Lecture 4, ALU Arithmetic and Logic Unit

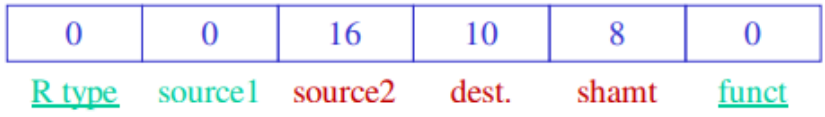
Computer Arithmetic

- two's complement: signed integer representation
  - e.g. $1011_{2C} = (1\times -2^3) + (0\times 2^2) + (1\times 2^1)+(1\times 2^0) = -5_{ten}$
  - n-bit: range $(-2^{n-1}) .. (2^{n-1}-1)$
  - sign extension: $1011_{2C} = 1111011_{2C}$
  - overflow: A, B > 0, A+B ≤ 0
    A, B < 0, A+B ≥ 0
  - in MIPS: slt, slti work with two's complement
    sltu, sltiu work with unsigned representation
    (do not cause exception when overflow)

Remember: digital electronics would just give us ones and zeros. We need to be clear what they mean – so number representation concerns how numbers are represented by a list of them and, more generally, how data are represented; while an instruction can be seen as a form of control representation. We will focus on two's complement representation, although we will also mention unsigned representation.

Logical operations

- shift left logical
  - `sll $10, $16, 8` # reg10 = reg16 « 8 bits
  - reg16 0 .. 0 0000 0000 1101
  - reg10 0 .. 0 1101 0000 0000
    (20 bits; introduce zeros)
  - format



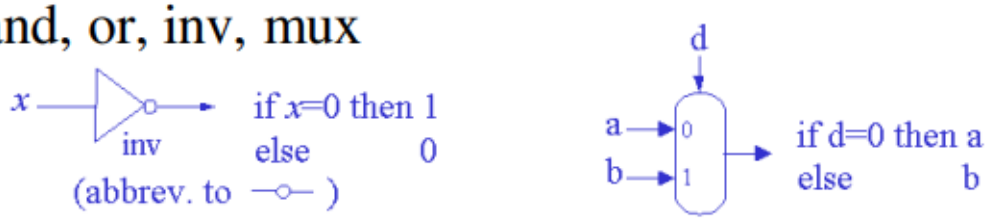
- shift left logical variable (`sllv`): `shamt` in register source1
- right shifts: `srl` , `srlv` , `sra` (sign-extend high order bits)

- bitwise: or, and - R-type
  ori, andi - I-type

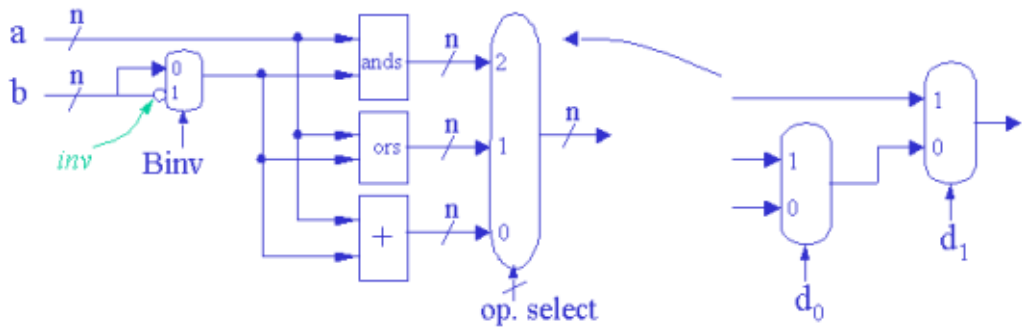
The logical operations are mainly shifting and bitwise Boolean operations.

ALU building blocks

- and, or, inv, mux



- ALU: Arithmetic Logic Unit n=32 for MIPS



- bit-level realisation: hierarchical, regular structure

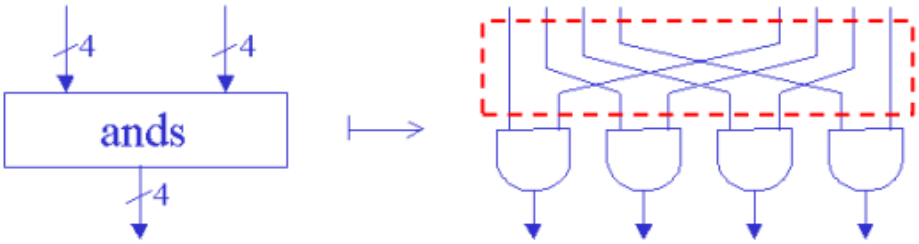
wl 2024

The figures show some basic digital logic elements, which can be used in building the ALU. This ALU can do four things: add, subtract, bitwise and, bitwise or.

