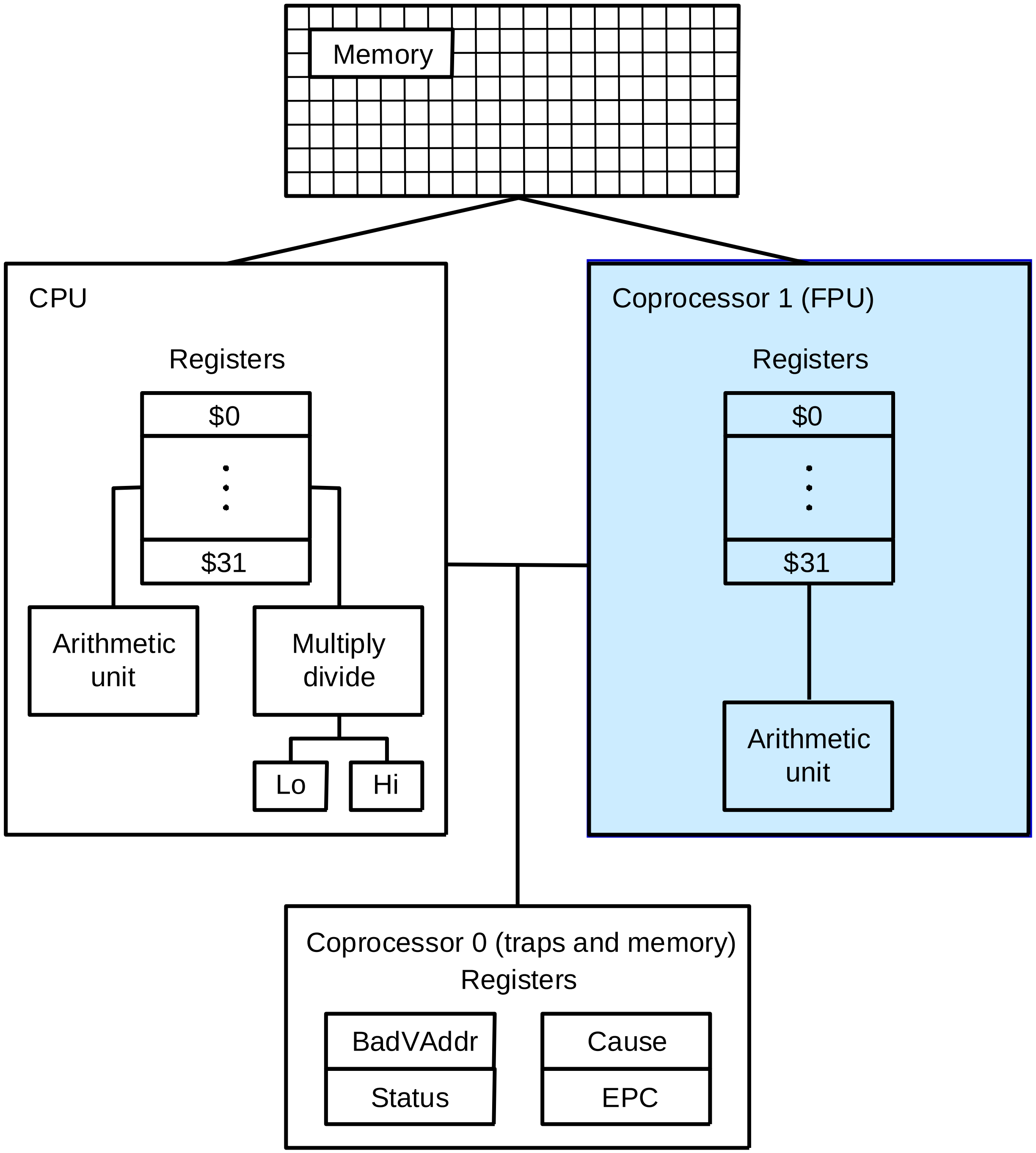
(4) Round the mantissa and renormalize if necessary

(5) Set the sign of product

Floating-Point Multiplication Example

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5×-0.4375)
- 1. Add exponents
  - Unbiased: $-1 + -2 = -3$
  - Biased: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$
- 2. Multiply operand mantissa
  - $1.000_2 \times 1.110_2 = 1.1102 \;\Rightarrow\; 1.110_2 \times 2^{-3}$
- 3. Normalize result & check for over/underflow
  - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
- 4. Round and renormalize if necessary
  - $1.110_2 \times 2^{-3}$ (no change)
- 5. Determine sign:
  - $-1.110_2 \times 2^{-3} = -0.21875$

