CENG/CSCI 3420
Computer Organization and Design
Spring 2014

Lecture 03: Arithmetic for Computers

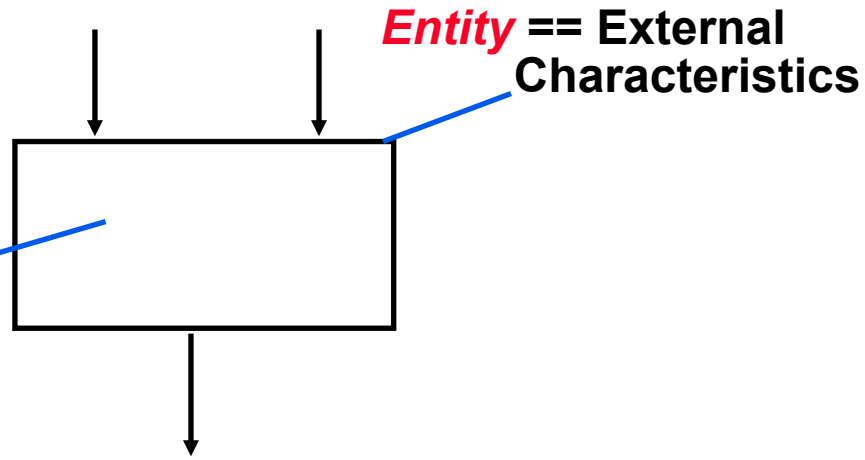
XU, Qiang 徐強

[Adapted from UC Berkeley's D. Patterson's and
from PSU's Mary J. Irwin's slides with additional credits to Y. Xie]

Review: VHDL

- Supports design, documentation, simulation & verification, and synthesis of hardware
- Allows integrated design at behavioral and structural levels
- Basic structure
  - Design entity-architecture descriptions
  - Time-based execution (discrete event simulation) model

Design Entity-Architecture ==
Hardware Component

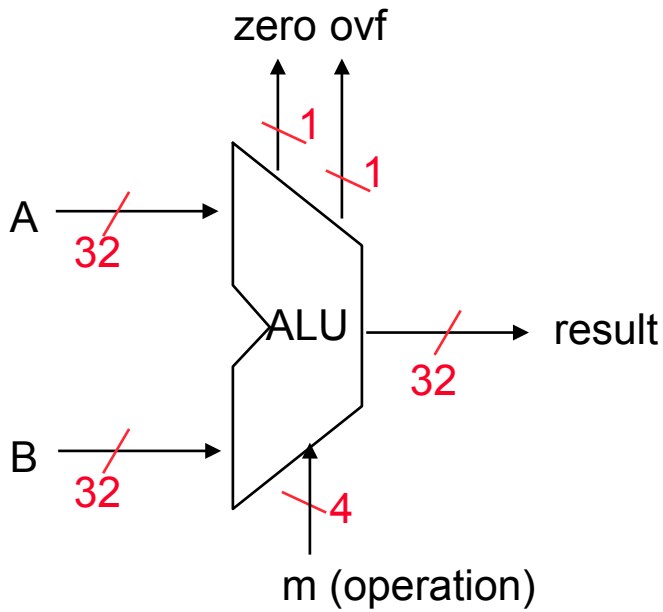


Review: Entity-Architecture Features

- Entity defines externally visible characteristics
  - Ports: channels of communication
    - signal names for inputs, outputs, clocks, control
  - Generic parameters: define class of components
    - timing characteristics, size (fan-in), fan-out
- Architecture defines the internal behavior or structure of the circuit
  - Declaration of internal signals
  - Description of behavior
    - collection of Concurrent Signal Assignment (CSA) statements (indicated by <=); can also model temporal behavior with the delay annotation
    - one or more processes containing CSAs and (sequential) variable assignment statements (indicated by :=)
  - Description of structure
    - interconnections of components; underlying behavioral models of each component must be specified

Arithmetic

- Where we've been
  - Abstractions
    - Instruction Set Architecture (ISA)
    - Assembly and machine language
- What's up ahead
  - Implementing the architecture (in VHDL)



ALU VHDL Representation

```
entity ALU is
  port(A, B:  in std_logic_vector (31 downto 0);
         m:  in std_logic_vector (3 downto 0);
         result: out std_logic_vector (31 downto 0);
         zero: out std_logic;
         ovf: out std_logic)
end ALU;

architecture process_behavior of ALU is
. . .
begin
    ALU: process(A, B, m)
    begin
        . . .
        result := A + B;
        . . .
    end process ALU;
end process_behavior;
```

Machine Number Representation

- Bits are just bits (have no inherent meaning)
  - conventions define the relationships between bits and numbers
- Binary numbers (base 2) - integers

  0000 -> 0001 -> 0010 -> 0011 -> 0100 -> 0101 -> . . .
  - in decimal from 0 to 2^n-1 for n bits
- Of course, it gets more complicated
  - storage locations (e.g., register file words) are finite, so have to worry about overflow (i.e., when the number is too big to fit into 32 bits)
  - have to be able to represent negative numbers, e.g., how do we specify -8 in

    ```
    addi   $sp, $sp, -8      #$sp = $sp - 8
    ```
  - in real systems have to provide for more than just integers, e.g., fractions and real numbers (and floating point) and alphanumeric (characters)

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

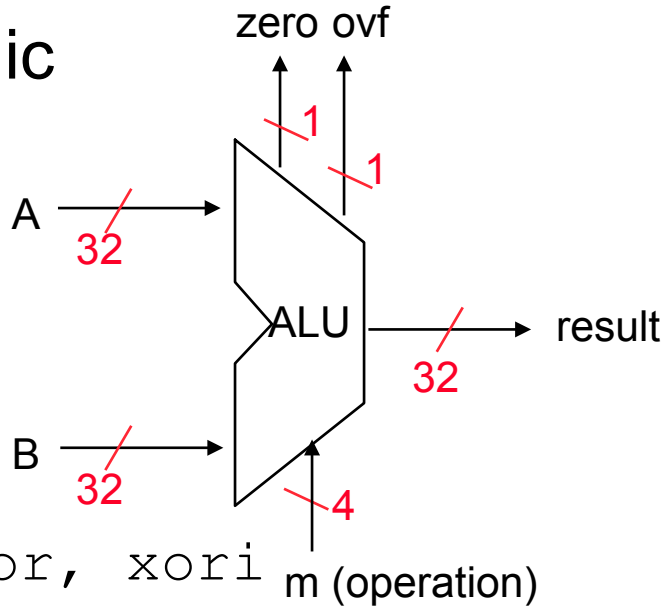
```
0010  -> 0000 0010
1010  -> 1111 1010
```

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

Design the MIPS Arithmetic Logic Unit (ALU)

