CENG 3420
Computer Organization and Design

Lecture 05: ALU Review

Bei Yu

香港中文大學
The Chinese University of Hong Kong

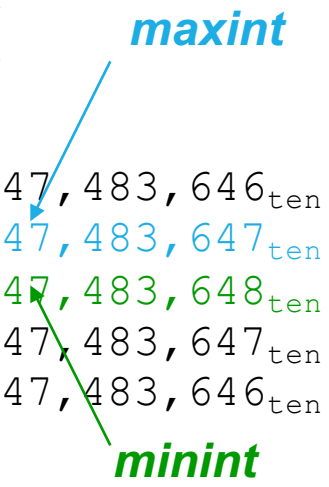
MIPS Representations

- 32-bit signed numbers (2's complement):

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +\ 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = +\ 2_{ten}$
...

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +\ 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -\ 2{,}147{,}483{,}646_{ten}$
...

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -\ 3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1_{ten}$



- What if the bit string represented addresses?
  - need operations that also deal with only positive (unsigned) integers

Two's Complement Operations

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

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

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

  - sign extension versus zero extend (`lb` vs. `lbu`)

Design the MIPS Arithmetic Logic Unit (ALU)

- Must support the Arithmetic/Logic operations of the ISA

  `add, addi, addiu, addu`

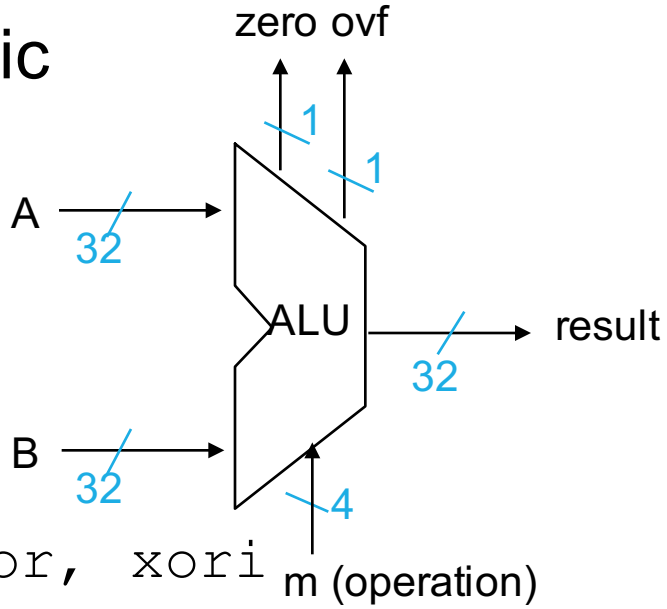
  `sub, subu`

  `mult, multu, div, divu`

  `sqrt`

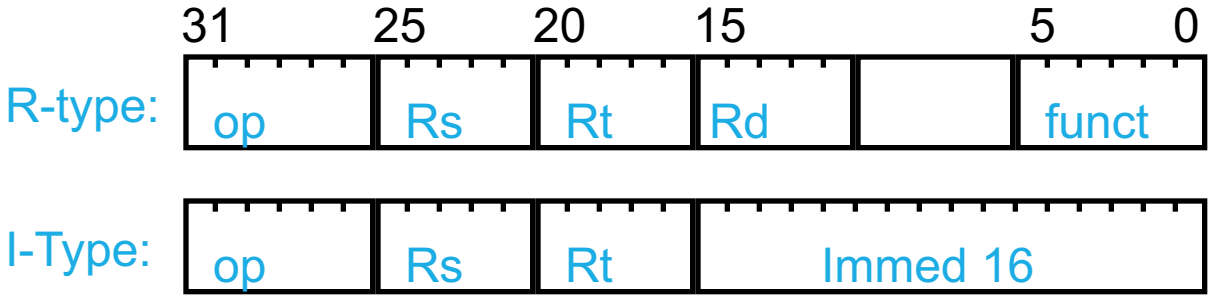
  `and, andi, nor, or, ori, xor, xori`

  `beq, bne, slt, slti, sltiu, sltu`



- With special handling for
  - sign extend – `addi, addiu, slti, sltiu`
  - zero extend – `andi, ori, xori`
  - Overflow detected – `add, addi, sub`

MIPS Arithmetic and Logic Instructions



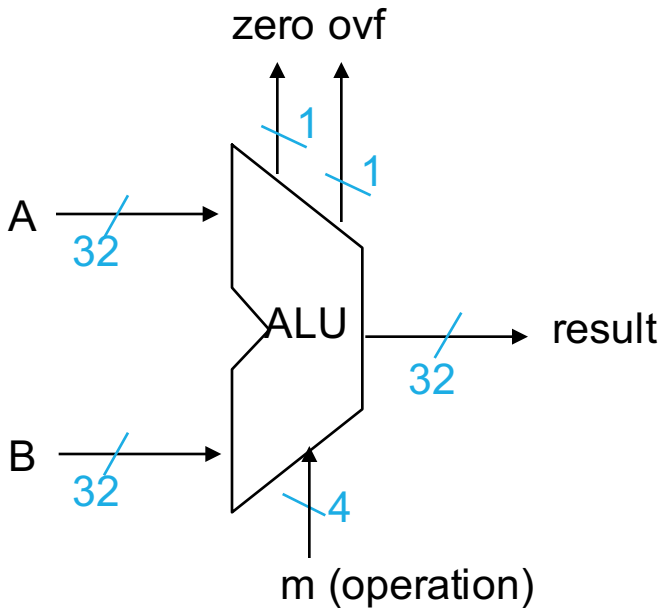
Type	op	funct
ADDI	001000	xx
ADDIU	001001	xx
SLTI	001010	xx
SLTIU	001011	xx
ANDI	001100	xx
ORI	001101	xx
XORI	001110	xx
LUI	001111	xx

Type	op	funct
ADD	000000	100000
ADDU	000000	100001
SUB	000000	100010
SUBU	000000	100011
AND	000000	100100
OR	000000	100101
XOR	000000	100110
NOR	000000	100111

