Computer Organization and Design

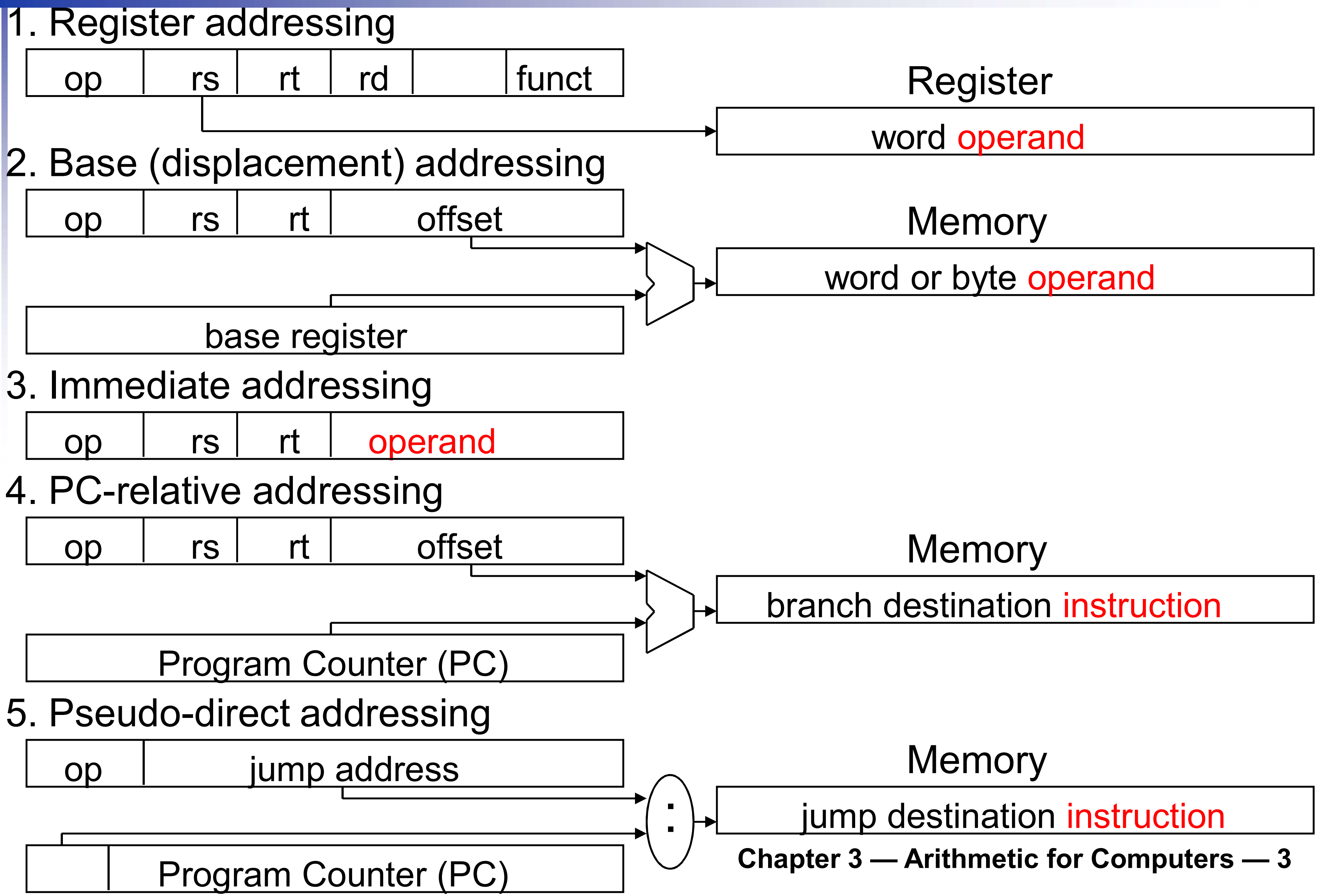
Arithmetic for Computers

Jiang Zhong
zhongjiang@cqu.edu.cn

Review: MIPS (RISC) Design Principles

- Simplicity favors regularity
  - fixed size instructions
  - small number of instruction formats
  - opcode always the first 6 bits
- Smaller is faster
  - limited instruction set
  - limited number of registers in register file
  - limited number of addressing modes
- Make the common case fast
  - arithmetic operands from the register file (load-store machine)
  - allow instructions to contain immediate operands
- Good design demands good compromises
  - three instruction formats

Review: MIPS Addressing Modes



Review: Number Representations

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

```
0000 0000 0000 0000 0000 0000 0000 0000two = 0ten
0000 0000 0000 0000 0000 0000 0000 0001two = + 1ten
...

0111 1111 1111 1111 1111 1111 1111 1110two = + 2,147,483,646ten
0111 1111 1111 1111 1111 1111 1111 1111two = + 2,147,483,647ten
1000 0000 0000 0000 0000 0000 0000 0000two = – 2,147,483,648ten
1000 0000 0000 0000 0000 0000 0000 0001two = – 2,147,483,647ten
...

1111 1111 1111 1111 1111 1111 1111 1110two = – 2ten
1111 1111 1111 1111 1111 1111 1111 1111two = – 1ten
```

maxint

minint

MSB

LSB

- Converting <32-bit values into 32-bit values
  - copy the most significant bit (the sign bit) into the "empty" bits

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

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

主观题 5分

设置

请分析为什么**任意逻辑函数**都可以使用与、或、非门电路来构成?

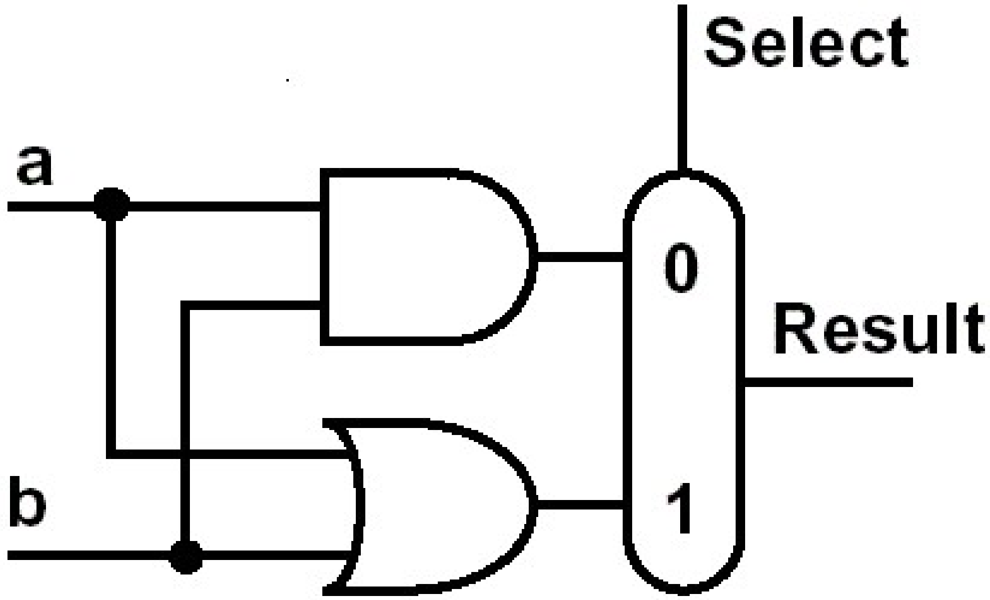
假设门电路的扇入/扇出没有限制的话，任意逻辑函数函数均可以由**几级**的逻辑电路来实现?