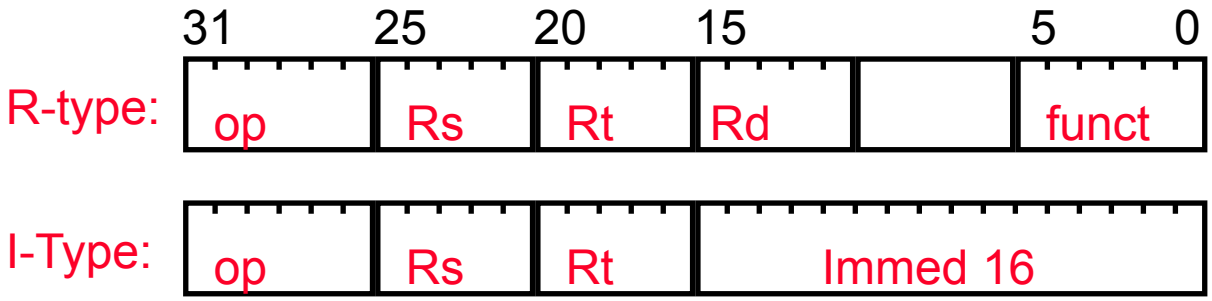
- Must support the Arithmetic/Logic operations of the ISA

```
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
```

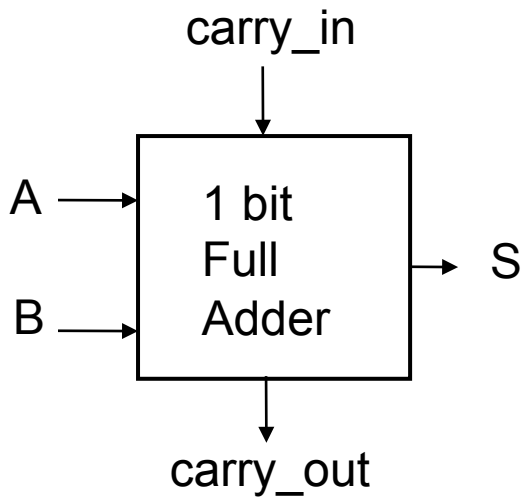
- Two's complement operations are easy
  - do subtraction by negating and then adding

```
  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

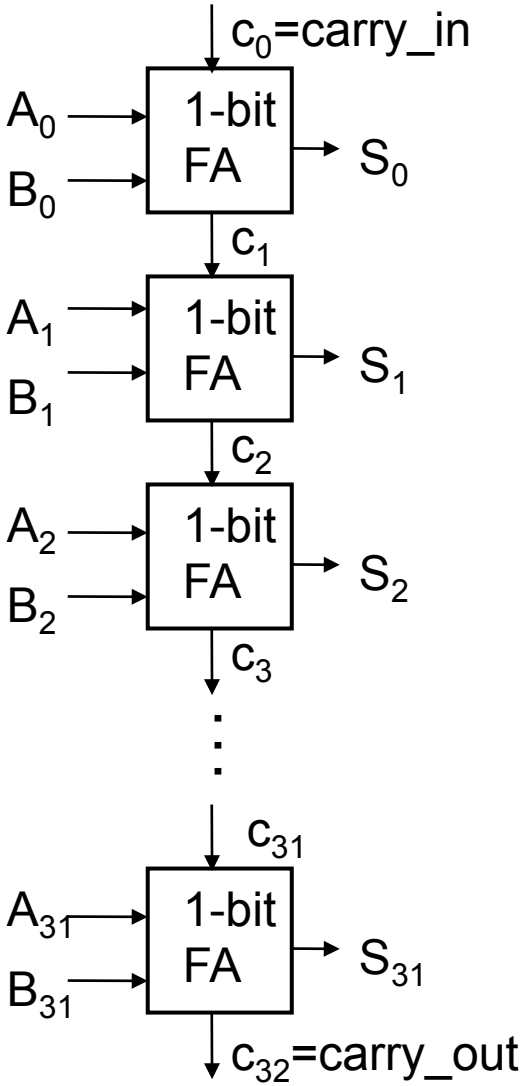
S = A xor B xor carry_in

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

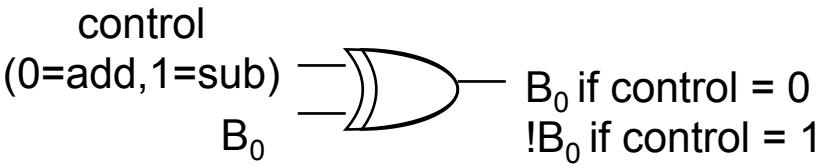


- Just connect the carry-out of the least significant bit FA to the carry-in of the next least significant bit and connect . . .

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

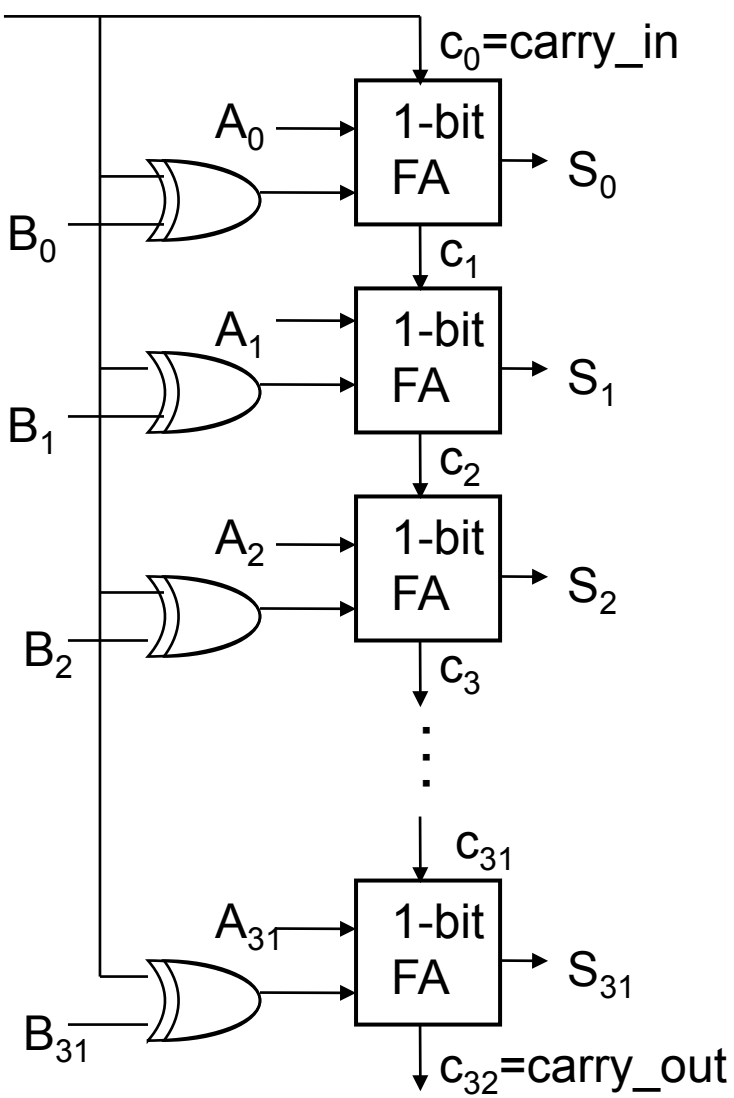
A 32-bit Ripple Carry Adder/Subtractor

- 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
```