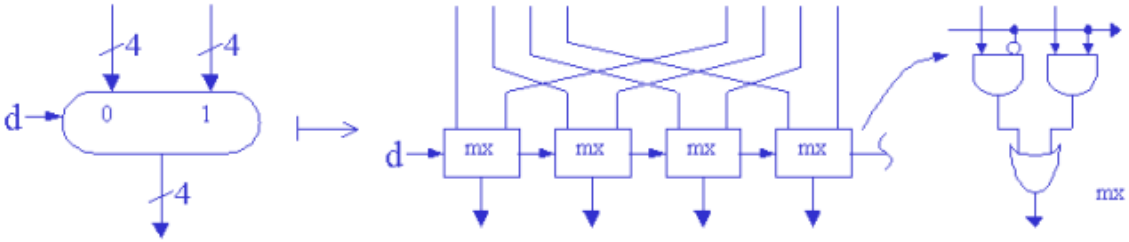
Last term you learn how an adder, a subtractor and two multiplexers can be used in building an ALU. Now we take one more step: deriving a 1-bit ALU that will, when replicated, support addition, subtraction, etc.

Bit-wise logical / selection operations

- and, or, …



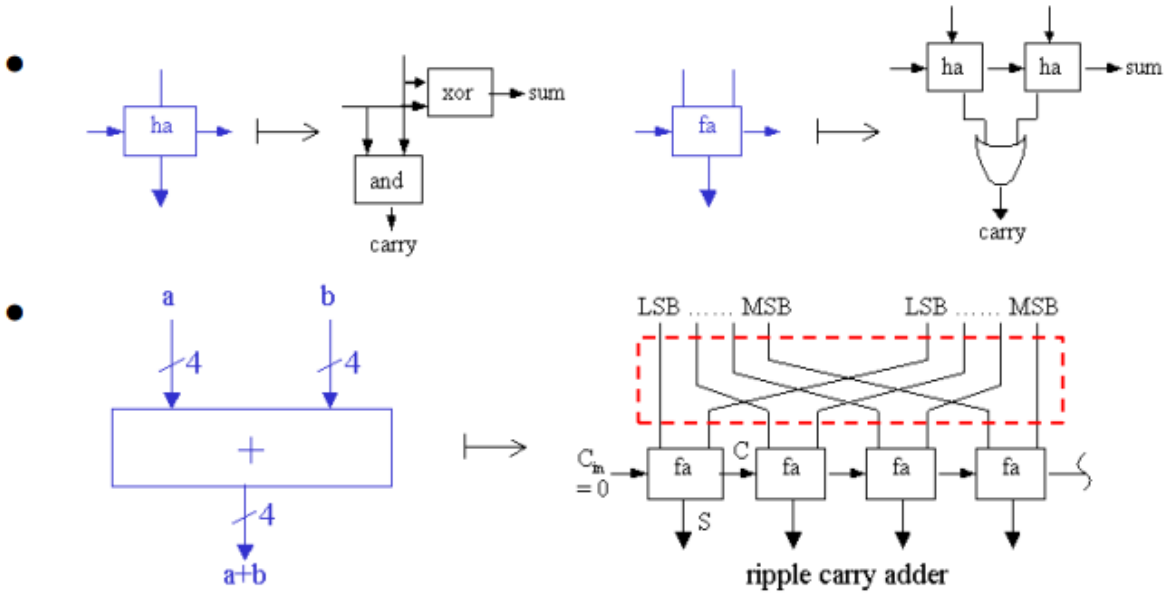
- selector / multiplexor



wl 2(

One key step is to ensure the interfaces – the ordering of the list of inputs or outputs – of a block and its bit-level realization are exactly the same. Hence we make explicit the interleave of wiring (inside the red dotted box) connecting the corresponding inputs to a particular gate.