正常使用主观题需2.0以上版本雨课堂

作答

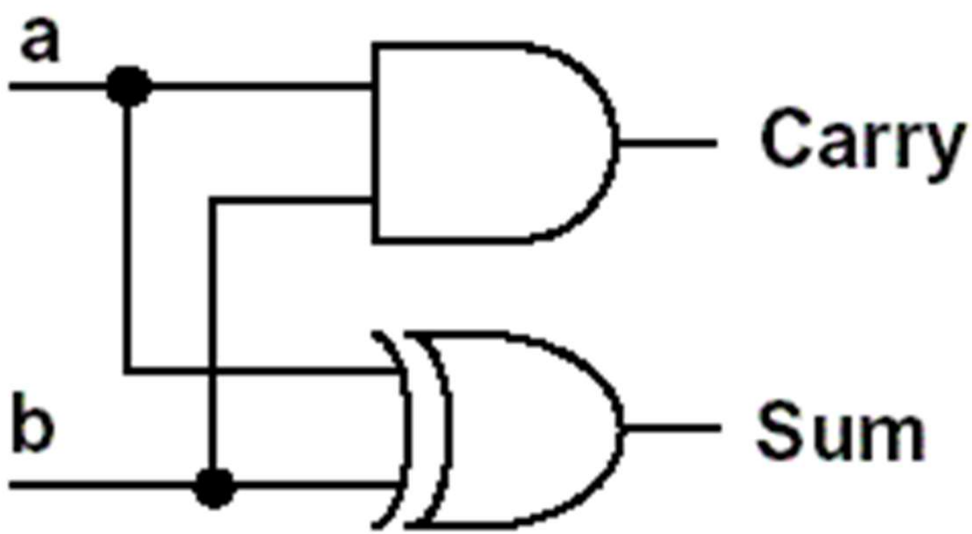
Constructing an ALU

- Step by step: build a single bit ALU and expand it to the desired width
- First function: logic AND and OR



A half adder

- Sum = $\bar{a}\,b + a\,\bar{b}$
- Carry = a b



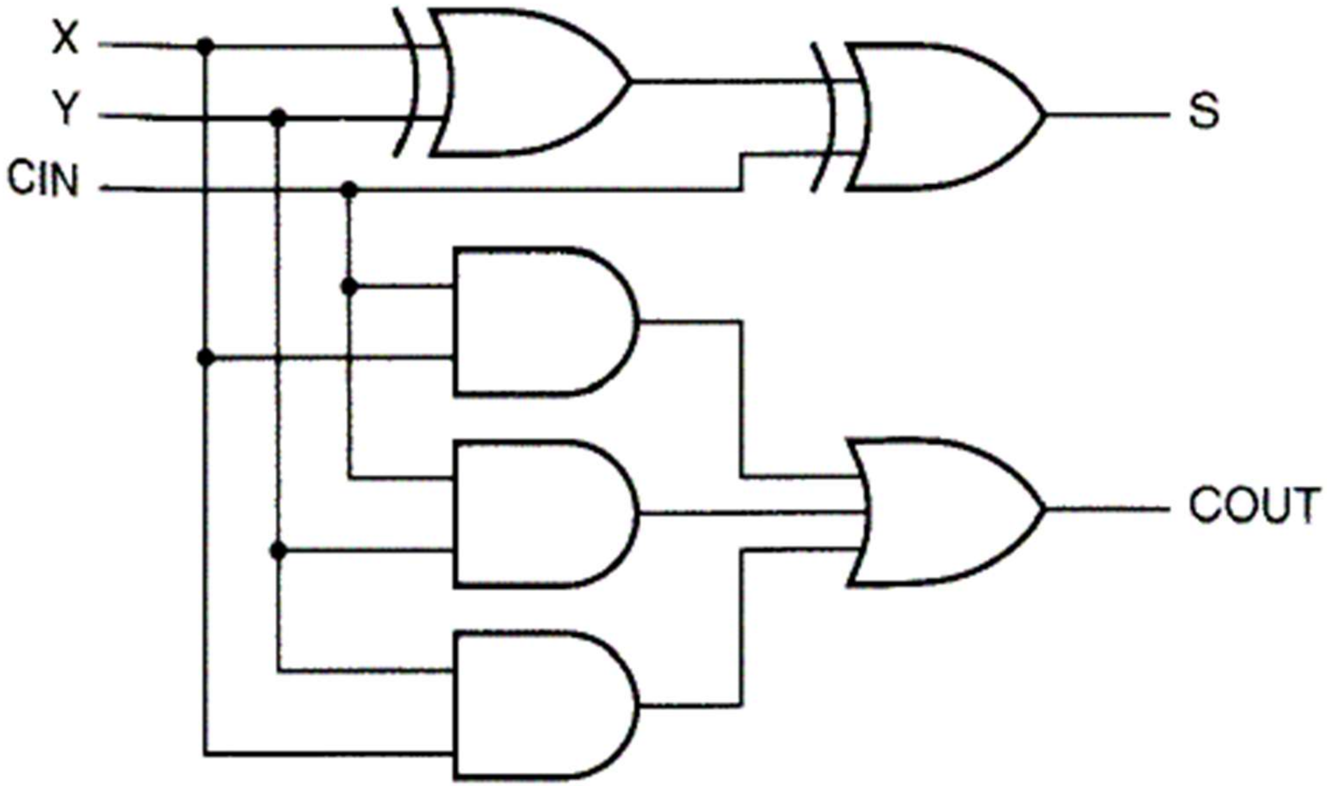
A full adder

- Accepts a carry in
- Sum = A xor B xor CarryIn
- CarryOut = B CarryIn + A CarryIn + A B

Inputs			Outputs		Comments (two)
A	B	CarryIn	CarryOut	Sum	
0	0	0	0	0	0+0+0=00
0	0	1	0	1	0+0+1=01
0	1	0	0	1	0+1+0=01
0	1	1	1	0	0+1+1=10
1	0	0	0	1	1+0+0=01
1	0	1	1	0	1+0+1=10
1	1	0	1	0	1+1+0=10
1	1	1	1	1	1+1+1=11