Overflow Detection and Effects

- Overflow: the result is too large to represent in the number of bits allocated
- When adding operands with different signs, overflow cannot occur! Overflow occurs when
  - 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 overflow, an exception (interrupt) occurs
  - Control jumps to predefined address for exception
  - Interrupted address (address of instruction causing the overflow) is saved for possible resumption
- Don't always want to detect (interrupt on) overflow

New MIPS Instructions

Category	Instr	Op Code	Example	Meaning
Arithmetic (R & I format)	add unsigned	0 and 21	addu $s1, $s2, $s3	$s1 = $s2 + $s3
	sub unsigned	0 and 23	subu $s1, $s2, $s3	$s1 = $s2 - $s3
	add imm.unsigned	9	addiu $s1, $s2, 6	$s1 = $s2 + 6
Data Transfer	ld byte unsigned	24	lbu $s1, 20($s2)	$s1 = Mem($s2+20)
	ld half unsigned	25	lhu $s1, 20($s2)	$s1 = Mem($s2+20)
Cond. Branch (I & R format)	set on less than unsigned	0 and 2b	sltu $s1, $s2, $s3	if ($s2<$s3) $s1=1 else $s1=0
	set on less than imm unsigned	b	sltiu $s1, $s2, 6	if ($s2<6) $s1=1 else $s1=0

- Sign extend – `addi, addiu, slti, sltiu`
- Zero extend – `andi, ori, xori`
- Overflow detected – `add, addi, sub`

Minimal Implementation of a Full Adder

- Gate library: inverters, 2-input nands, or-and-inverters

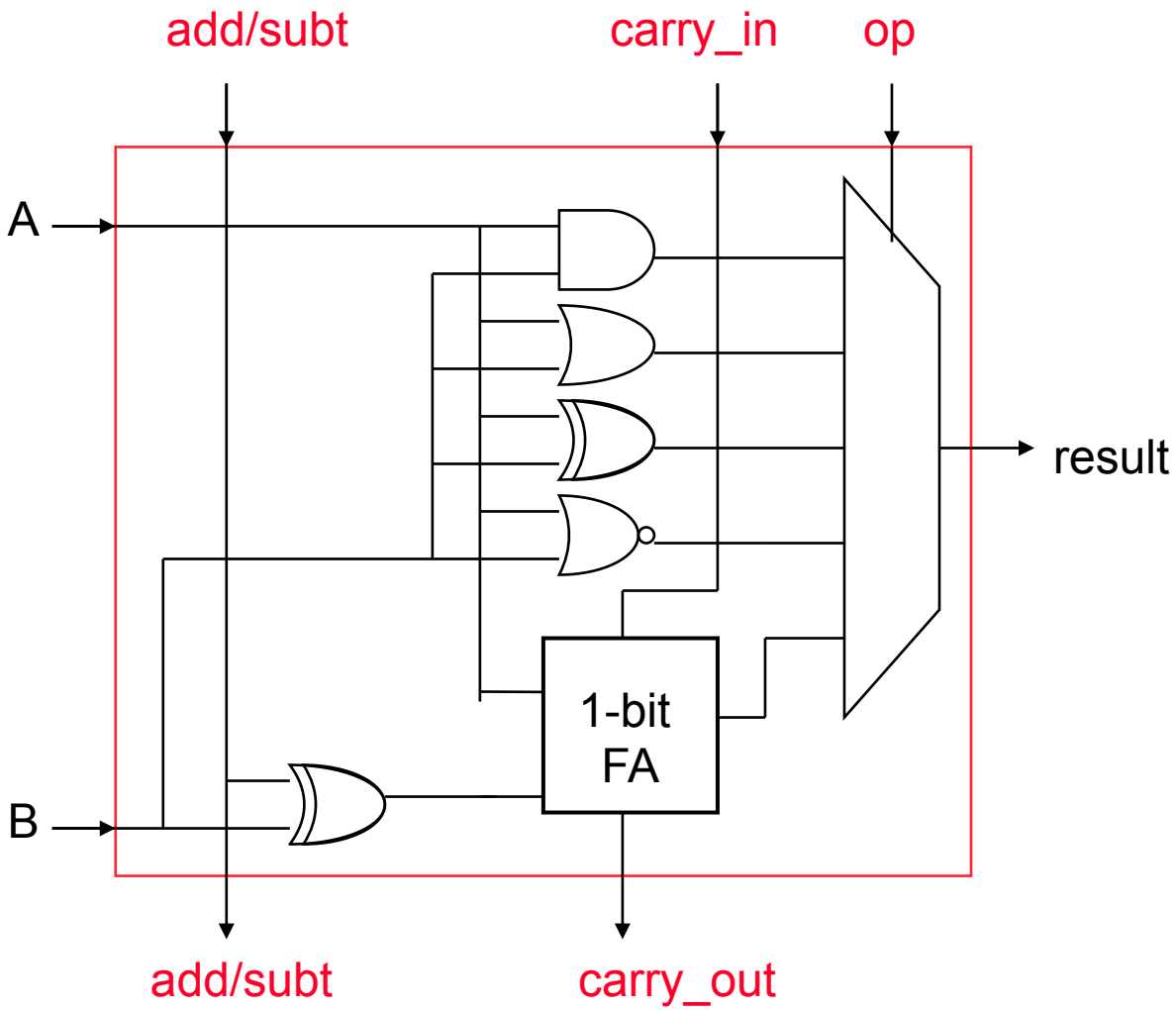
```
architecture concurrent_behavior of full_adder is
   signal t1, t2, t3, t4, t5: std_logic;
begin
      t1 <= not A after 1 ns;
      t2 <= not cin after 1 ns;
      t4 <= not((A or cin) and B) after 2 ns;
      t3 <= not((t1 or t2) and (A or cin)) after 2 ns;
      t5 <= t3 nand B after 2 ns;
      S <= not((B or t3) and t5) after 2 ns;
      cout <= not(t1 or t2) and t4) after 2 ns;
end concurrent_behavior;
```

- Can you create the equivalent schematic? Can you determine worst case delay (the worst case timing path through the circuit)?