Add / subtract



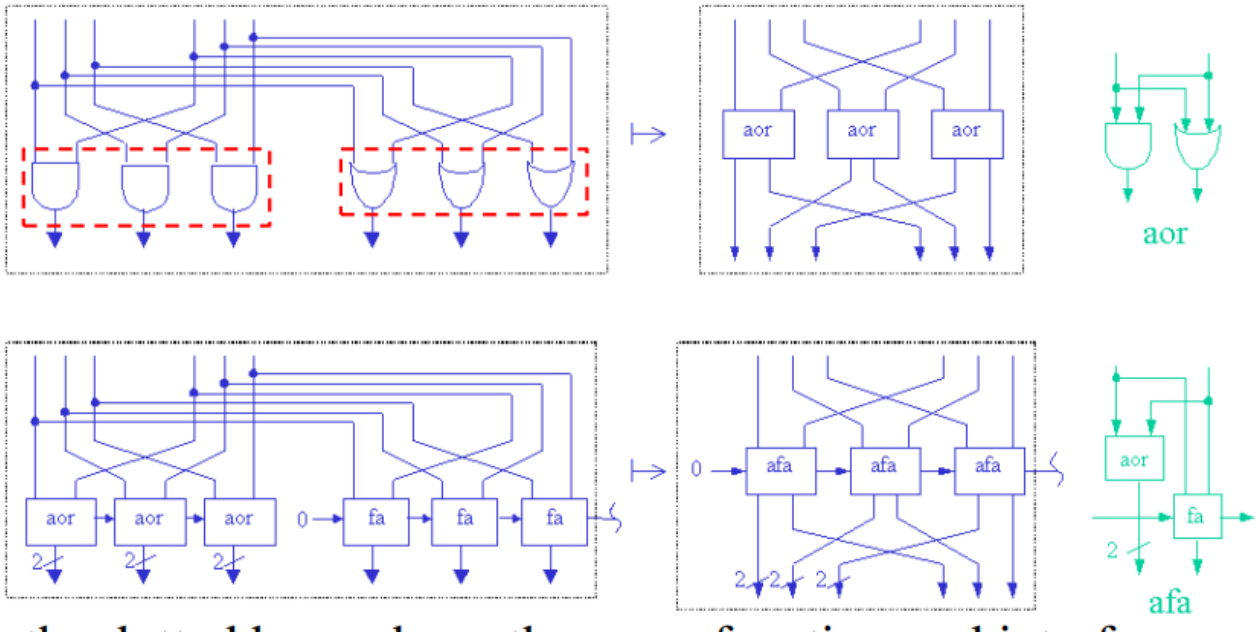
- subtractor: a-b
  - $(\sum 2^i \times b_i) + (\sum 2^i \times \overline{b_i}) = -1$ e.g. $0101_2 \oplus 1010_2 = 1111_2 = -1_{10}$
  - $(\sum 2^i \times a_i) - (\sum 2^i \times b_i) = (\sum 2^i \times a_i) + (\sum 2^i \times \overline{b_i}) + 1$
    i.e. invert b bitwise and set $C_{in} = 1$ for adder

wl 2024 4.8

Similarly, the wires connecting the two inputs to an adder block would need to be interleaved, such that the corresponding bits would be connected to the inputs of a full adder (`fa`) in a ripple carry adder. Subtraction can be computed by inverting the second input and by initialising the carry-in of the LSB full adder to a constant one.

Deriving ALU cell by interleaving components

- group components together to form larger repeated unit

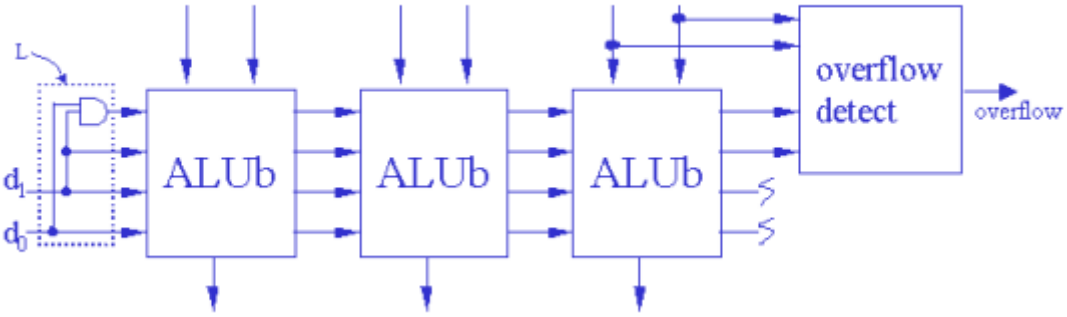


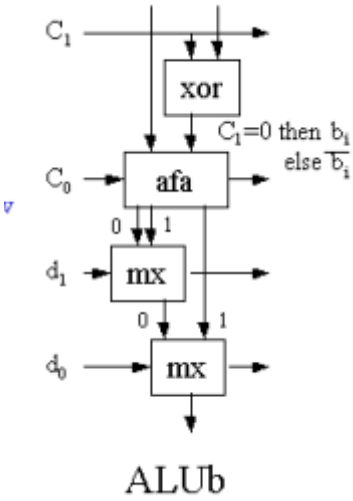
- the dotted boxes have the same function and interface