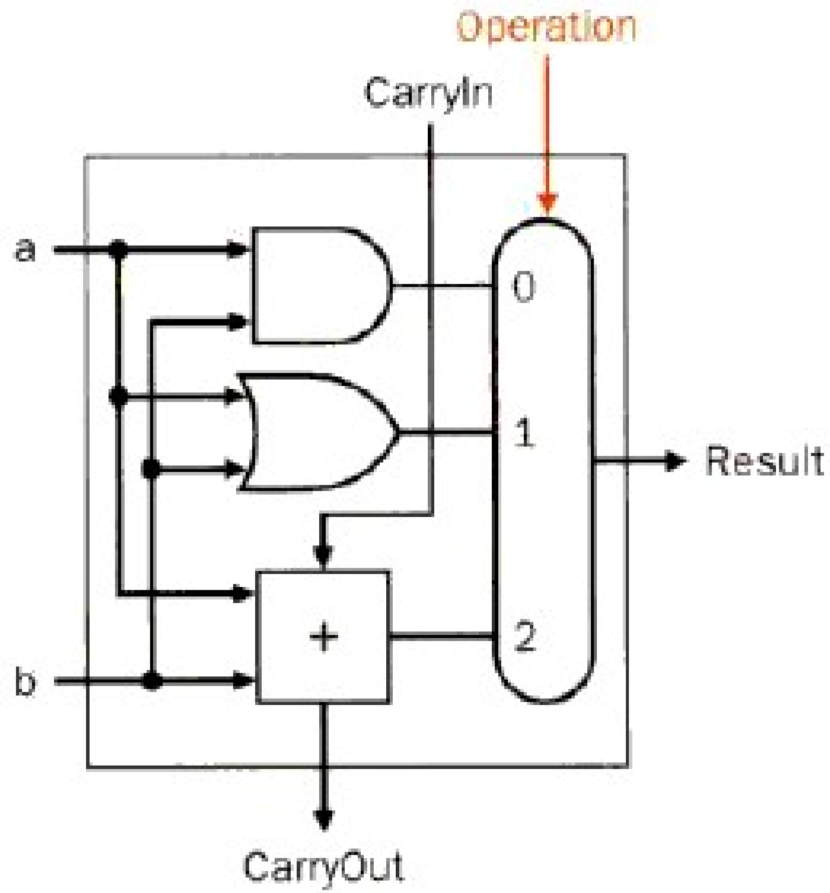
Full adder

- Full adder in 2-level design



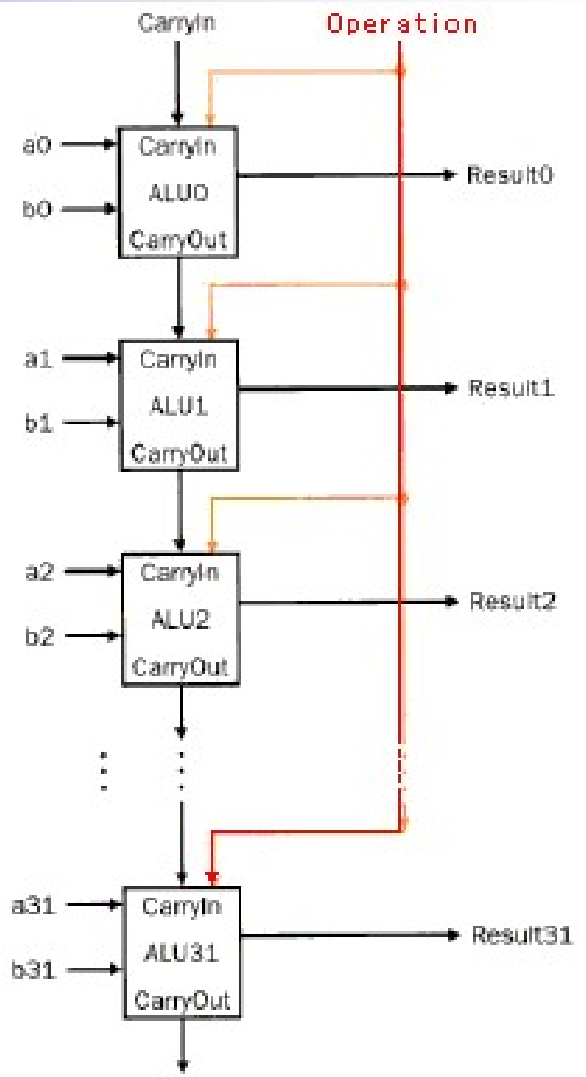
1 bit ALU

- ALU
  - AND
  - OR
  - ADD
- Cascade Element



Basic 32 bit ALU

- Inputs parallel
- Carry is cascaded
- Ripple carry adder
- Slow, but simple
- 1st Carry In = 0