Type	op	funct
	000000	101000
	000000	101001
SLT	000000	101010
SLTU	000000	101011
	000000	101100

Design Trick: Divide & Conquer

- Break the problem into simpler problems, solve them and glue together the solution
- Example: assume the immediates have been taken care of before the ALU
  - now down to 10 operations
  - can encode in 4 bits



0	add
1	addu
2	sub
3	subu
4	and
5	or
6	xor
7	nor
a	slt
b	sltu

Addition & Subtraction

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

```
  0111        0111        0110
+ 0110      - 0110      - 0101
------      ------      ------
  1101        0001        0001
```

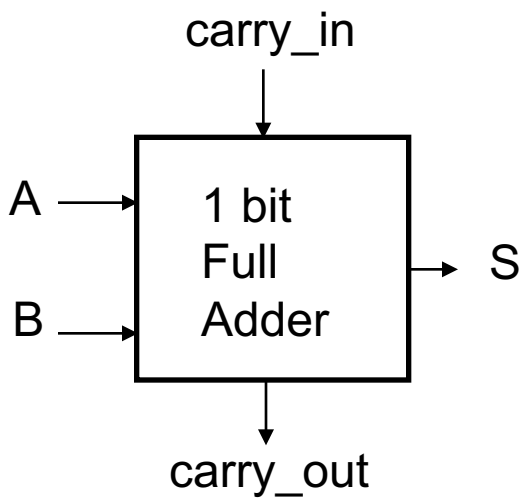
- Two's complement operations are easy
  - do subtraction by negating and then adding

```
  0111    ->      0111
- 0110    ->    + 1010
------            ------
  0001          1 0001
```

