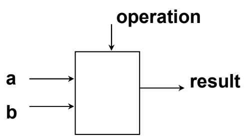
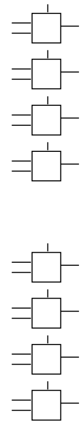
An ALU (arithmetic logic unit)

- Let's build an ALU to support the `andi` and `ori` instructions
  - we'll just build a 1 bit ALU, and use 32 of them



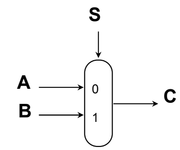
op	a	b	res



- Possible Implementation (sum-of-products):

Review: The Multiplexor

- Selects one of the inputs to be the output, based on a control input

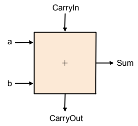


note: we call this a 2-input mux even though it has 3 inputs!

- Lets build our ALU using a MUX:

Different Implementations

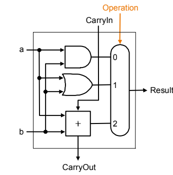
- Not easy to decide the "best" way to build something
  - Don't want too many inputs to a single gate
  - Don't want to have to go through too many gates
  - for our purposes, ease of comprehension is important
- Let's look at a 1-bit ALU for addition:

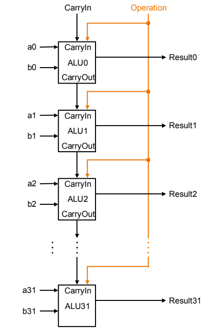


$$c_{out} = a\ b + a\ c_{in} + b\ c_{in}$$
$$sum = a \text{ xor } b \text{ xor } c_{in}$$

- How could we build a 1-bit ALU for add, and, and or?
- How could we build a 32-bit ALU?

Building a 32 bit ALU

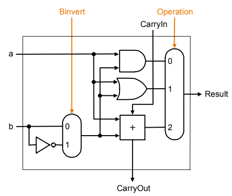




What about subtraction (a – b) ?

- **Two's complement approach: just negate b and add.**
- **How do we negate?**

- **A very clever solution:**

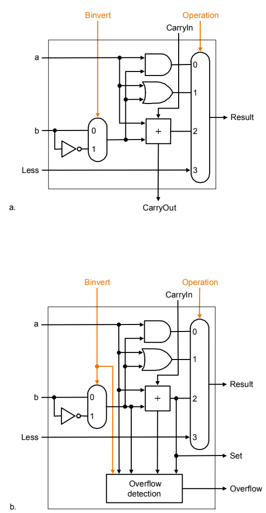


Tailoring the ALU to the MIPS

- **Need to support the set-on-less-than instruction (slt)**
  - **remember: slt is an arithmetic instruction**
  - **produces a 1 if rs < rt and 0 otherwise**
  - **use subtraction: (a-b) < 0 implies a < b**
- **Need to support test for equality (beq $t5, $t6, $t7)**
  - **use subtraction: (a-b) = 0 implies a = b**

Supporting slt

- **Can we figure out the idea?**

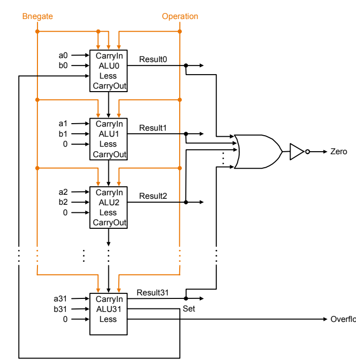


Test for equality

- **Notice control lines:**

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

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

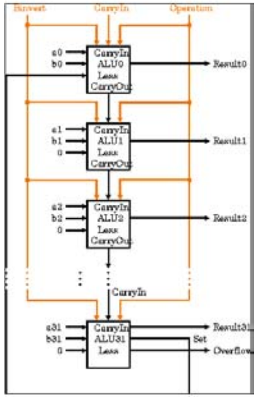


Conclusion

- **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**
  - **we can replicate a 1-bit ALU to produce a 32-bit ALU**
- **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")**
- **Our primary focus: comprehension, however,**
  - **Clever changes to organization can improve performance (similar to using better algorithms in software)**
  - **we'll look at two examples for addition and multiplication**

A 32-bit ALU

- A Ripple carry ALU
- Two bits decide operation
  - Add/Sub
  - AND
  - OR
  - LESS
- 1 bit decide add/sub operation
- A carry in bit
- Bit 31 generates overflow and set bit



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 and sum-of-products**

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

$c_1 = b_0c_0 + a_0c_0 + a_0b_0$

$c_2 = b_1c_1 + a_1c_1 + a_1b_1$ $\quad c_2 =$

$c_3 = b_2c_2 + a_2c_2 + a_2b_2$ $\quad c_3 =$

$c_4 = b_3c_3 + a_3c_3 + a_3b_3$ $\quad c_4 =$

Not feasible! Why?