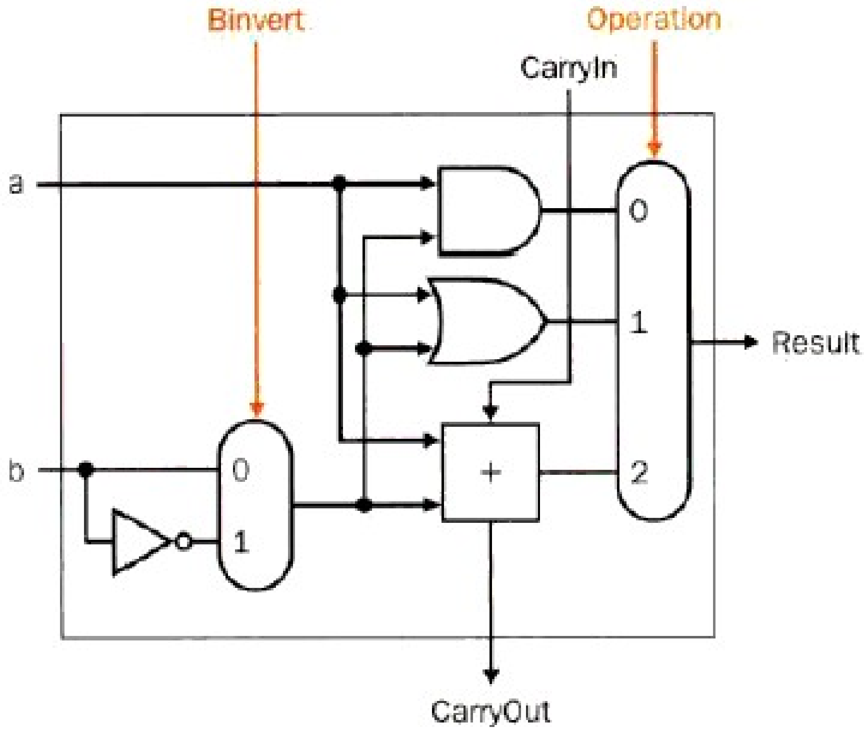


Extended 1 bit ALU

- Subtraction

  a - b

  - Inverting b
  - 1st CarryIn= 1

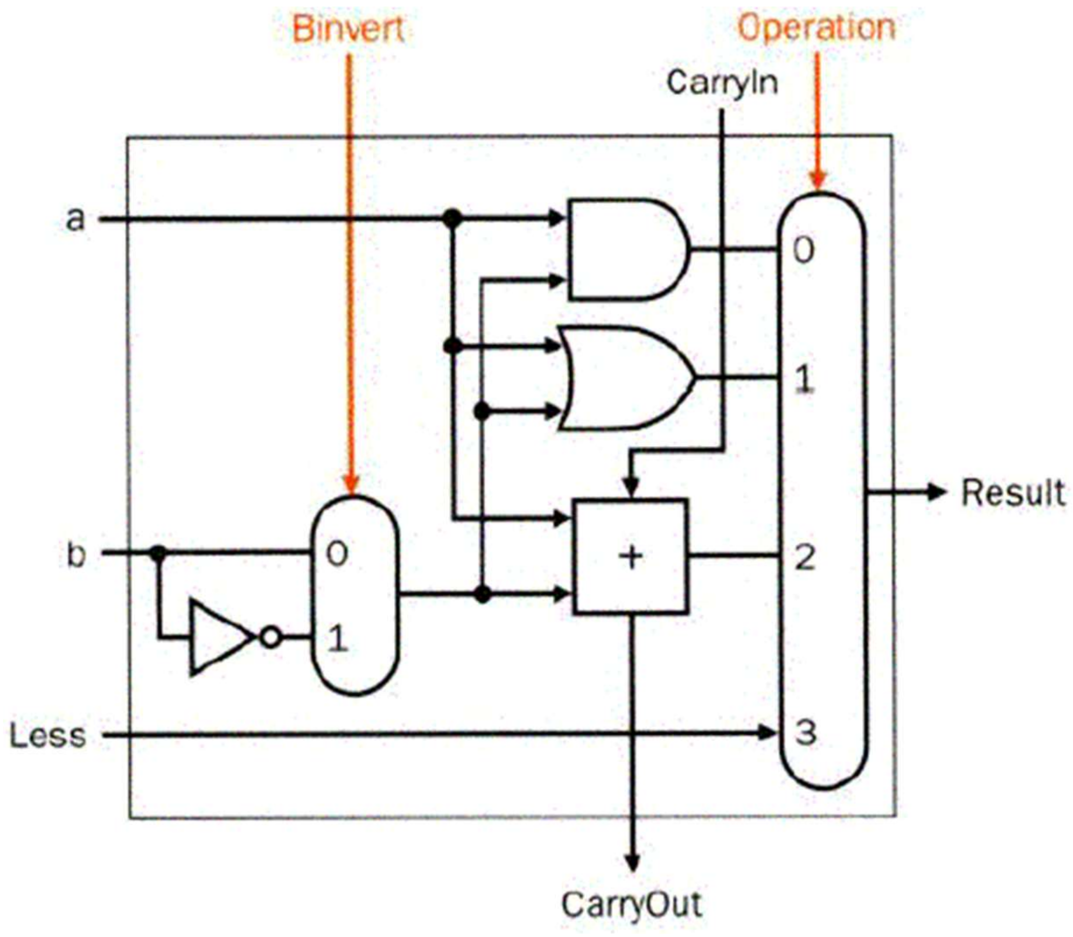


Extended 1 bit ALU

- Functions
  - AND
  - OR
  - Add
  - Subtract
- Missing: comparison
  - Slt rd,rs,rt
  - If rs < rt, rd=1, else rd=0
  - All bits = 0 except the least significant
  - Subtraction (rs - rt), if the result is negative -> rs < rt
  - Use of sign bit as indicator