- 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
```

Building a 1-bit Binary Adder



A	B	carry_in	carry_out	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

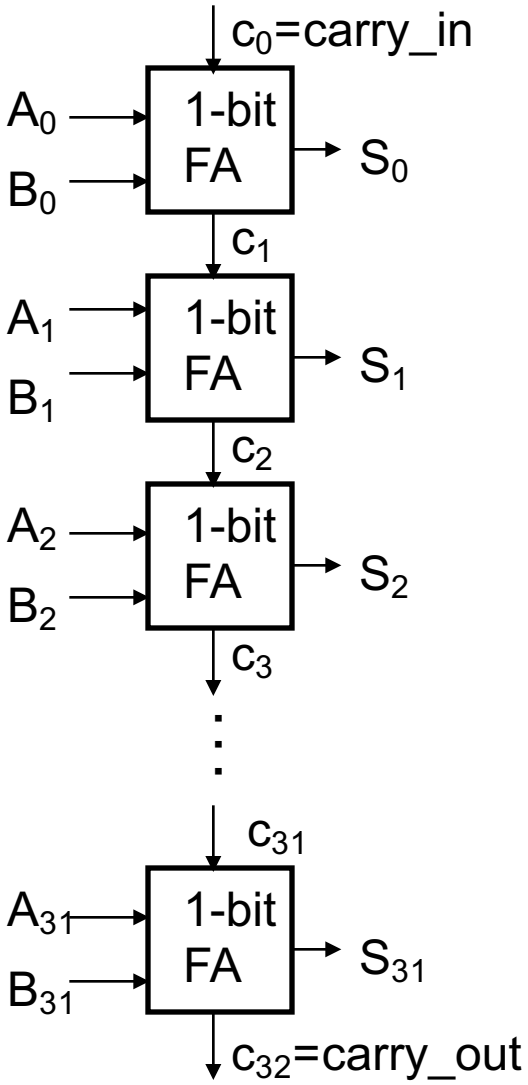
S = A xor B xor carry_in

carry_out = A&B | A&carry_in | B&carry_in

(majority function)

- How can we use it to build a 32-bit adder?
- How can we modify it easily to build an adder/subtractor?

Building 32-bit Adder

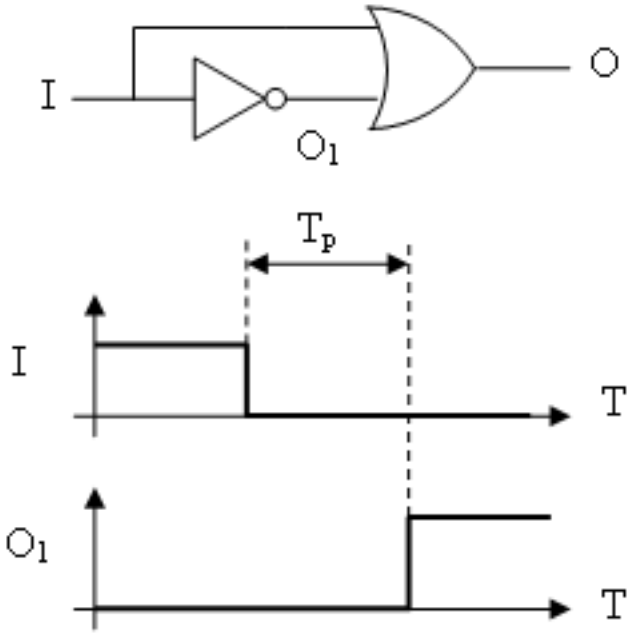


- Just connect the carry-out of the least significant bit FA to the carry-in of the next least significant bit and connect . . .

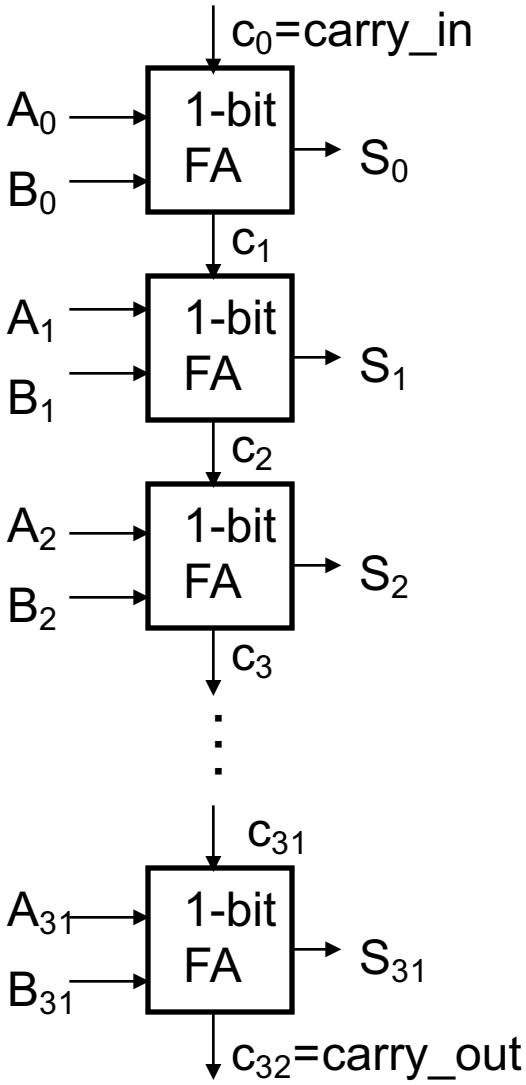
- Ripple Carry Adder (RCA)
  - advantage: simple logic, so small (low cost)