Now let us optimize it to form an efficient implementation: the corresponding elements of the two blocks in the red dotted boxes, each containing and gates and or gates (the design in blue on the left), are placed together to form an optimised version shown on the right, based on a new unit of replication `aor` containing an and gate and an or gate (shown in green on the far right at the top). This is similar to loop merging: under the right conditions, two separate loops with the same number of iterations can be merged together to form a single loop with the merged loop bodies in each iteration. It is important to ensure that the interfaces of the merged block are exactly the same as those of the original block. This process can be repeated: after merging the and gates and or gates to form a new design based on the `aor` block, this design can be further merged with an array of full adders to form a design based on the `afa` blocks, each containing an `aor` and a full adder, as shown in green on the far right at the bottom.

Selecting ALU operation

- Programmable inverter for b_i (using xor)
- Connecting `mux` in series





ALUb

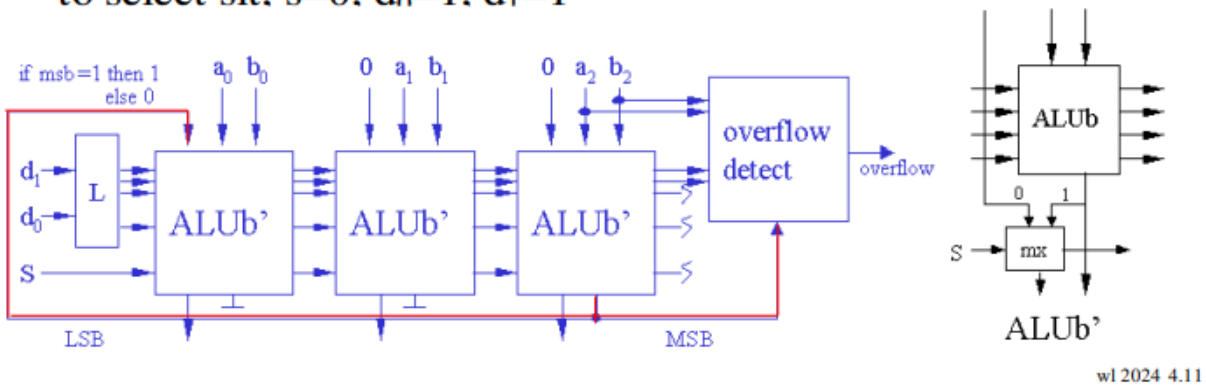
- d_0d_1: 00 and, 01 or,
  10 add, 11 subtract
- detecting overflow: exercise

After adding `xor` gates as programmable inverters and multiplexers for selecting the appropriate outputs, we can form our first ALU design based on `ALUb`, a one-bit ALU element. This design will be managed by two control signals, d0 and d1, to select one of the four functions to be performed by the ALU at run time. Each would correspond to a MIPS instruction. The wiring block L is needed to provide the appropriate initialization to the 4 left inputs of `ALUb`.

Comparison operations

This implements the `slt` function using our original circuit.

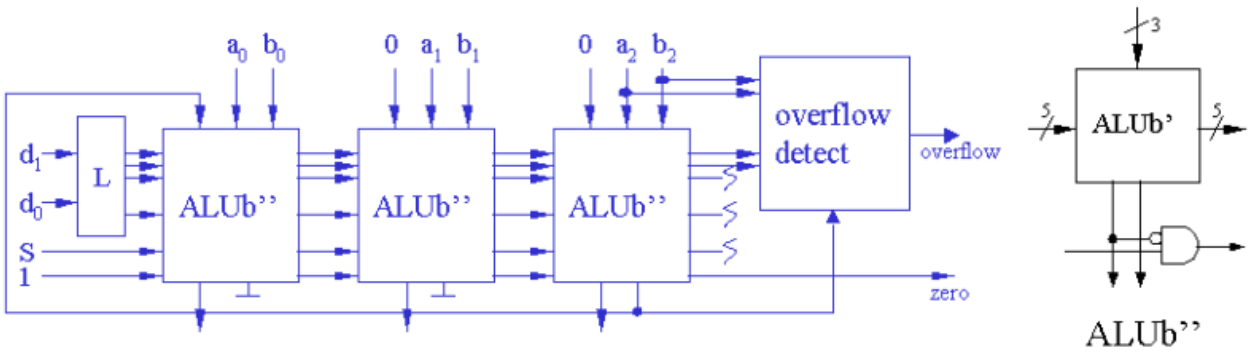
- slt: set on less than, if a < b then 1 else 0
- if a < b, a-b < 0, so MSB of (a-b) is 1 (32 bits)
- implementation
  - provide additional input to each cell, left of a_i, b_i
  - LSB input from MSB ALUb output, other inputs set to 0
  - include additional mux in cell for selection
  - to select slt. s=0. d_0=1. d_1=1