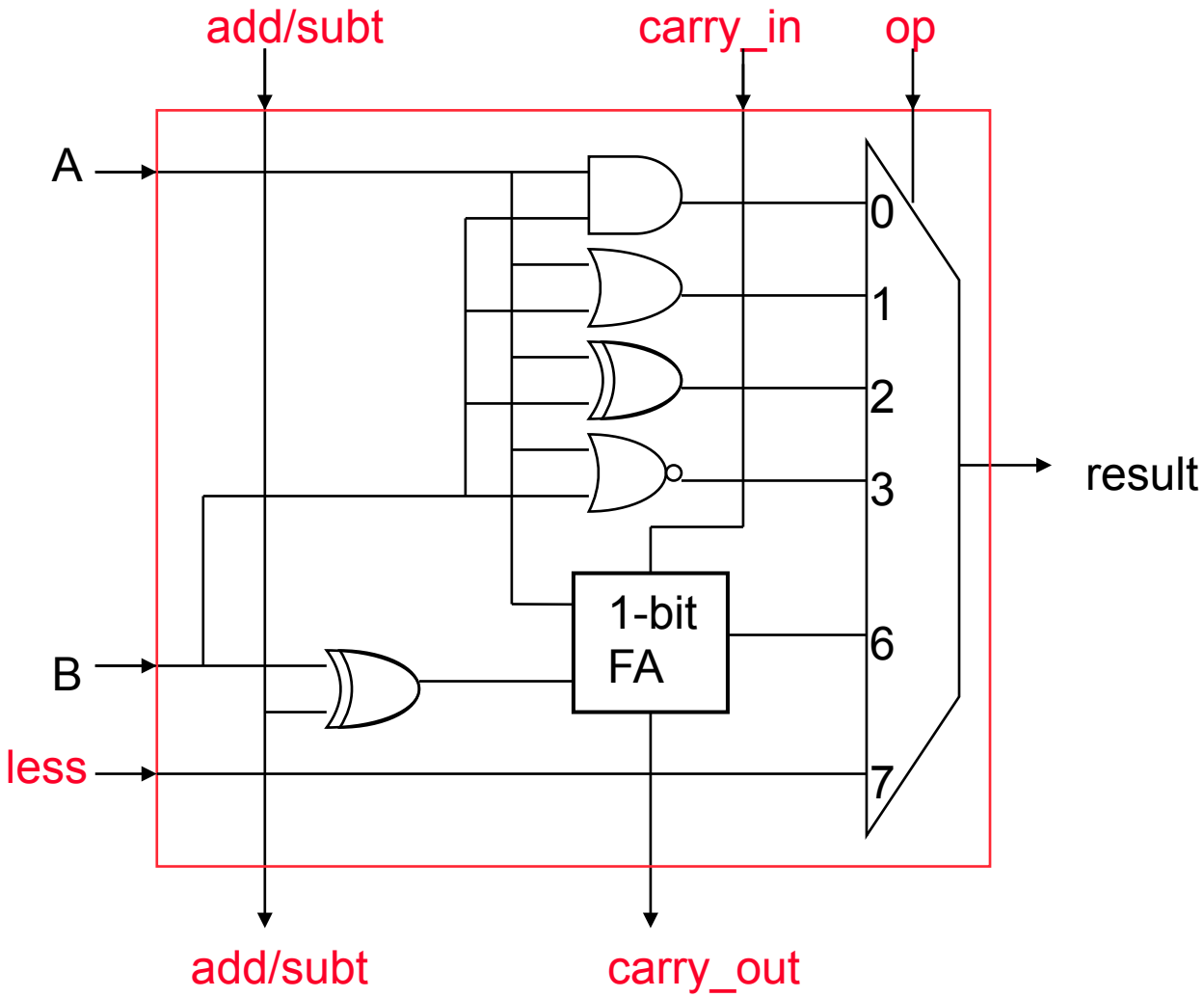
Tailoring the ALU to the MIPS ISA

- Also need to support the logic operations (`and,nor,or,xor`)
  - Bit wise operations (no carry operation involved)
  - Need a logic gate for each function and a mux to choose the output
- Also need to support the set-on-less-than instruction (`slt`)
  - Uses subtraction to determine if (a – b) < 0 (implies a < b)
- Also need to support test for equality (`bne, beq`)
  - Again use subtraction: (a - b) = 0 implies a = b
- Also need to add overflow detection hardware
  - overflow detection enabled only for `add, addi, sub`
- Immediates are sign extended outside the ALU with wiring (i.e., no logic needed)