  - disadvantage: slow and lots of glitching (so lots of energy consumption)

Glitch

- Glitch: invalid and unpredicted output that can be read by the next stage and result in a wrong action
- Example: Draw the propagation delay



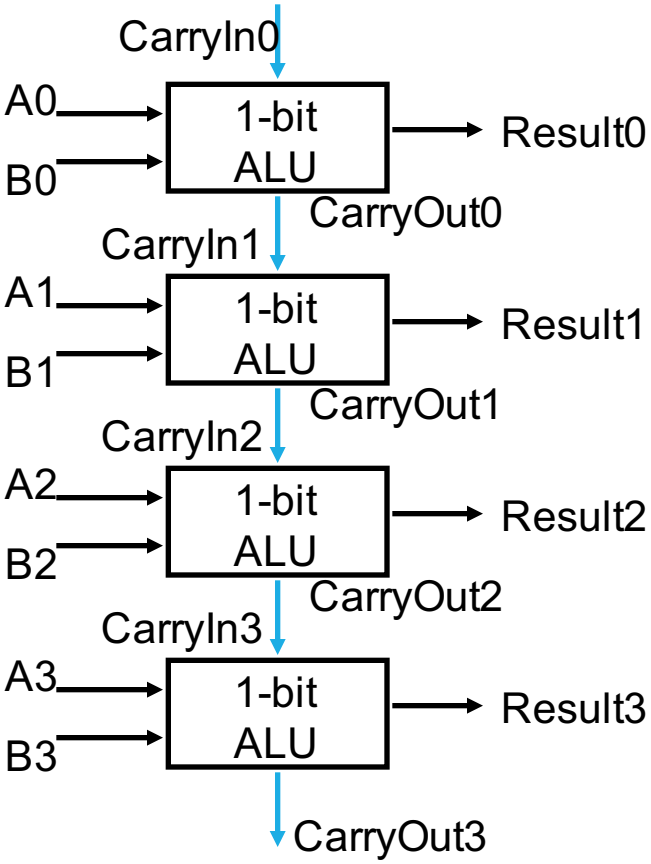
Glitch in RCA



A	B	carry_in	carry_out	S
0	0	**0**	0	**0**
0	0	**1**	0	**1**
0	1	**0**	0	**1**
0	1	**1**	1	**0**
1	0	**0**	0	**1**
1	0	**1**	1	**0**
1	1	**0**	1	**0**
1	1	**1**	1	**1**

But What about Performance?

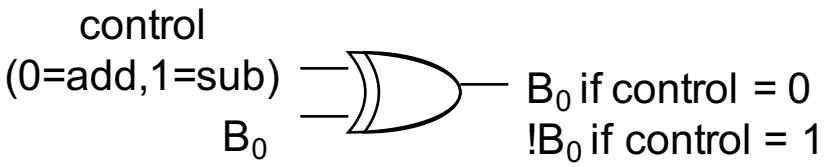
- Critical path of n-bit ripple-carry adder is n*CP



- Design trick – throw hardware at it (Carry Lookahead)

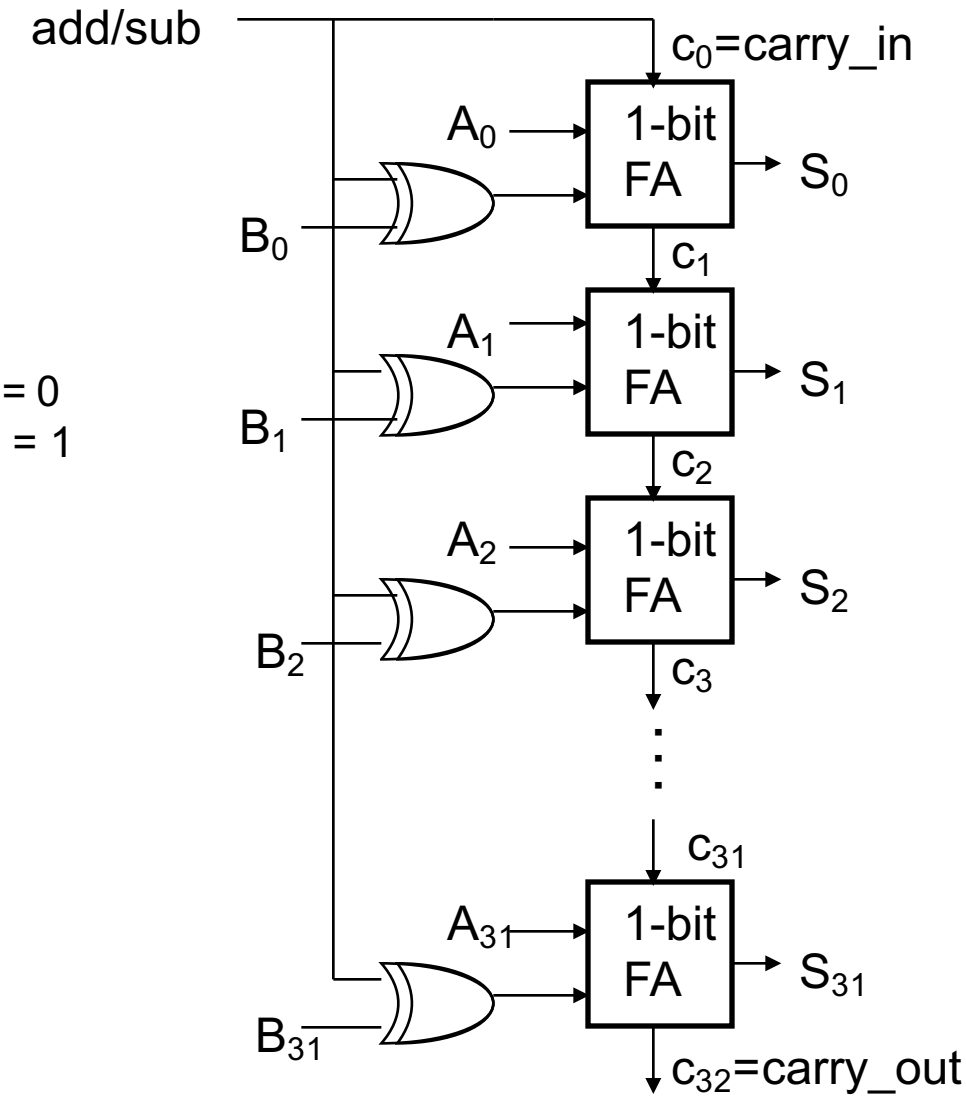