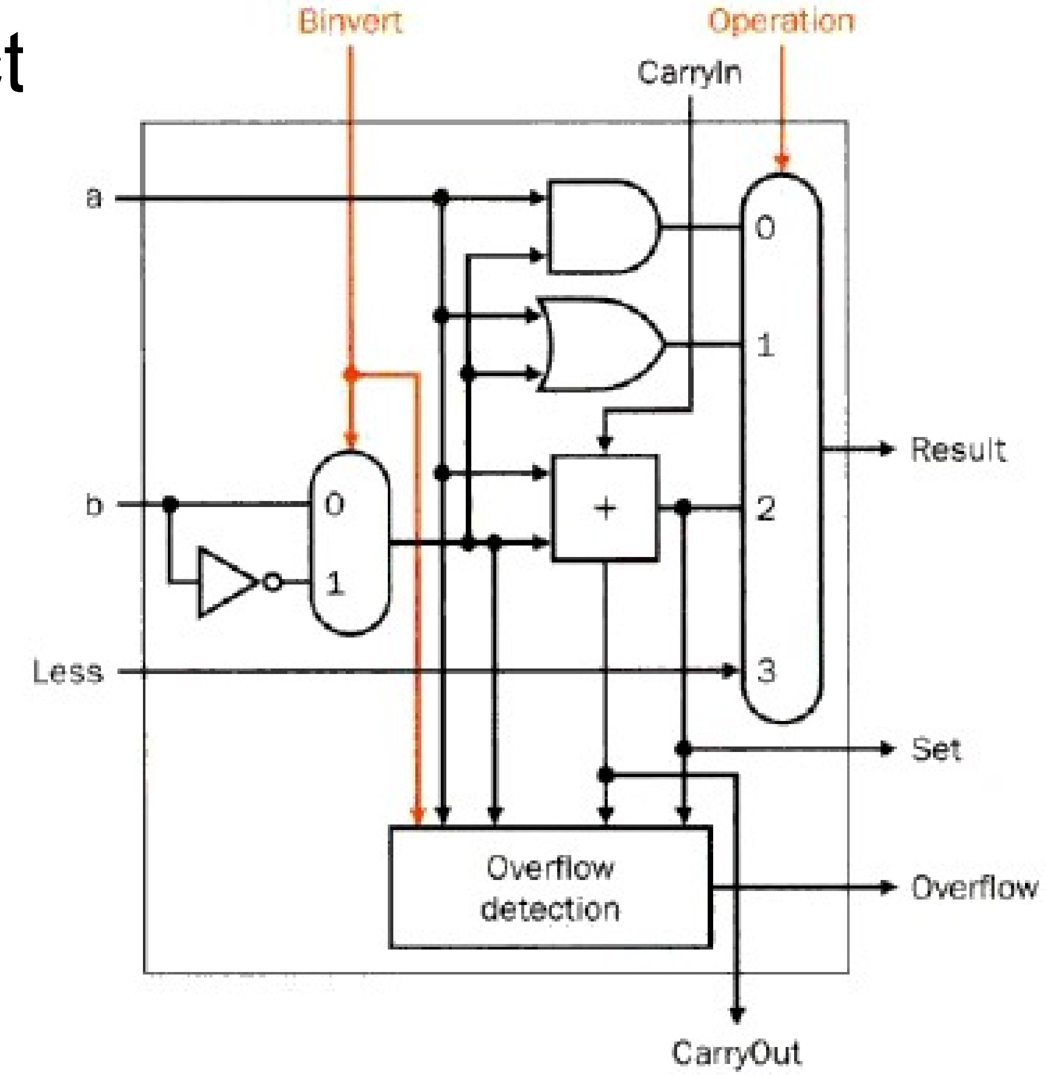
Extended 1 bit ALU

- ALU bit with input for Less data



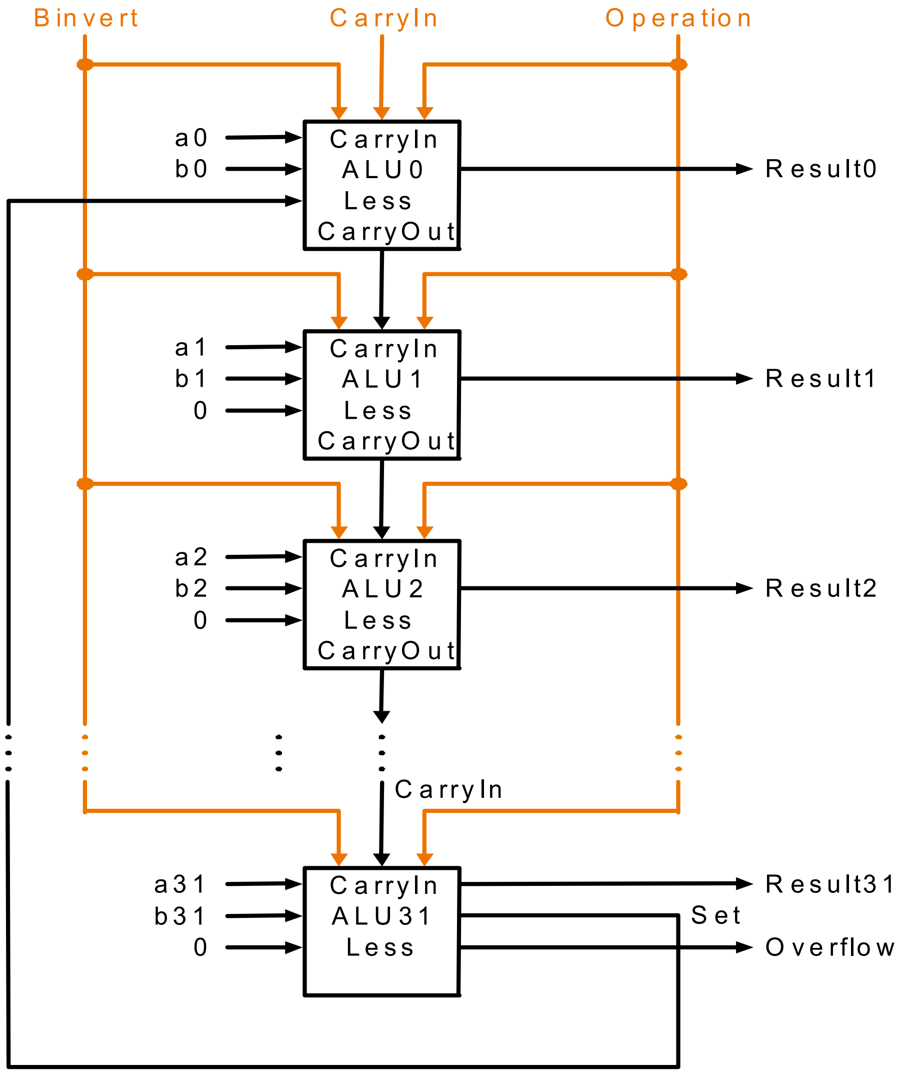
Most significant bit

- Set for comparison
- Overflow detect



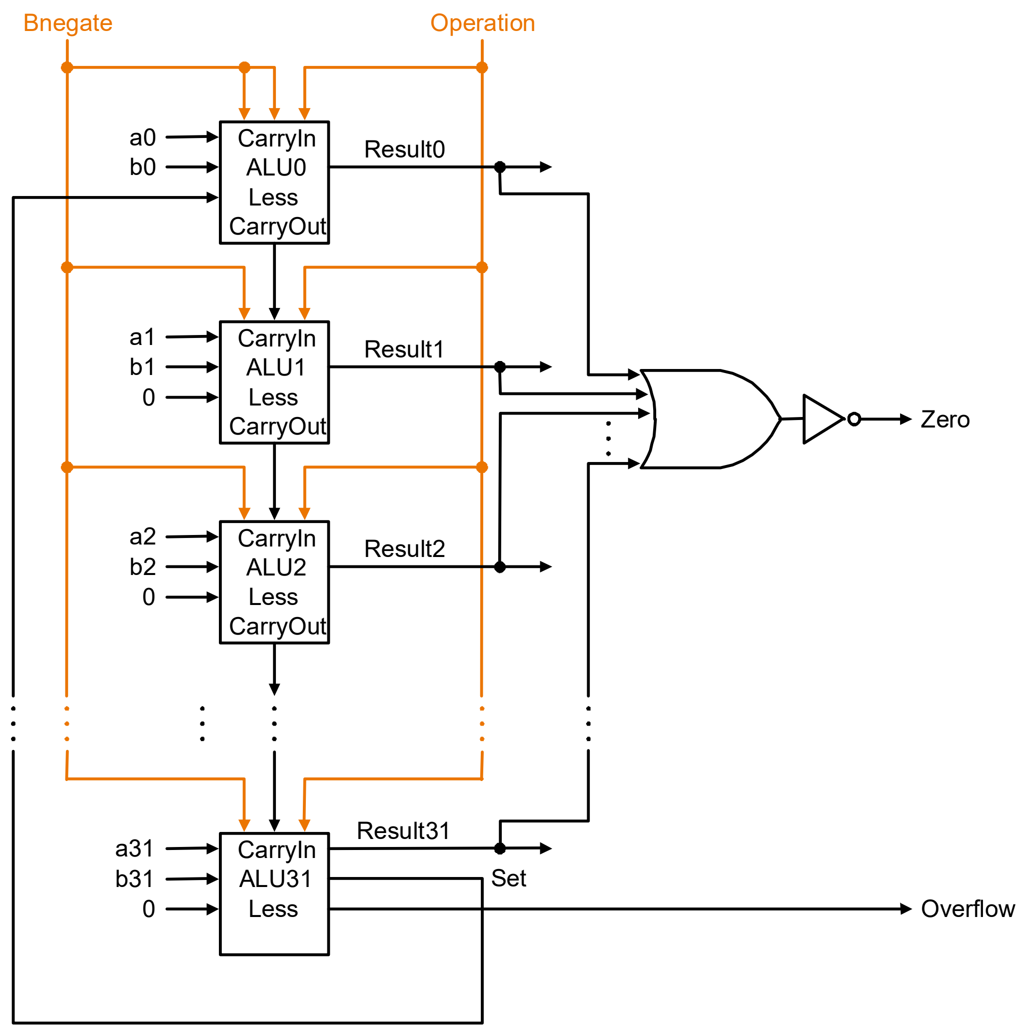
Complete ALU

- Input
  - A
  - B
- Control lines
  - Binvert
  - Operation
  - Carryin
- Output
  - Result
  - Overflow



Complete ALU

- Add a Zero detector

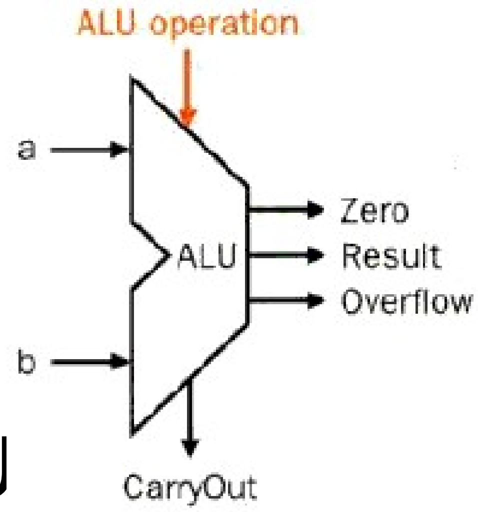


ALU symbol & control

- Function table

ALU Control Lines	Function
000	And
001	Or
010	Add
110	Sub
111	Set on less than

- Symbol of the ALU

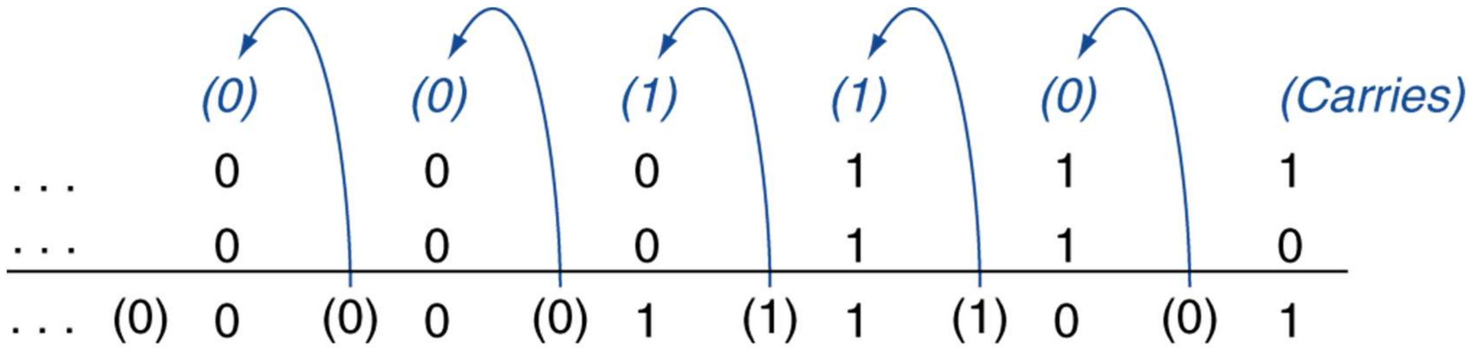


Arithmetic for Computers

- Operations on integers
  - Addition and subtraction
  - Multiplication and division
- What about fractions and real numbers?
  - Representation and operations
- How are overflow scenarios handled?
  - e.g. An operation creates a number bigger than can be represented
- How does hardware really multiply and divide numbers?