A Simple ALU Cell with Logic Op Support

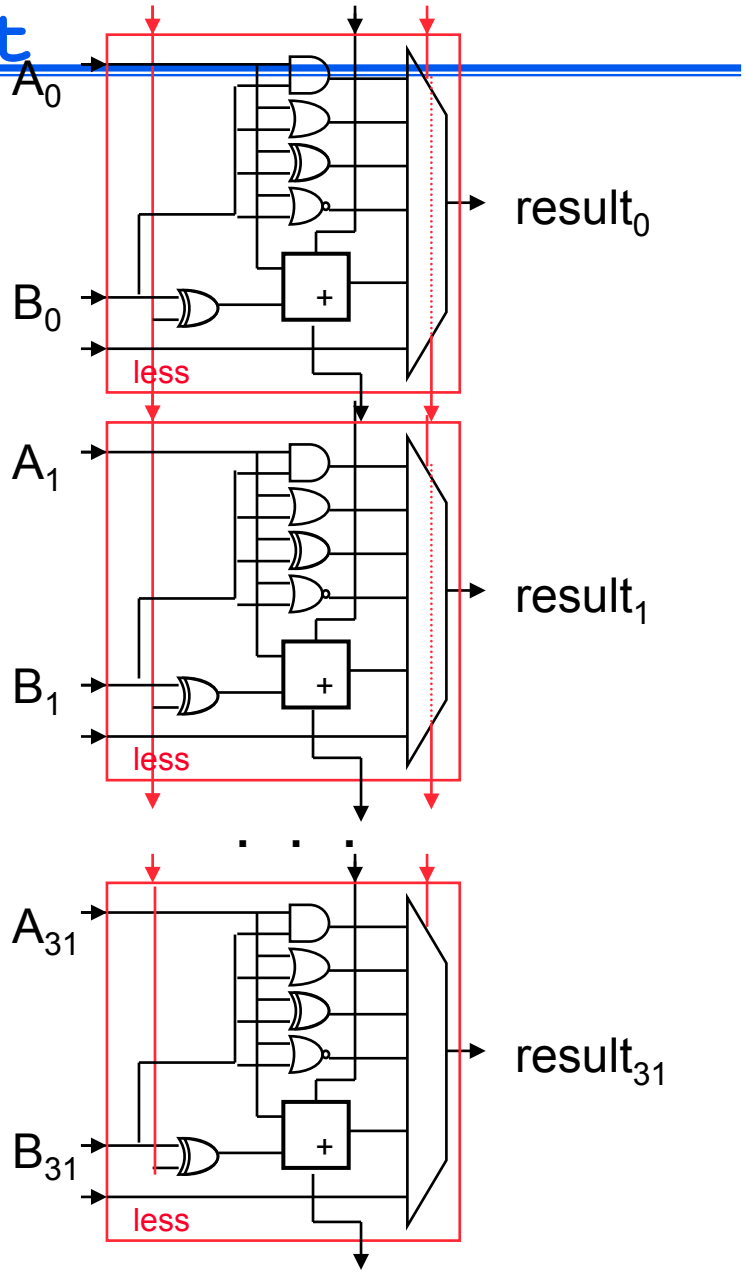


Modifying the ALU Cell for `slt`

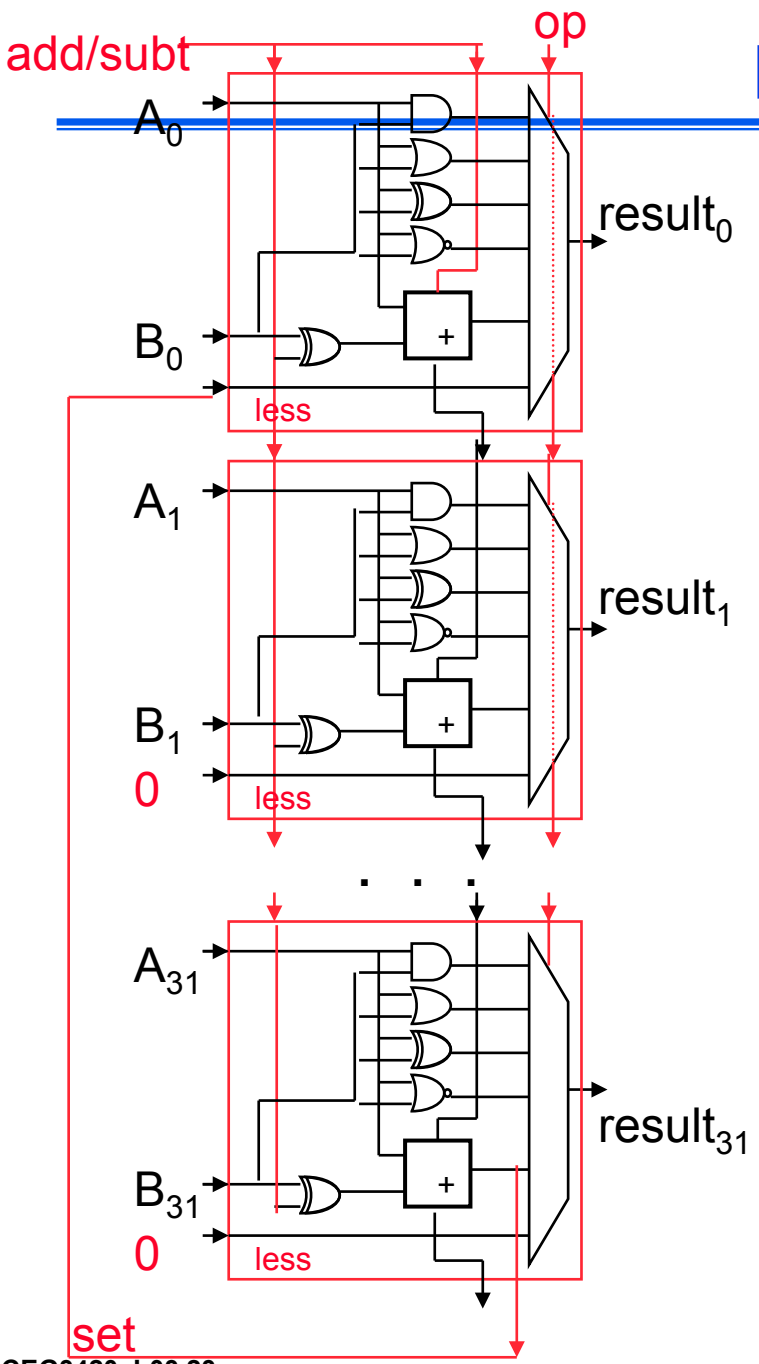


Modifying the ALU for `slt`

- First perform a subtraction
- Make the result 1 if the subtraction yields a negative result
- Make the result 0 if the subtraction yields a positive result
  - tie the most significant sum bit (sign bit) to the low order less input



Modifying the ALU for Zero

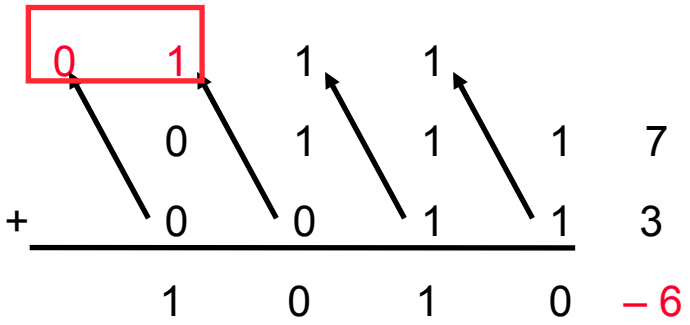


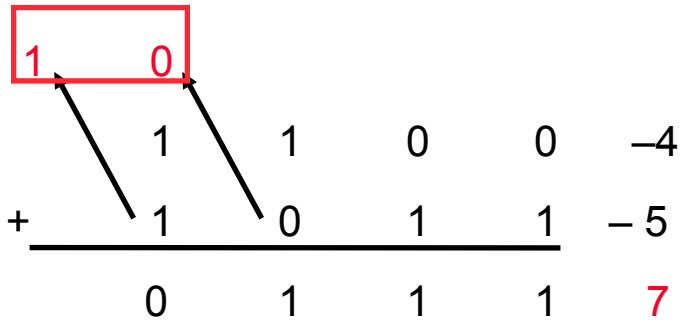
- First perform subtraction

- Insert additional logic to detect when all result bits are zero
  - Note zero is a 1 when result is all zeros