A 32-bit Ripple Carry Adder/Subtractor

- Remember 2's complement is just
  - complement all the bits

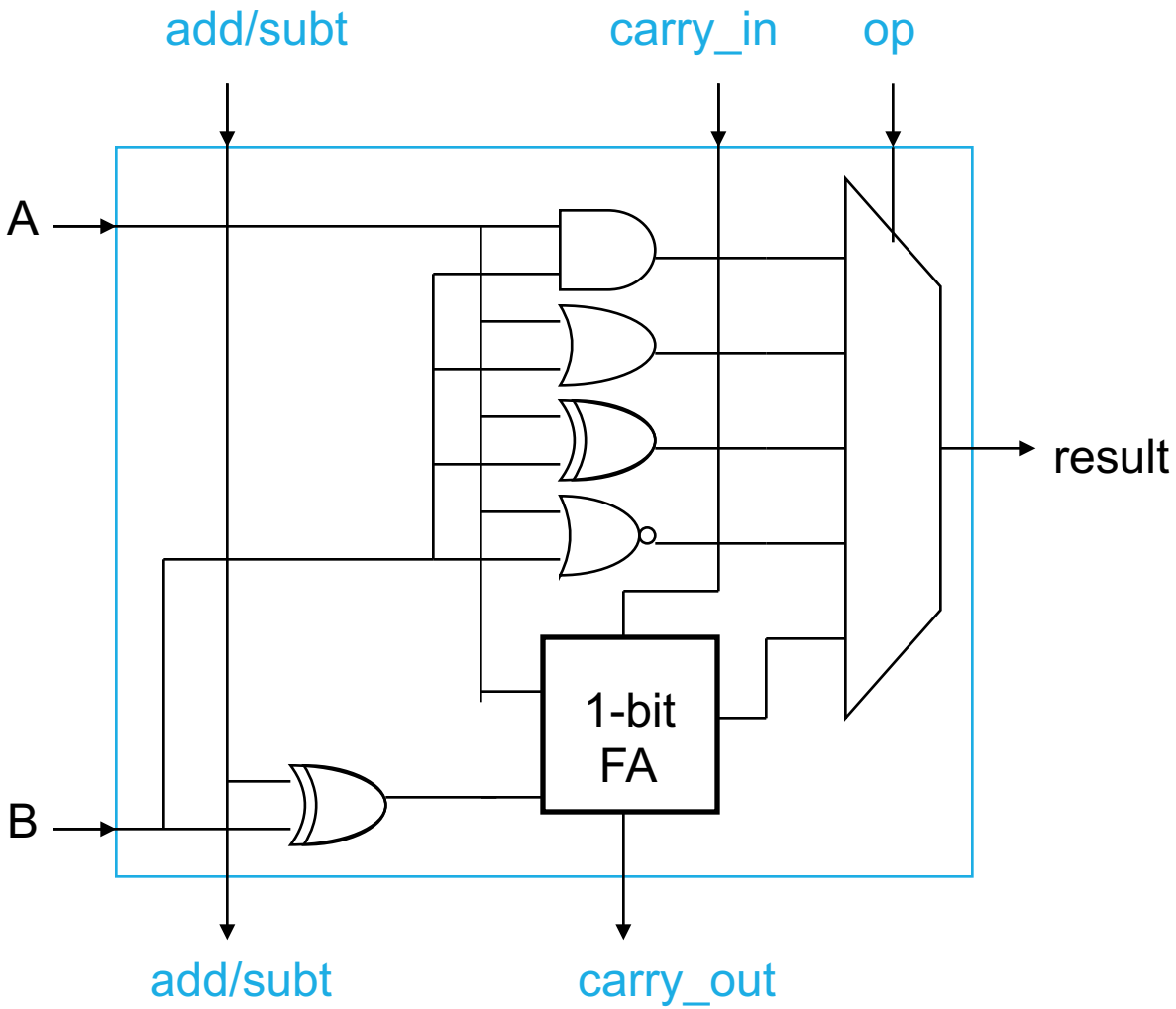


  - add a 1 in the least significant bit

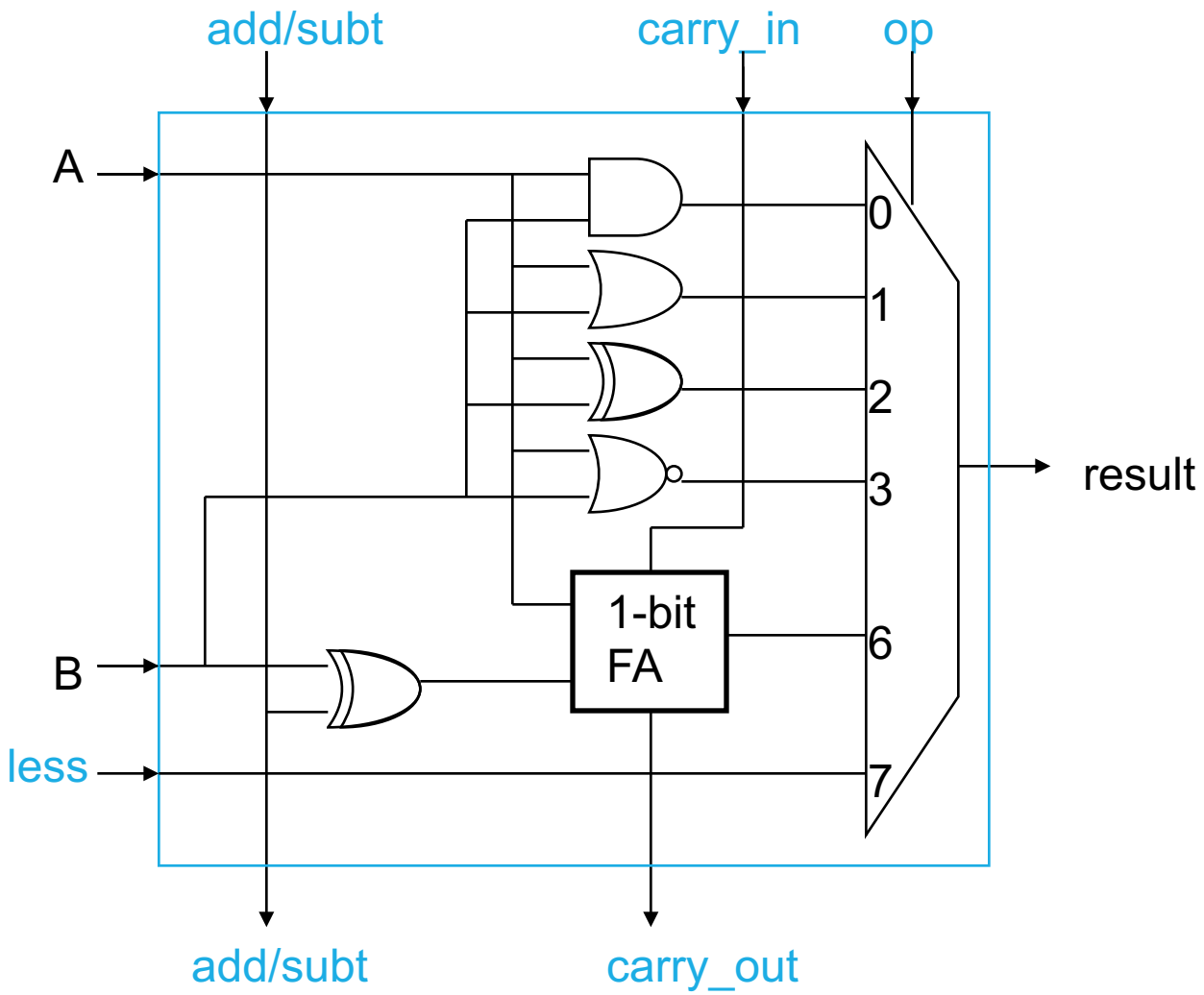
```
A    0111   ->    0111
B  - 0110   -> +  1001
     0001              1
                  1 0001
```



A Simple ALU Cell with Logic Op Support

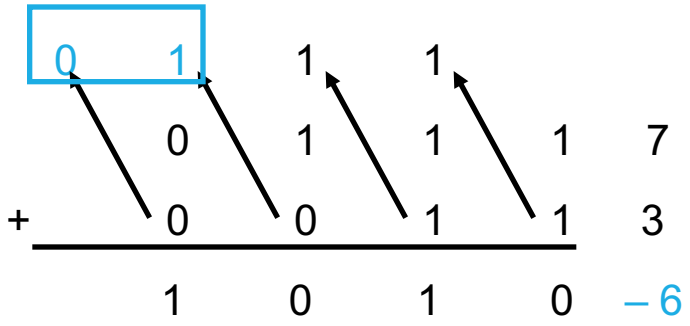


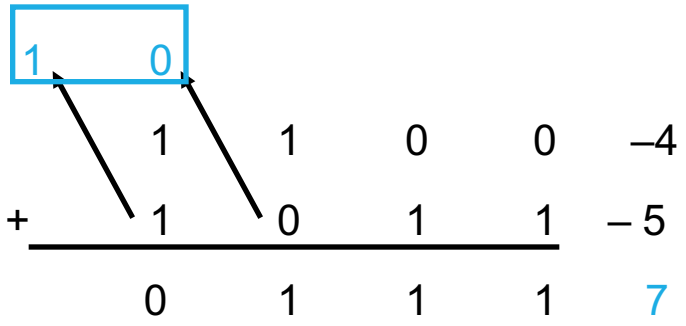
Modifying the ALU Cell for `slt`



Overflow Detection

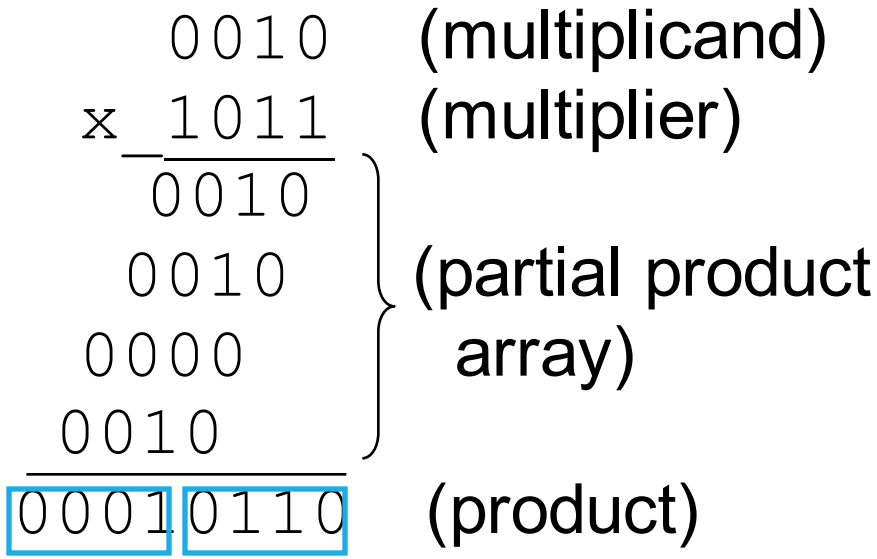
- Overflow occurs when the result is too large to represent in the number of bits allocated
  - adding two positives yields a negative