Carry-look-ahead adder

- **An approach in-between our two extremes**
- **Motivation:**
  - **If we didn't know the value of carry-in, what could we do?**
  - **When would we always generate a carry?** $g_i = a_i\, b_i$
  - **When would we propagate the carry?** $p_i = a_i + b_i$
- **Did we get rid of the ripple?**

$c_1 = g_0 + p_0c_0$

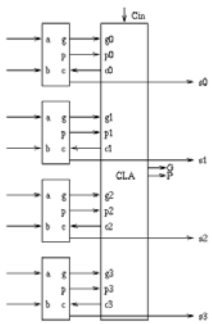
$c_2 = g_1 + p_1c_1$ $\quad c_2 = g_1 + p_1g_0 + p_1p_0c_0$

$c_3 = g_2 + p_2c_2$ $\quad c_3 = g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0$

$c_4 = g_3 + p_3c_3$ $\quad c_4 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_0$

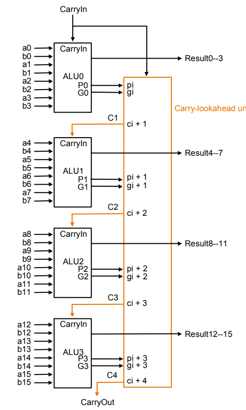
Feasible! Why?

A 4-bit carry look-ahead adder



- Generate g and p term for each bit
- Use g's, p's and carry in to generate all C's
- Also use them to generate block G and P
- CLA principle can be used recursively

Use principle to build bigger adders



- A 16 bit adder uses four 4-bit adders
- It takes block g and p terms and cin to generate block carry bits out
- Block carries are used to generate bit carries
  - could use ripple carry of 4-bit CLA adders
  - Better: use the CLA principle again!

Delays in carry look-ahead adders

- 4-Bit case
  - Generation of g and p: 1 gate delay
  - Generation of carries (and G and P): 2 more gate delay
  - Generation of sum: 1 more gate delay
- 16-Bit case
  - Generation of g and p: 1 gate delay
  - Generation of block G and P: 2 more gate delay
  - Generation of block carries: 2 more gate delay
  - Generation of bit carries: 2 more gate delay
  - Generation of sum: 1 more gate delay
- 64-Bit case
  - 12 gate delays

Multiplication

- More complicated than addition
  - accomplished via shifting and addition
- More time and more area
- Let's look at 3 versions based on grade school algorithm

```
 01010010   (multiplicand)
x01101101   (multiplier)
```

- Negative numbers: convert and multiply
- Use other better techniques like Booth's encoding

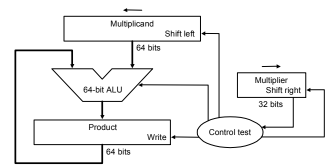
Multiplication

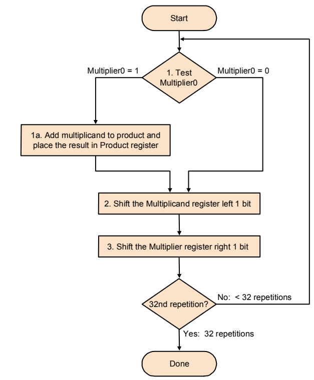
```
               01010010  (multiplicand)
              x01101101  (multiplier)
               00000000
               01010010   x1
              01010010
              000000000   x0
             001010010
             0101001000   x1
            0110011010
            01010010000   x1
           10000101010
           000000000000   x0
          010000101010
          0101001000000   x1
         0111001101010
         01010010000000   x1
        10001011101010
        000000000000000   x0
       0010001011101010
```

```
               01010010  (multiplicand)
              x01101101  (multiplier)
               00000000
               01010010   x1
                01010010
               000000000   x0
                001010010
               0101001000   x1
                0110011010
               01010010000   x1
                10000101010
               000000000000   x0
                010000101010
               0101001000000   x1
                0111001101010
               01010010000000   x1
                10001011101010
               000000000000000   x0
              0010001011101010
```