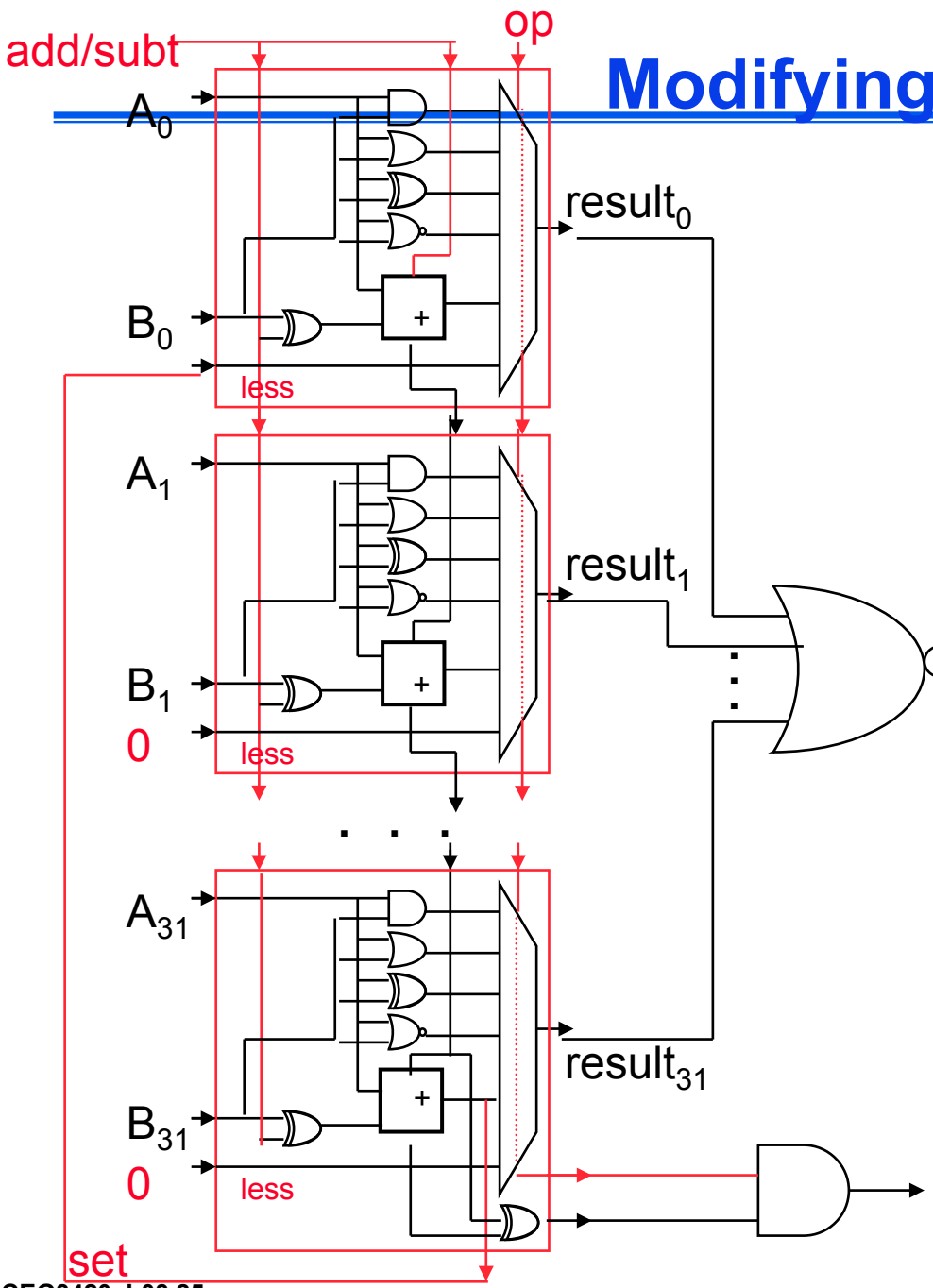
Overflow Detection

- 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





Modifying the ALU for Overflow

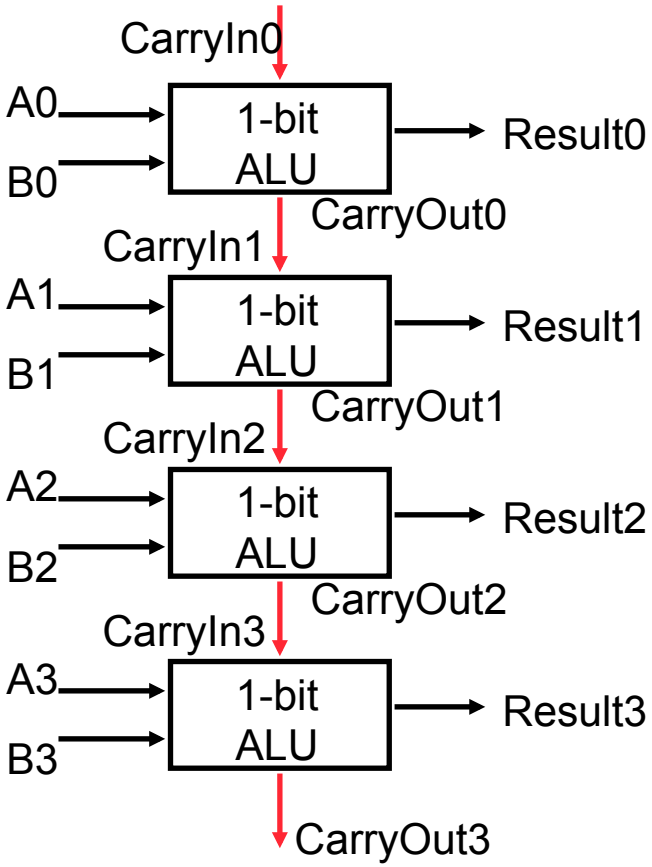


- Modify the most significant cell to determine overflow output setting

- Enable overflow bit setting for signed arithmetic (`add`, `addi`, `sub`)

But What about Performance?

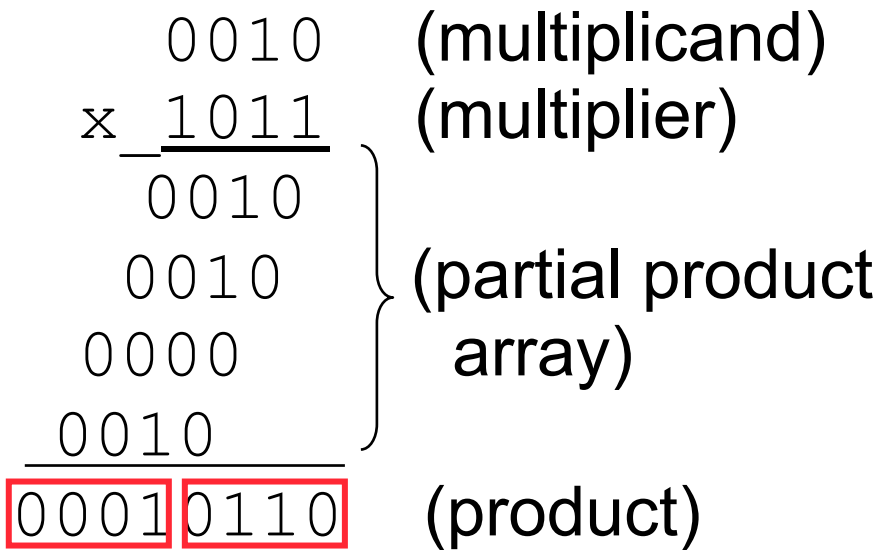
- Critical path of n-bit ripple-carry adder is n*CP