  - or, adding two negatives gives a positive
  - or, subtract a negative from a positive gives a negative
  - or, subtract a positive from a negative gives a positive
- On your own: Prove you can detect overflow by:
  - Carry into MSB xor Carry out of MSB





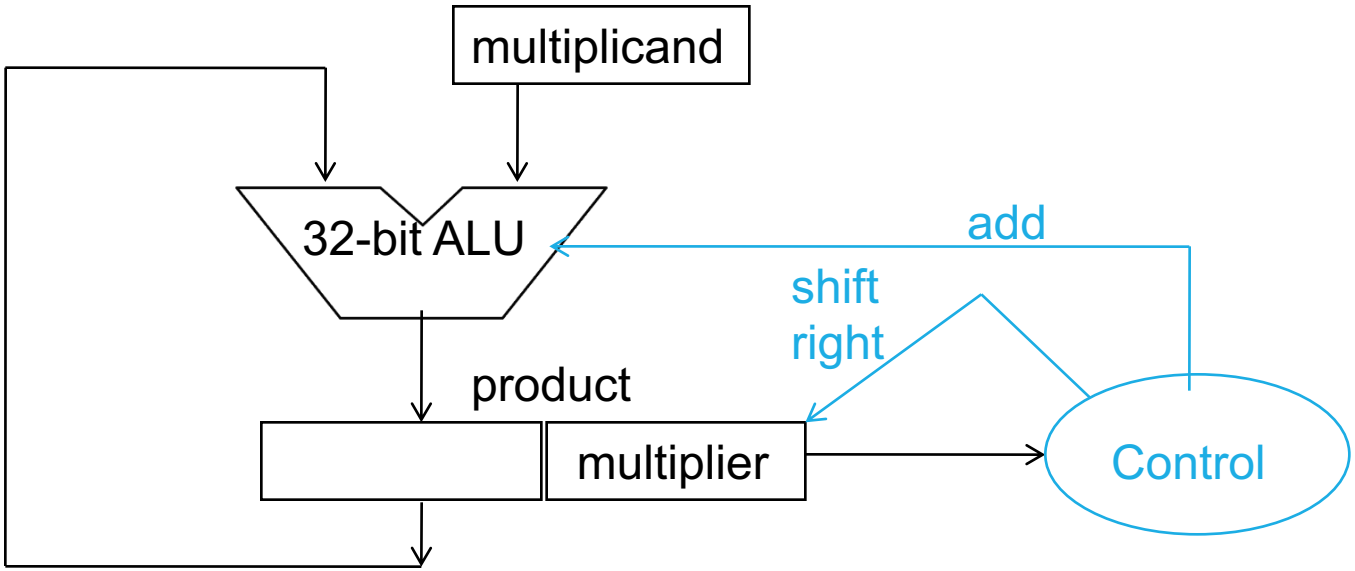
Multiplication

- More complicated than addition
  - Can be accomplished via shifting and adding

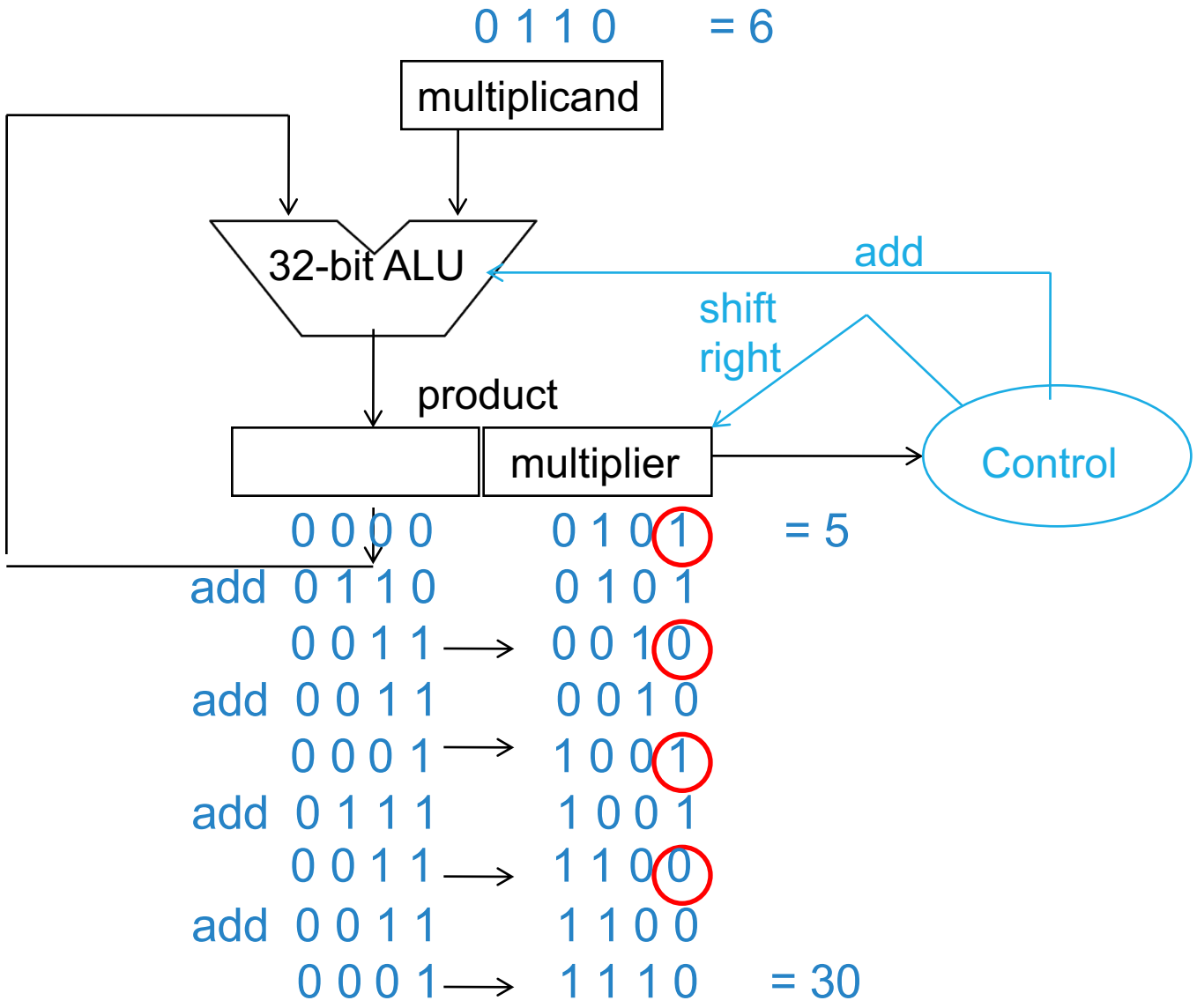


- Double precision product produced
- More time and more area to compute

Add and Right Shift Multiplier Hardware



Add and Right Shift Multiplier Hardware



MIPS Multiply Instruction

- Multiply (`mult` and `multu`) produces a double precision product