MIPS R2000 Organization



MIPS Floating Point

- Separate floating point instructions:
  - Single precision: `add.s,sub.s,mul.s,div.s`
  - Double precision: `add.d,sub.d,mul.d,div.d`
- FP part of the processor:
  - contains 32 32-bit registers: `$f0`, `$f1`, …
  - most registers specified in .s and .d instruction refer to this set
  - Double precision: by convention, even/odd pair contain one DP FP number: `$f0/$f1`, `$f2/$f3`
  - separate load and store: `lwc1` and `swc1`
  - Instructions to move data between main processor and coprocessors:
    - `mfc0`, `mtc0`, `mfc1`, `mtc1`, etc.

Interpretation of Data

The BIG Picture

- Bits have no inherent meaning
  - Interpretation depends on the instructions applied
- Computer representations of numbers
  - Finite range and precision
  - Need to account for this in programs

Associativity

- Floating Point add, subtract associative ?

		(x+y)+z	x+(y+z)
x	-1.50E+38		-1.50E+38
y	1.50E+38	0.00E+00	
z	1.0	1.0	1.50E+38
		1.00E+00	0.00E+00

- Therefore, Floating Point add, subtract are not associative!
  - Why? FP result approximates real result!
  - This example: 1.5 x 10^{38} is so much larger than 1.0 that 1.5 x 10^{38} + 1.0 in floating point representation is still 1.5 x 10^{38}

Associativity in Parallel Programming

- Parallel programs may interleave operations in unexpected orders
  - Assumptions of associativity may fail
- Need to validate parallel programs under varying degrees of parallelism

x86 FP Architecture

- Originally based on 8087 FP coprocessor
  - 8 × 80-bit extended-precision registers
  - Used as a push-down stack
  - Registers indexed from TOS: ST(0), ST(1), …
- FP values are 32-bit or 64 in memory
  - Converted on load/store of memory operand
  - Integer operands can also be converted on load/store
- Very difficult to generate and optimize code
  - Result: poor FP performance

x86 FP Instructions

Data transfer	Arithmetic	Compare	Transcendental
FILD mem/ST(i)	FIADDP mem/ST(i)	FICOMP	FPATAN
FISTP mem/ST(i)	FISUBRP mem/ST(i)	FIUCOMP	F2XMI
FLDPI	FIMULP mem/ST(i)	FSTSW AX/mem	FCOS
FLD1	FIDIVRP mem/ST(i)		FPTAN
FLDZ	FSQRT		FPREM
	FABS		FPSIN
	FRNDINT		FYL2X

- Optional variations
  - I: integer operand
  - P: pop operand from stack
  - R: reverse operand order
  - But not all combinations allowed

Streaming SIMD Extension 2 (SSE2)

- Adds 4 $\times$ 128-bit registers
  - Extended to 8 registers in AMD64/EM64T
- Can be used for multiple FP operands
  - 2 $\times$ 64-bit double precision
  - 4 $\times$ 32-bit double precision
  - Instructions operate on them simultaneously
    - Single-Instruction Multiple-Data

Right Shift and Division

- Left shift by *i* places multiplies an integer by 2^i
- Right shift divides by 2^i?
  - Only for unsigned integers
- For signed integers
  - Arithmetic right shift: replicate the sign bit
  - e.g., –5 / 4
    - 11111011_2 >> 2 = 11111110_2 = –2
    - Rounds toward –∞
  - c.f. 11111011_2 >>> 2 = 00111110_2 = +62

Who Cares About FP Accuracy?

- Important for scientific code
  - But for everyday consumer use?
    - "My bank balance is out by 0.0002¢!" ☹
- The Intel Pentium FDIV bug
  - The market expects accuracy
  - See Colwell, *The Pentium Chronicles*

Concluding Remarks

- ISAs support arithmetic
  - Signed and unsigned integers
  - Floating-point approximation to reals
- Bounded range and precision
  - Operations can overflow and underflow
- MIPS ISA
  - Core instructions: 54 most frequently used
    - 100% of SPECINT, 97% of SPECFP
  - Other instructions: less frequent