17

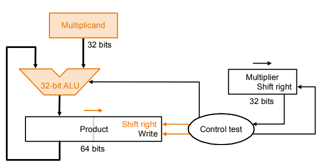
Multiplication: Implementation

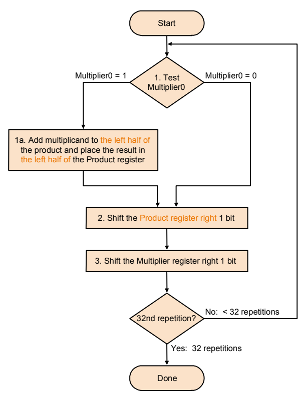




18

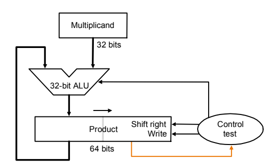
Second Version

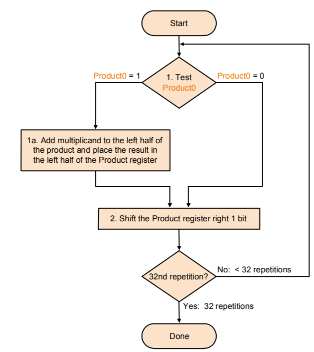




19

Final Version





20

Multiplication Example

Itera-tion	multi-plicand	Orignal algorithm	
		Step	Product
0	0010	Initial values	0000 0110
1	0010	1:0 ⇒ no operation	0000 0110
	0010	2: Shift right Product	0000 0011
2	0010	1a:1⇒ prod = Prod + Mcand	0010 0011
	0010	2: Shift right Product	0001 0001
3	0010	1a:1⇒ prod = Prod + Mcand	0011 0001
	0010	2: Shift right Product	0001 1000
4	0010	1:0 ⇒ no operation	0001 1000
	0010	2: Shift right Product	0000 1100

Signed Multiplication

- Let Multiplier be Q[n-1:0], multiplicand be M[n-1:0]
- Let F = 0 (shift flag)
- Let result A[n-1:0] = 0….00
- For n-1 steps do
  - A[n-1:0] = A[n-1:0] + M[n-1:0] x Q[0] /* add partial product */
  - F<= F .or. (M[n-1] .and. Q[0]) /* determine shift bit */
  - Shift A and Q with F, i.e.,
  - A[n-2:0] = A[n-1:1]; A[n-1]=F; Q[n-1]=A[0]; Q[n-2:0]=Q[n-1:1]
- Do the correction step
  - A[n-1:0] = A[n-1:0] - M[n-1:0] x Q[0] /* subtract partial product */
  - Shift A and Q while retaining A[n-1]
  - This works in all cases excepts when both operands are 10..00

Booth's Encoding

- Numbers can be represented using three symbols, 1, 0, and -1
- Let us consider -1 in 8 bits
  - One representation is 1 1 1 1 1 1 1 1
  - Another possible one 0 0 0 0 0 0 0 -1
- Another example +14
  - One representation is 0 0 0 0 1 1 1 0
  - Another possible one 0 0 0 1 0 0 -1 0
- We do not explicitly store the sequence
- Look for transition from previous bit to next bit
  - 0 to 0 is 0; 0 to 1 is -1; 1 to 1 is 0; and 1 to 0 is 1
- Multiplication by 1, 0, and -1 can be easily done
- Add all partial results to get the final answer

Using Booth's Encoding for Multiplication

- Convert a binary string in Booth's encoded string
- Multiply by two bits at a time
- For n bit by n-bit multiplication, n/2 partial product
- Partial products are signed and obtained by multiplying the multiplicand by 0, +1, -1, +2, and -2 (all achieved by shift)
- Add partial products to obtain the final result
- Example, multiply 0111 (+7) by 1010 (-6)
- Booths encoding of 1010 is -1 +1 -1 0
- With 2-bit groupings, multiplication needs to be carried by -1 and -2
-

```
1 1 1 1 0 0 1 0   (multiplication by -2)
1 1 1 0 0 1 0 0   (multiplication by -1 and shift by 2 positions)
```

- Add the two partial products to get 11010110 (-42) as result