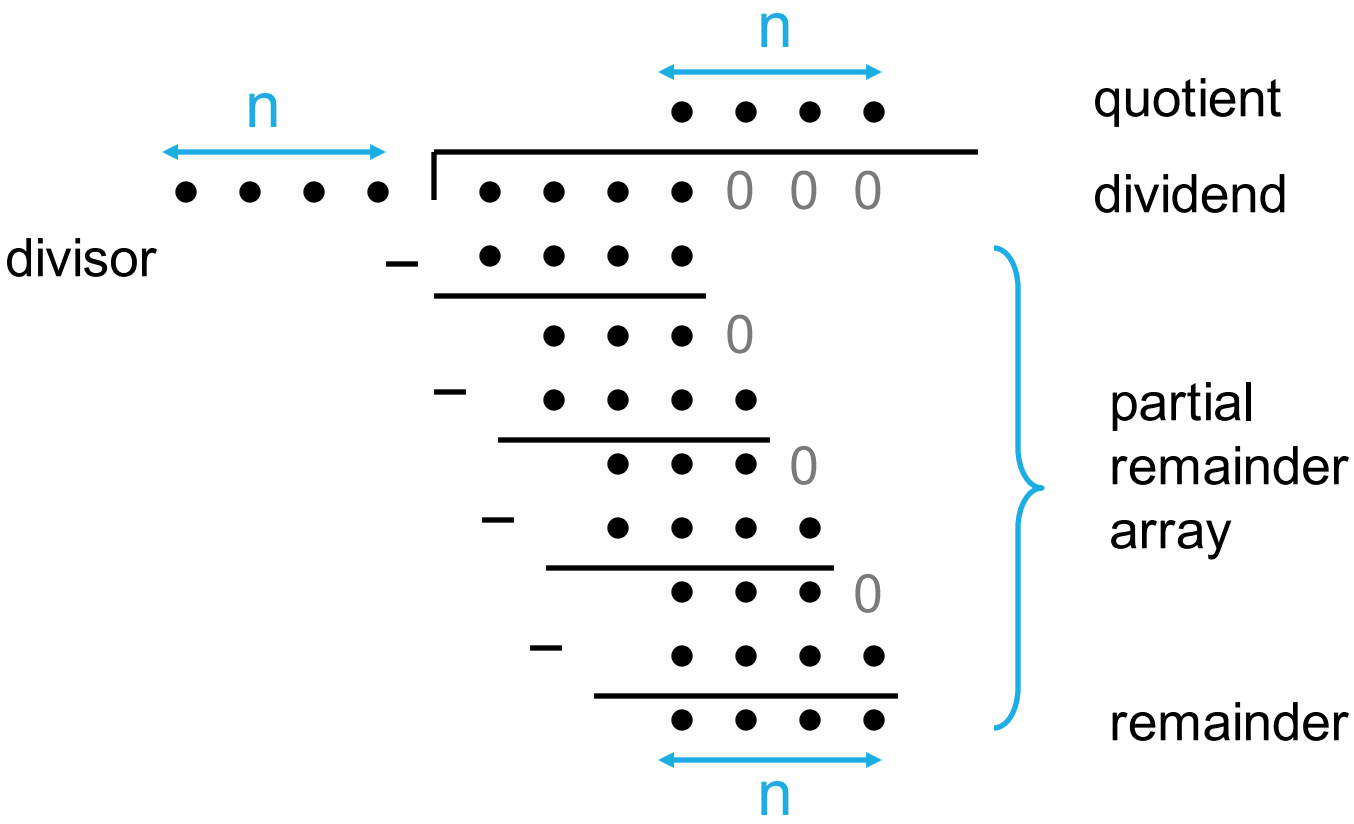
```
mult   $s0, $s1    # hi||lo = $s0 * $s1
```

  - Low-order word of the product is left in processor register `lo` and the high-order word is left in register `hi`
  - Instructions `mfhi rd` and `mflo rd` are provided to move the product to (user accessible) registers in the register file

- Multiplies are usually done by fast, dedicated hardware and are much more complex (and slower) than adders

Division

- Division is just a *bunch* of quotient digit guesses and left shifts and subtracts



Example: Division

- Dividing 1001010 by 1000

MIPS Divide Instruction

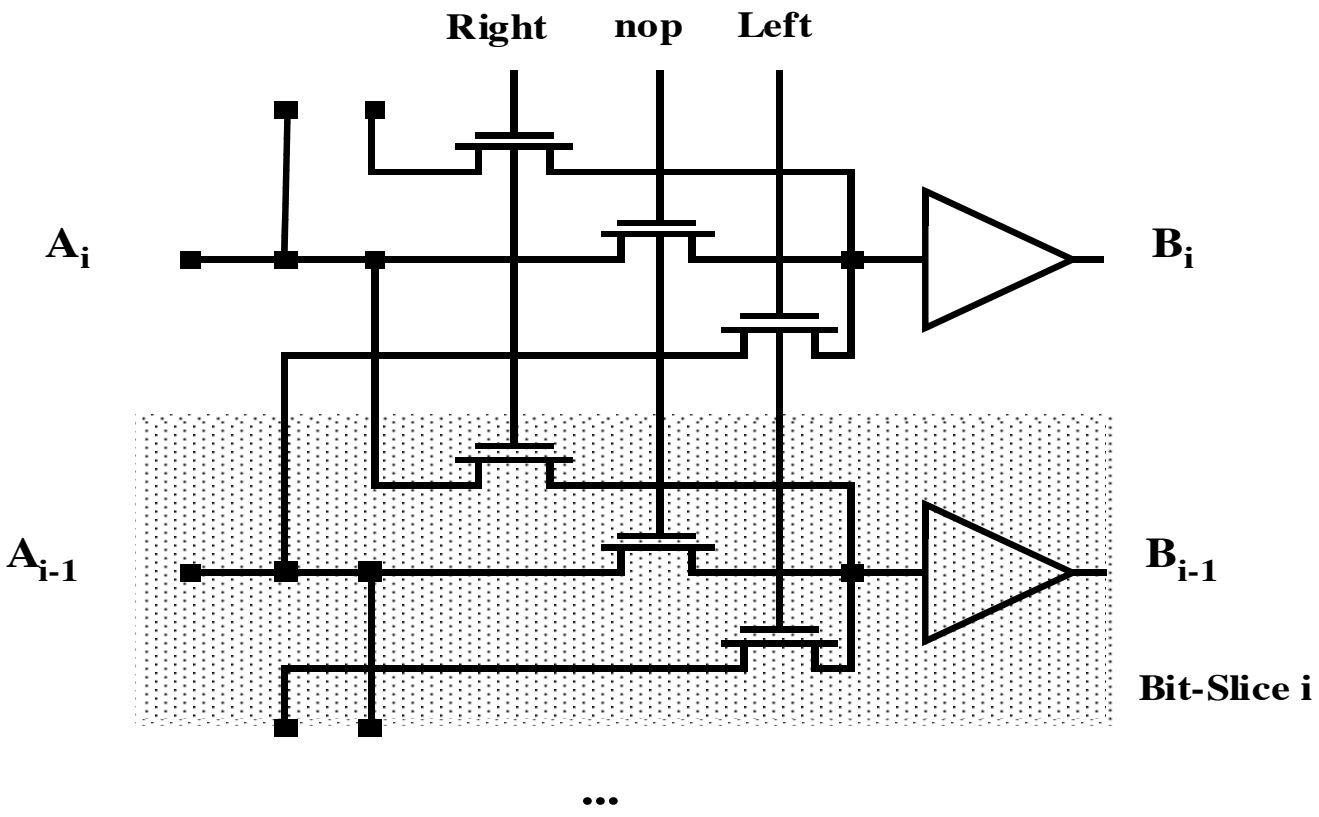
- Divide generates the reminder in `hi` and the quotient in `lo`