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

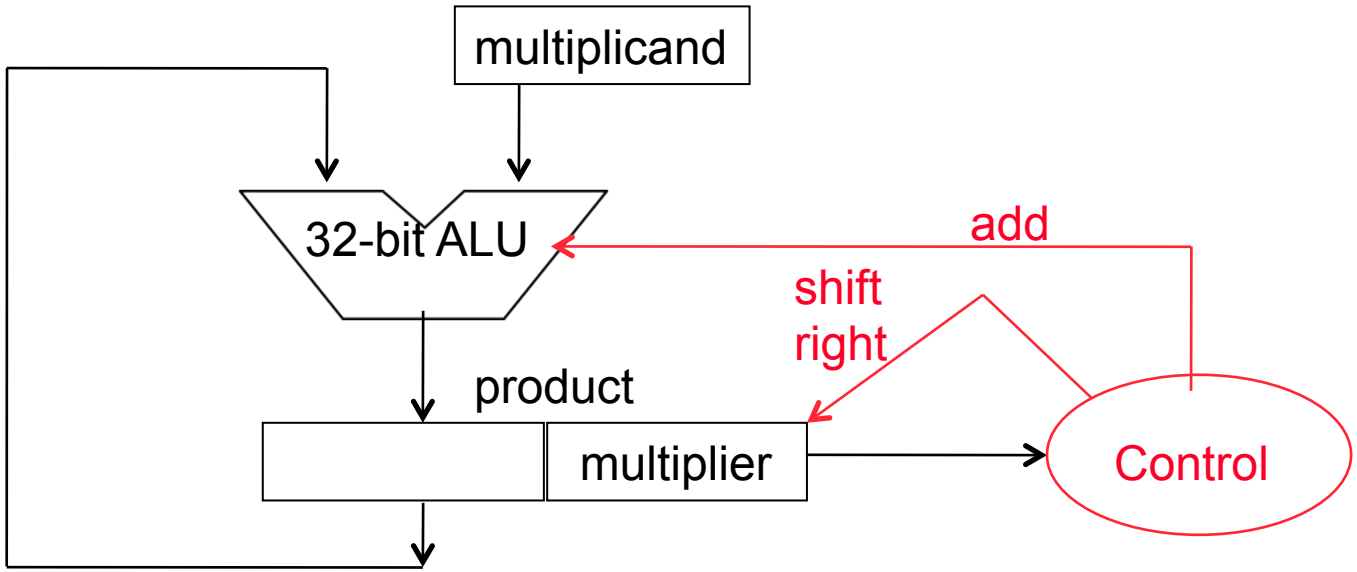
Multiplication

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

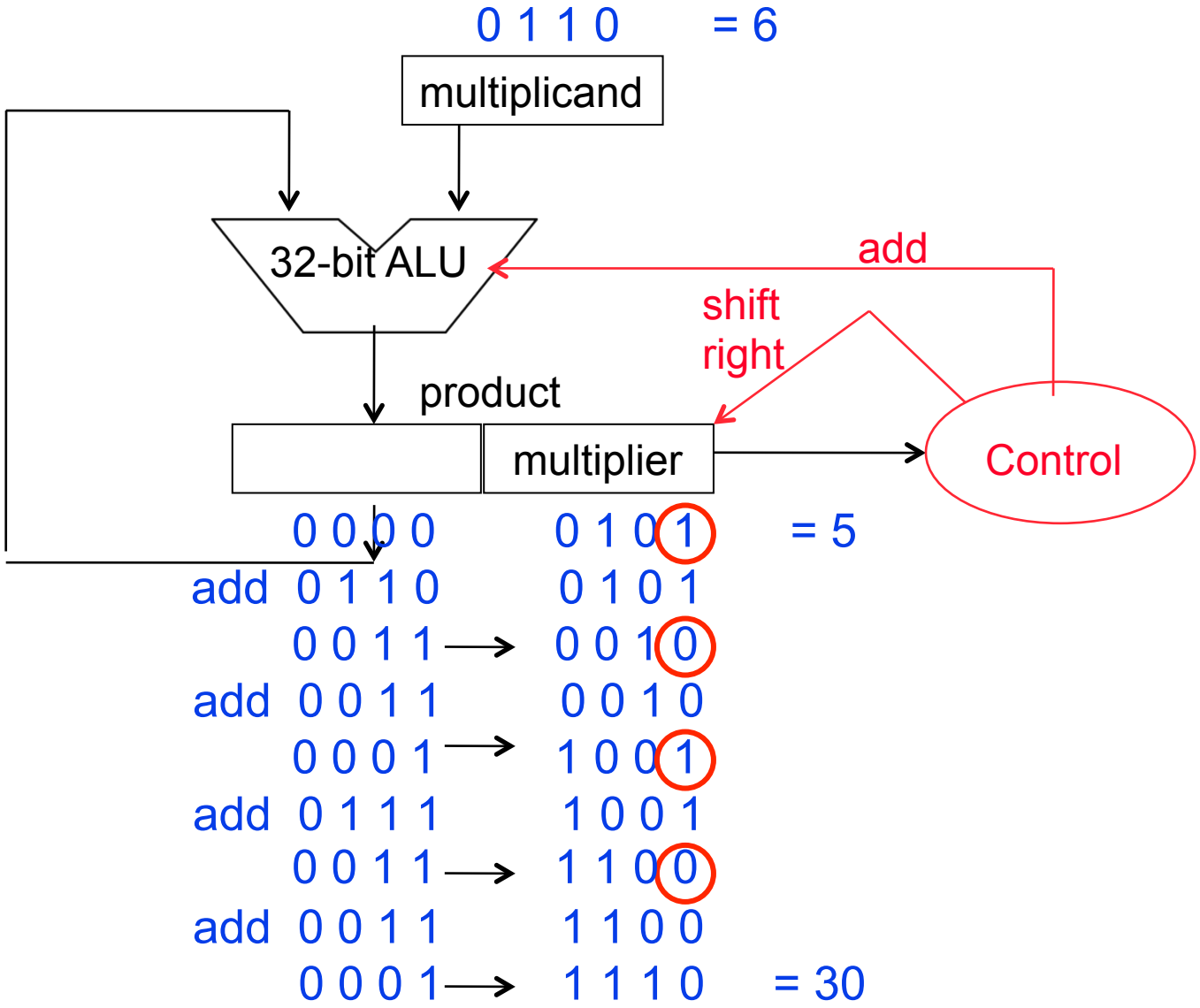


- Double precision product produced
- More time and more area to compute

Add and Right Shift Multiplier Hardware



Add and Right Shift Multiplier Hardware



MIPS Multiply Instruction

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

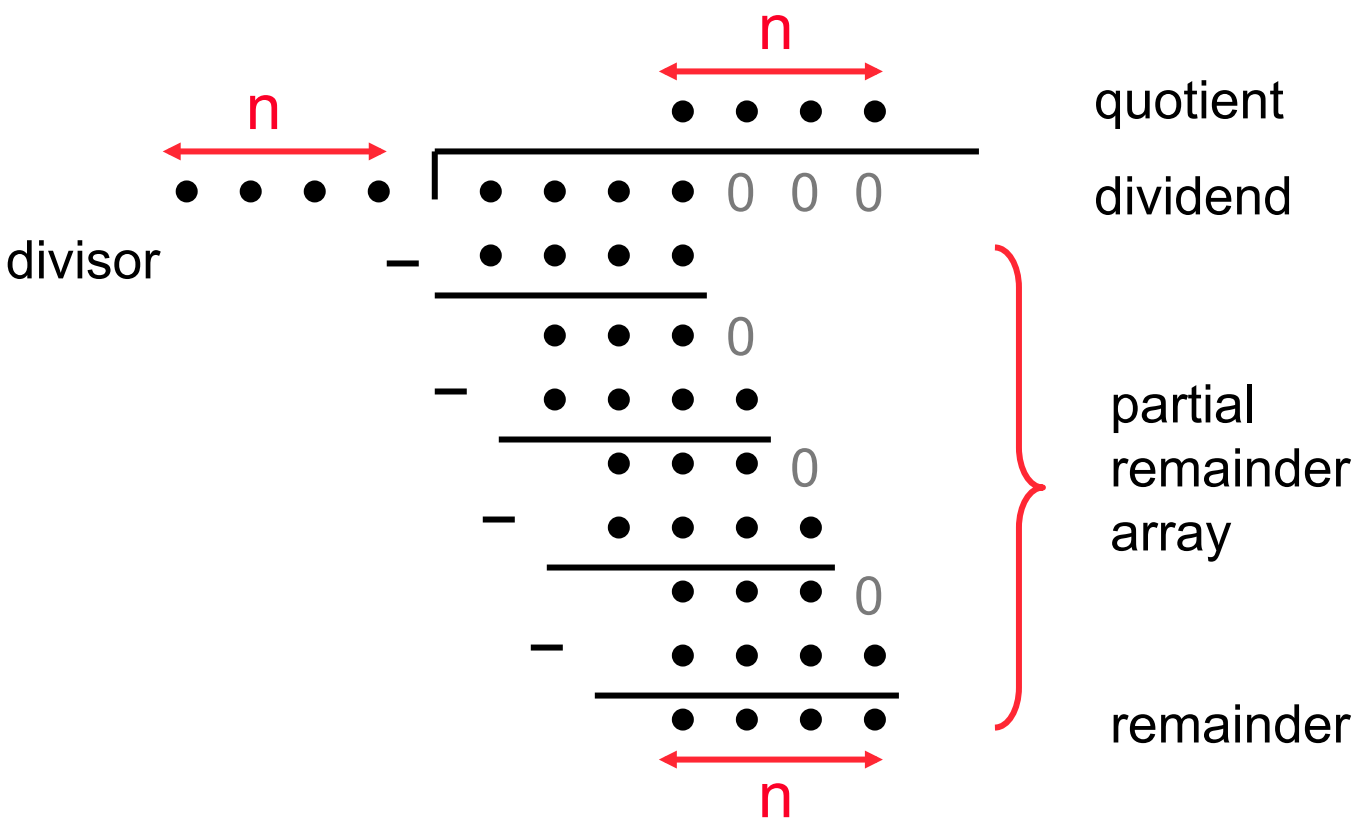
```
mult    $s0, $s1     # hi||lo = $s0 * $s1
```