Integer Addition

- Example: 7 + 6



- Overflow if result out of range
  - Adding +ve and –ve operands, no overflow
  - Adding two +ve operands
    - Overflow if result sign bit is 1
  - Adding two –ve operands
    - Overflow if result sign bit is 0

Integer Subtraction

- Add negation of second operand
- Example: 7 – 6 = 7 + (–6)

+7:	0000 0000 … 0000 0111
–6:	1111 1111 … 1111 1010
+1:	0000 0000 … 0000 0001

- Overflow if result out of range
  - Subtracting two +ve or two –ve operands, no overflow
  - Subtracting +ve from –ve operand
    - Overflow if result sign bit is 0
  - Subtracting –ve from +ve operand
    - Overflow if result sign bit is 1

主观题 10分

设置

请在二进制下，计算-10+4

正常使用主观题需2.0以上版本雨课堂

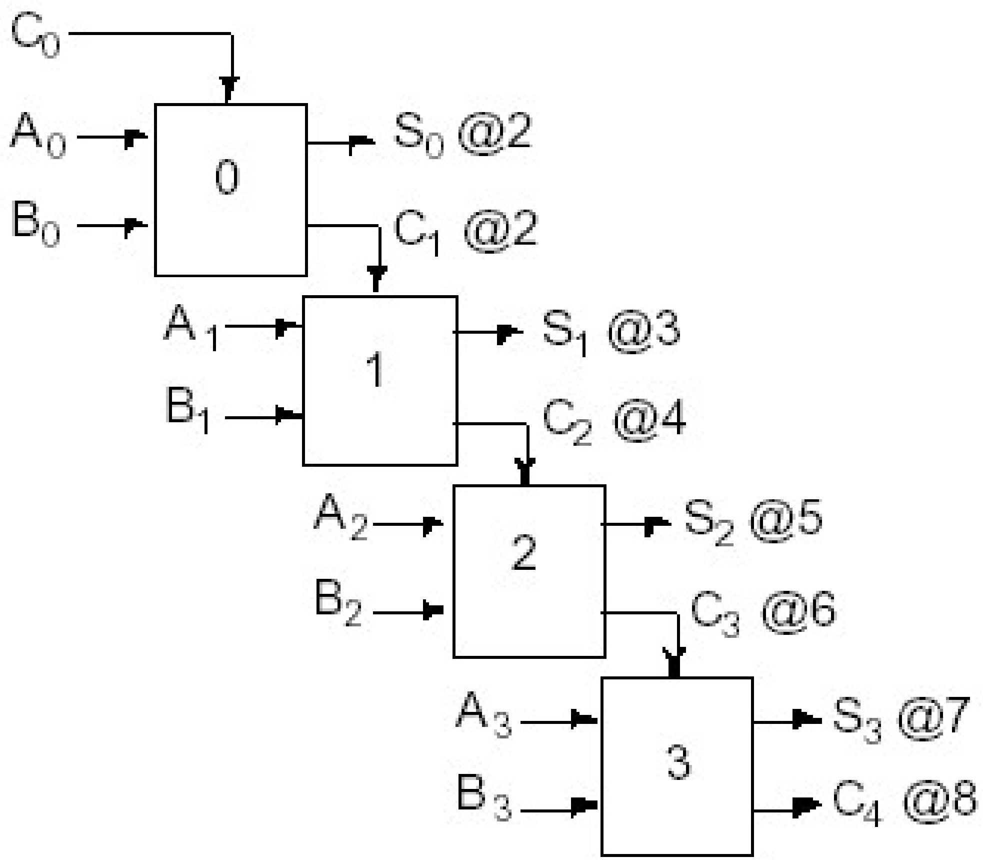
作答

Dealing with Overflow

- Some languages (e.g., C) ignore overflow
  - C compilers 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

Speed considerations

- Delay of one adder
  - 2 time units
- Total delay for stages:
  2n unit delays
- Not appropriate for high speed application



Fast adders

- All functions can be represented in 2-level logic.
- But:
  - The number of inputs of the gates would drastically rise
- Target:

  Optimum between speed and size

Fast adders

- **Carry look-ahead adder**
  - Calculating the carries before the sum is ready
- Carry skip adder
  - Accelerating the carry calculation by skipping some blocks
- Carry select adder
  - Calculate two results and use the correct one
- ...

Carry look ahead adder (CLA)

- Separation of
  - add operation
  - carry calculation
- Factorisation
  - $C_{i+1} = (b_i * c_i) + (a_i * c_i) + (a_i * b_i)$

    $= (a_i * b_i) + (a_i + b_i) * c_i$
  - Generate $g_i = a_i * b_i$
  - Propagate $p_i = a_i + b_i$

Carry look ahead adder

- Ci+1 = gi + pi * ci
- Carry generate: gi = ai * bi
  - If a and b are '1' ->
    we always have a carryout independent of ci
- Carry propagate: pi = ai+ bi
  - If only one of a and b is '1' ->
    the carry out depends on the carry in
  - pi propagates the carry

Four bit carry look ahead adder

- c1 = g0 + (p0 * c0)
- c2 = g1 + p1*c1 = g1 + (p1 * g0) + (p1 * p0 * c0)
- c3 = g2 + p2*c2 = g2 + (p2 * g1) + (p2 * p1 * g0) + (p2 * p1 * p0 * c0)
- c4 = g3 + p3*c3 = g3 + (p3 * g2) + (p3 * p2 * g1)
  +(p3 * p2 * p1 * g0) + (p3 * p2 * p1 * p0 * c0)

COMMENT:

This kind of adder will be faster than the ripple carry adder, and smaller than the adder with the tow-level logic.

PROBLEM:

If the number of the adder bits is very large, this kind of adder will be too large. So we must seek more efficient ways.

Four bit carry look ahead adder

Let’s consider a 16-bit adder.

Divide 16 bits into 4 groups.Each group has 4 bits.

As we know:

$$c4 = g3 + p3*g2 + p3*p2*g1 + p3*p2*p1*g0 + p3*p2*p1*p0*c0$$

So,we can get the following:

$$c8 = g7 + p7*g6 + p7*p6*g5 + p7*p6*p5*g4 + p7*p6*5*p4*c4$$

$$c12 = g11+p11*g10+p11*p10*g9+p11*p10*p9*g8+p11*p10*p9*p8*c8$$

$$c16=g15+p15*g14+p15*p14*g13+p15*p14*p13*g12+p15*p14*p13*p12*c12$$

Assume:

$$G0= g3 + p3*g2 + p3*p2 *g1 +p3*p2*p1*g0$$

$$G1= g7 + p7*g6 + p7*p6*g5 + p7*p6*p5*g4$$

$$G2= g11+p11*g10+p11*p10*g9+p11*p10*p9*g8$$

$$G3= g15+p15*g14+p15*p14*g13+p15*p14*p13*g12$$

$$P0= p3 * p2 * p1 * p0$$

$$P1= p7 * p6 * p5 * p4$$

$$P2= p11 * p10 * p9 * p8$$

$$P3= p15 * p14 * p13 * p12$$

Four bit carry look ahead adder

Then we get:

$c4=G0+P0*c0$; $c8=G1+P1*c4$

$c12=G2+P2*c8$; $c16=G3+P3*c12$

Assume: $C1=c4, C2=c8, C3=c12, C4=c16$

Then:

$C1=G0+P0*c0$; $C2=G1+P1*C1$

$C3=G2+P2*C2$; $C4=G3+P3*C3$

And, we can further get:

$C1=G0+P0*c0$;