The new red wire connects the sum output of the rightmost `ALUb'` (the MSB of a-b) to a new multiplexer which either selects it or the sum output of the LSB of `ALUb`. So the LSB output will be the same as the sum output in the MSB for the SLT instruction. This is because if $a < b$, $a - b < 0$ so the MSB will be set, so the output of SLT will be a one as expected. Note that except for the LSB, the other `ALUb'` blocks are initialised to 0 by a new input at the top. This is because the value one in 32 bits includes 31 zeros, except the LSB which is one.

Zero detection

- beq, bne: test a=b or a-b=0
- include another gate to test if output zero



- summary:

s d_0 d_1	011	100	101	110	111
function:	set on less than	and	or	add	subtract

Finally, several instructions such as `bne`, `beq` require testing whether a given value from the ALU is zero or one. We use an and-gate with an inverted input to form a new 1-bit ALU (in black on the right), This completes the ALU development. The summary above shows the code that would need to provide to perform a given instruction.

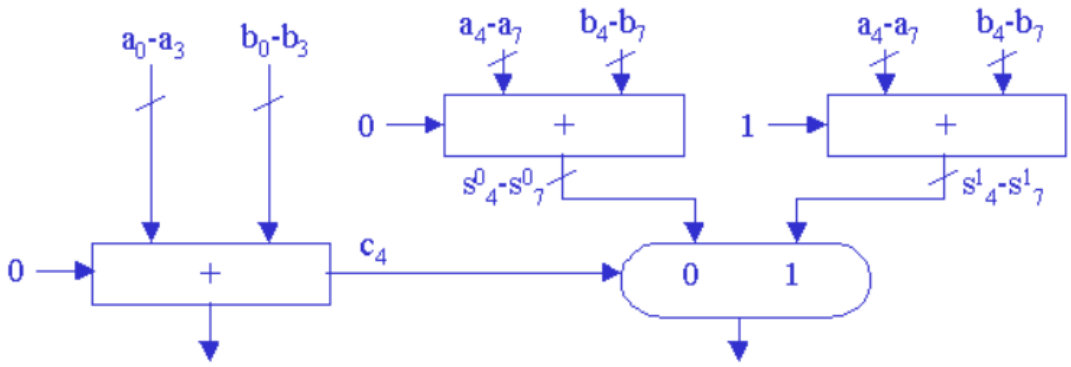
Performance estimation

- clocked circuit: no combinational loops
- speed limited by propagation delay through the slowest combinational path
- slowest path: usually carry path
- clock rate: approx. 1/(delay of slowest path) assuming
  - edge-triggered design
  - flip-flop propagation delay, set-up time, clock skew etc. negligible

To estimate performance, we find the longest combinational path with the largest delay. This delay is usually due to the long carry chain in a ripple carry adder. To reduce this delay, one would need a way to reduce the combinational path of the ripple carry adder.

Fast addition

- carry select
  - compute both zero-carry-in and one-carry-in after n stages
  - e.g. 8 bits: use three 4-bit ripple carry adders



- other possibilities
  - carry-lookahead adder
  - conditional-sum adder

Trade off between performance and resource usage. Here, we use 50% more hardware, but gain almost 50% in speed.

One way to improve performance is to adopt a carry select adder, which runs faster than a ripple carry adder: the top half inputs (a4 -a7, b4 -b7) are added by two ripple carry adders, one initialized to zero and the other initialized to one. So they can run in parallel with the ripple carry adder for the bottom half inputs (a0 -a3, b0 -b3). So a carry select adder can run almost twice as fast as a ripple carry adder, at the expense of 50% more full adders. There are other fast adder designs such as the carry-lookahead adder – more details from Appendix B.6 available from the course homepage.