  - 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

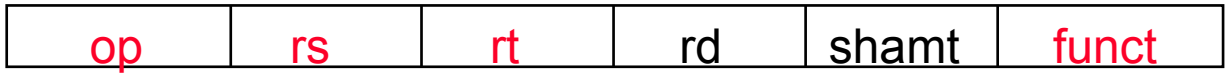
- Division is just a *bunch* of quotient digit guesses and left shifts and subtracts



MIPS Divide Instruction

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

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



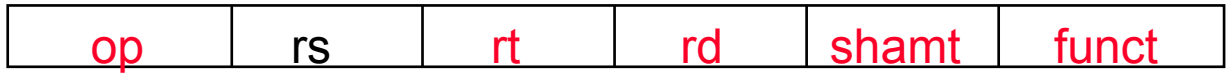
  - Instructions `mflo rd` and `mfhi rd` are provided to move the quotient and reminder to (user accessible) registers in the register file

- 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.

Shift Operations

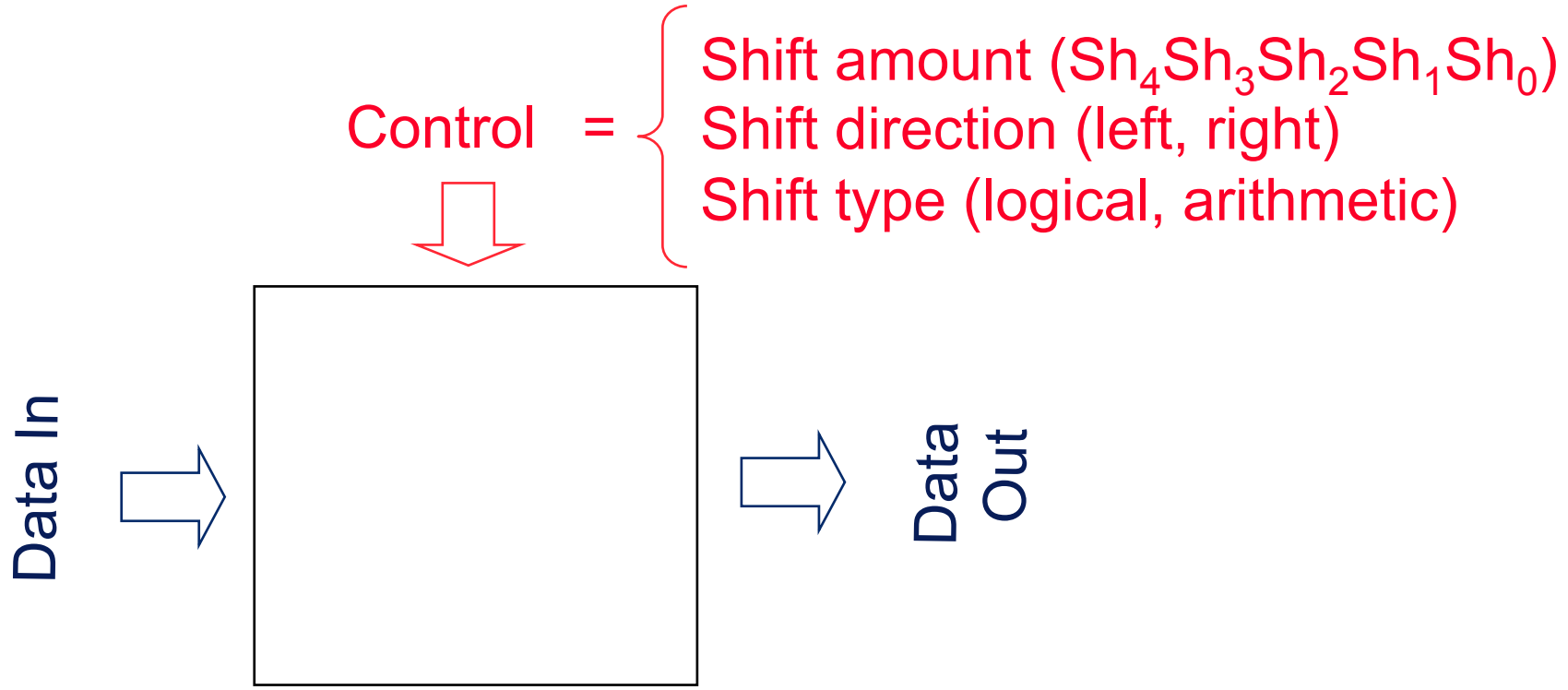
- Shifts move all the bits in a word left or right

```
sll  $t2, $s0, 8 #$t2 = $s0 << 8 bits
srl  $t2, $s0, 8 #$t2 = $s0 >> 8 bits
sra  $t2, $s0, 8 #$t2 = $s0 >> 8 bits
```



- Notice that a 5-bit shamt field is enough to shift a 32-bit value $2^5 - 1$ or 31 bit positions
- Logical shifts fill with zeros, arithmetic left shifts fill with the sign bit
- The shift operation is implemented by hardware separate from the ALU
  - using a barrel shifter (which would takes lots of gates in discrete logic, but is pretty easy to implement in VLSI)

Parallel Programmable Shifters



Wrap-Up

- We can build an ALU to support the MIPS ISA
  - we can efficiently perform subtraction using two's complement
  - we can replicate a 1-bit ALU to produce a 32-bit ALU
- Important points about hardware
  - all of the gates are always working (concurrent)
  - the speed of a gate is affected by the number of inputs to the gate (fan-in) and the number of gates that the output is connected to (fan-out)
  - the speed of a circuit is affected by the speed of and number of gates in series (on the "critical path" or the "number of levels of logic") and the length of wires interconnecting the gates
- Our primary focus is comprehension, however,
  - clever changes to organization can improve performance (similar to using better algorithms in software)

Next Lecture and Reminders

- Next lecture (possibly next Monday 6:30-9:15pm)
  - Floating-point instructions
  - MIPS single-cycle implementation
- Reminders
  - Group registration soon