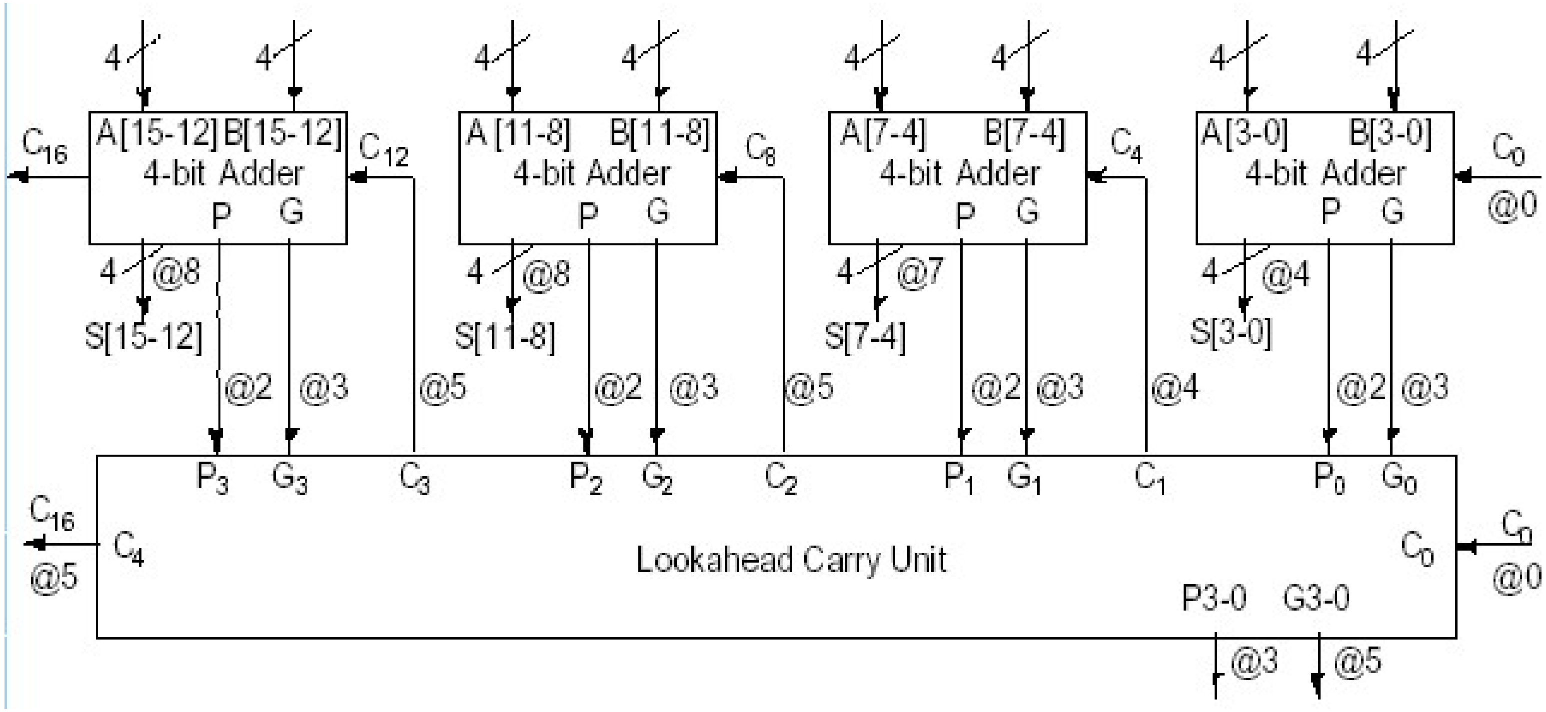
$C2= G1+P1*C1 = G1+P1*G0+ P1* P0*c0$

$C3=G2+P2*C2 = G2+P2*G1 + P2* P1*G0+ P2* P1* P0*c0$

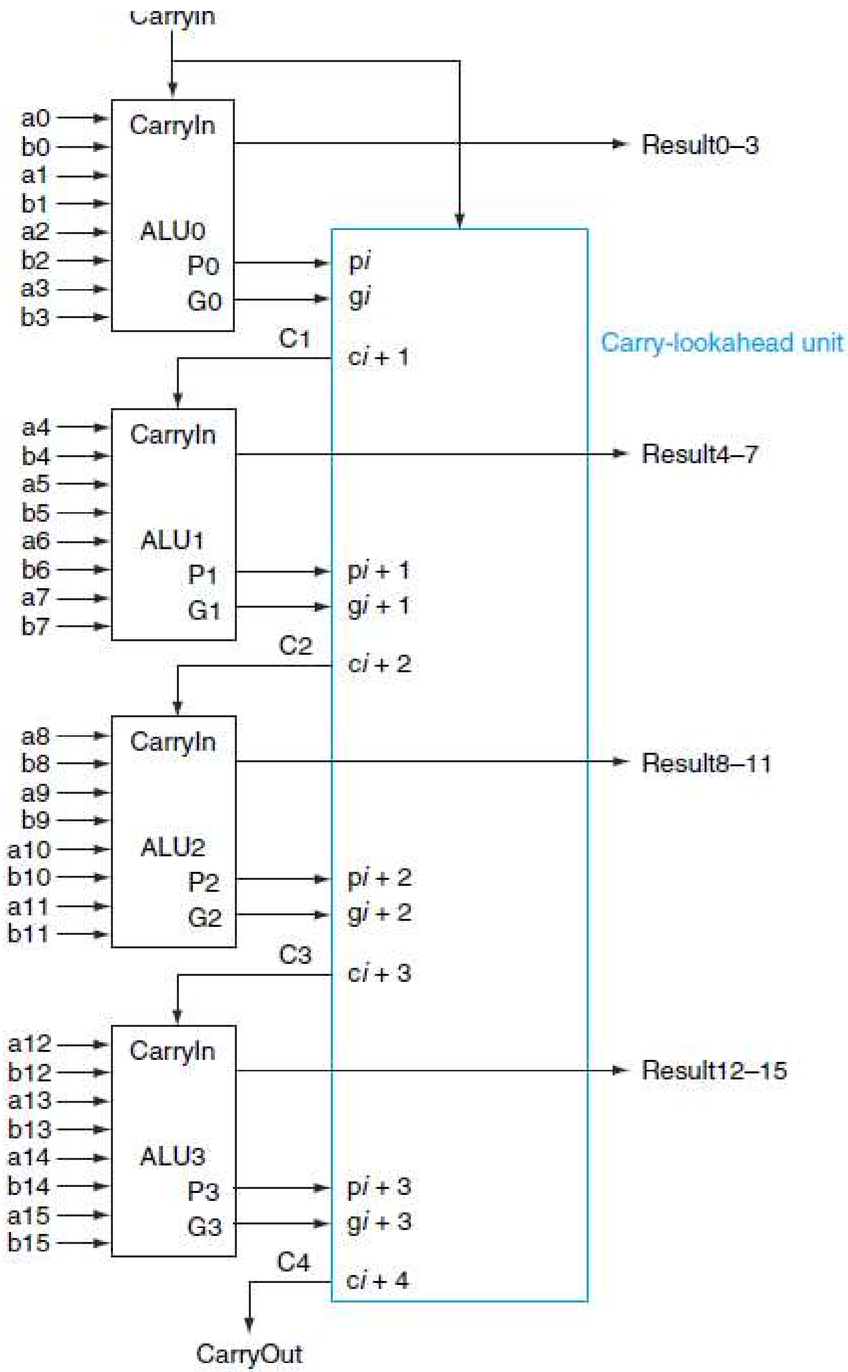
$C4=G3+P3*C3= G3+P3*G2+P3*P2*G1+P3*P2*P1*G0+P3*P2*P1*P0*c0$