```
div  $s0, $s1        # lo = $s0 / $s1
                     # hi = $s0 mod $s1
```

  - Instructions `mflo rd` and `mfhi rd` are provided to move the quotient and reminder to (user accessible) registers in the register file

- As with multiply, divide ignores overflow so software must determine if the quotient is too large. Software must also check the divisor to avoid division by 0.

A Simple Shifter



Representing Big (and Small) Numbers

- What if we want to encode the approx. age of the earth?

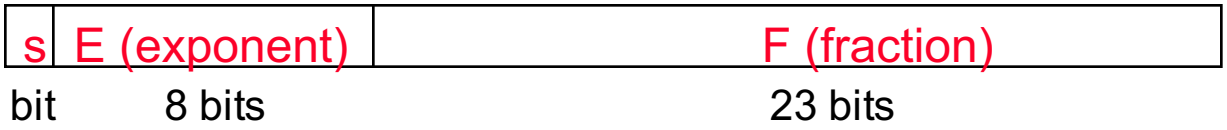
  4,600,000,000 or 4.6×10^9

  or the weight in kg of one a.m.u. (atomic mass unit)

  0.000000000000000000000000166 or 1.6×10^{-27}

  There is no way we can encode either of the above in a 32-bit integer.

- Floating point representation $(-1)^{sign} \times F \times 2^E$
  - Still have to fit everything in 32 bits (single precision)



  - The base (2, *not* 10) is hardwired in the design of the FPALU
  - More bits in the fraction (F) or the exponent (E) is a trade-off between precision (accuracy of the number) and range (size of the number)

Exception Events in Floating Point

- Overflow (floating point) happens when a positive exponent becomes too large to fit in the exponent field
- Underflow (floating point) happens when a negative exponent becomes too large to fit in the exponent field

- One way to reduce the chance of underflow or overflow is to offer another format that has a larger exponent field
  - Double precision – takes two MIPS words