Hybrid CLA + Ripple carry

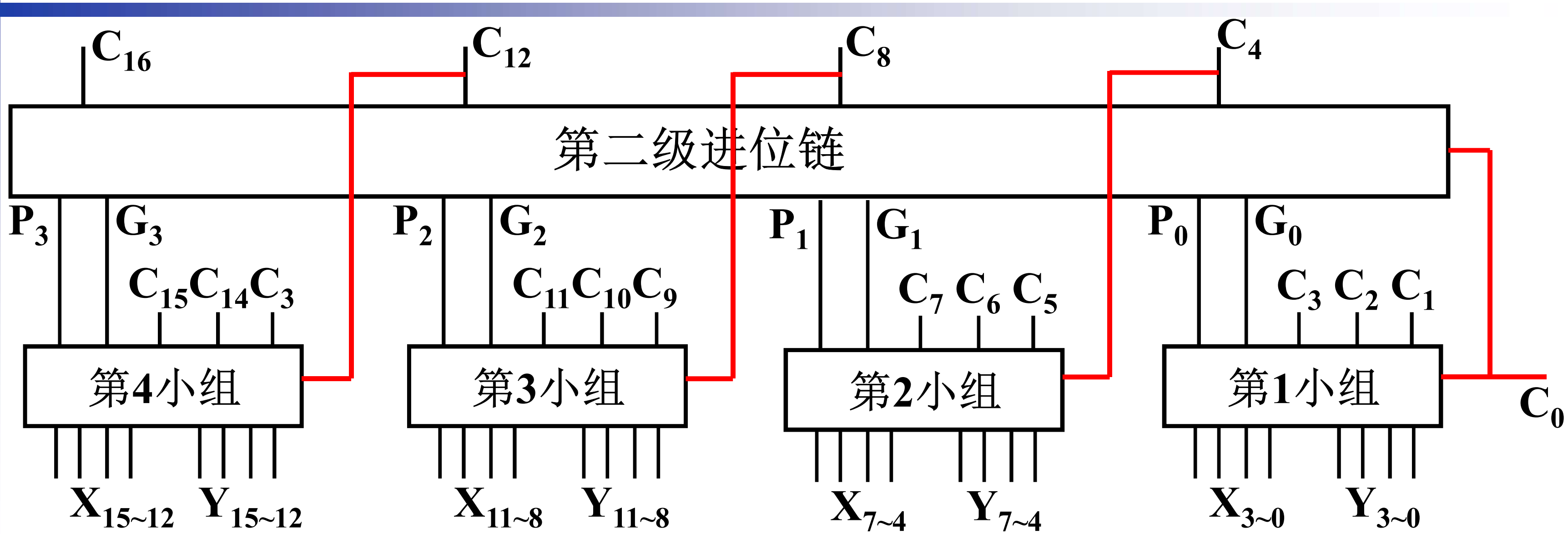
- Realisation:
  - Ripple carry adders and
  - Carry look ahead logic

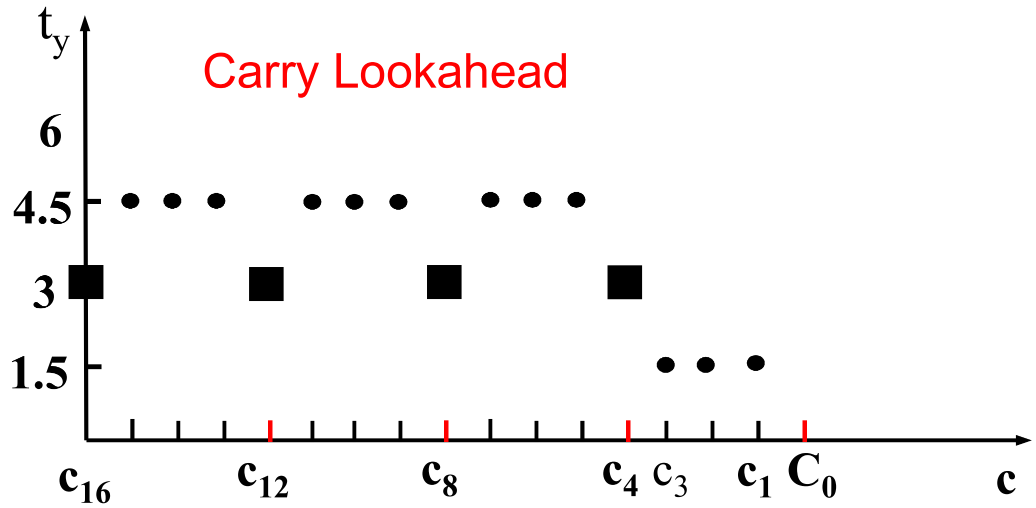


Four 4-bit ALUs using carry lookahead to form a 16-bit adder.



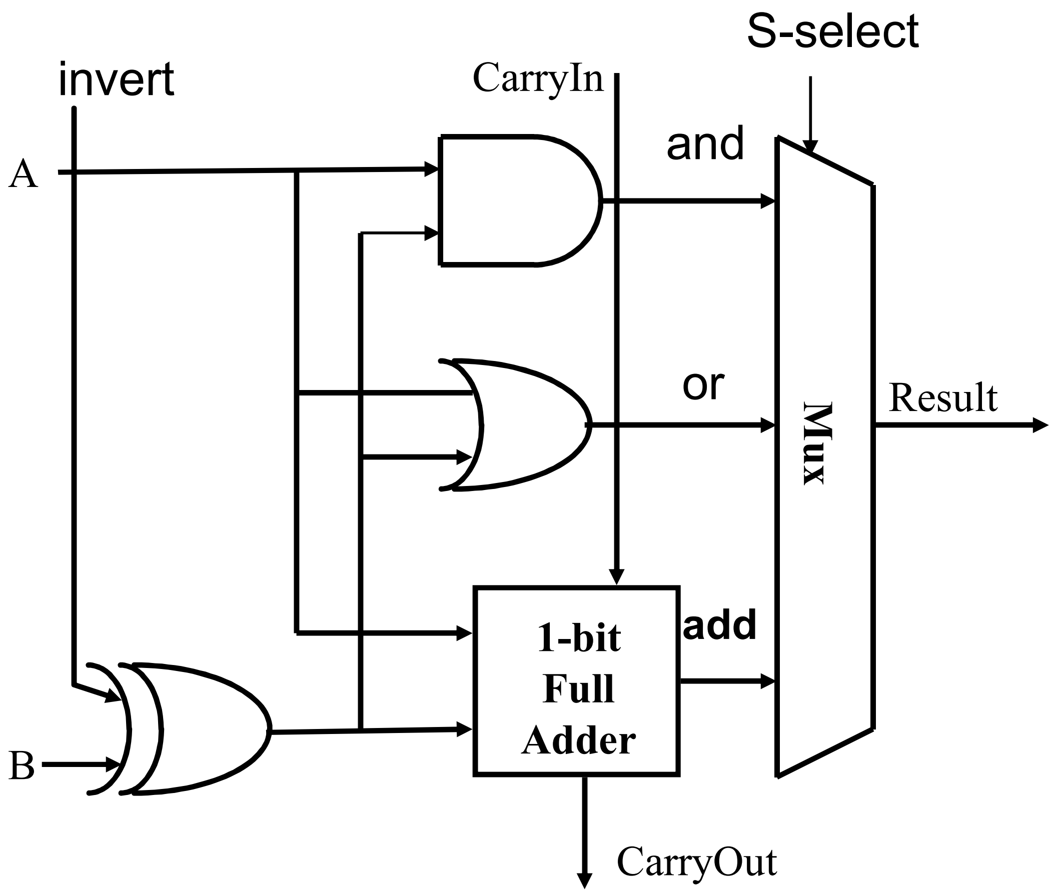
Suppose Time (AND) =0.5 T ,Time(Or)=1.0 T





Complete ALU

- A - B = A + (– B)
  - form two complement by invert and add one



Set-less-than? – left as an exercise