Computer Architecture

Lecturer: Mihai Negru

2nd Year, Computer Science

Lecture 5: ALU Design

http://users.utcluj.ro/~negrum/

Binary Number Representation

Sign Magnitude	One's Complement	Two's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

- 2's complement – advantages
  - Subtract can share the same logic as add
  - Sign bit can be treated as a normal number bit in addition
- 1's complement disadvantage
  - Two zero representations

Binary Number Representation – MIPS

- Unsigned binary integers
  - Typically represent addresses or other values that are guaranteed not to be negative
  - Unsigned value

$$value = \sum_{i=0}^{n-1} b_i \cdot 2^i$$

  - An n-bit unsigned binary integer has a range from 0 to $2^n - 1$
- Signed binary integers
  - Typically used to represent data that is either positive or negative
  - The most common representation → the 2's complement format
  - Signed value (2’s complement)

$$value = -b_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i \cdot 2^i$$

  - An n-bit 2’s complement binary integer has a range from -2^{n-1} to $2^{n-1} - 1$

Binary Number Representation – MIPS

- 32-bit signed numbers

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +1_{ten}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = +2_{ten}$

...

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +2{,}147{,}483{,}646_{ten}$

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +2{,}147{,}483{,}647_{ten}$ → max_int

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2{,}147{,}483{,}648_{ten}$ → min_int

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -2{,}147{,}483{,}647_{ten}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -2{,}147{,}483{,}646_{ten}$

...

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

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

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1_{ten}$

Binary Number Operations – MIPS

- 2’s complement negation
  - Invert all bits and add one to the least significant bit
  - 2’s complement representation: 6 = 0110 -4 = (not 0100 + 0001) = 1100
- 2’s complement addition
  - Add the corresponding bits of both numbers with carry between bits

3 = 0011	-3 = 1101	-3 = 1101	3 = 0011
+ 2 = 0010	+-2 = 1110	+ 2 = 0010	+ -2 = 1110
- - - -	- - - -	- - - -	- - - -

  - Unsigned and 2’s complement addition are performed in exactly the same way, only the overflow detection differs
- 2’s complement subtraction
  - Negate the second number and then perform addition

3 = 0011	-3 = 1101	-3 = 1101	3 = 0011
- 2 = 0010	- -2 = 1110	-2 = 0010	- -2 = 1110
- - - -	- - - -	- - - -	- - - -

Binary Number Operations – MIPS

- Overflow
  - The sum or difference can go beyond the range of representable numbers
  - Overflow: the result is too large or too small for proper representation

5 = 0101	-5 = 1011	+5 = 0101	-5 = 1011
+ 6 = 0110	+-6 = 1010	-- 6 = 1010	- +6 = 0110
- - - -	- - - -	- - - -	- - - -
-5=1011	5=0101	-5=1011	5=0101

  - Overflow generates an incorrect result that should be detected
  - Overflow occurs when the resulting value affects the sign
    - 2 positive numbers and the sum is negative
    - 2 negative numbers and the sum is positive

Operation	A	B	Result indicating overflow
A + B	>= 0	>= 0	< 0
A + B	< 0	< 0	>= 0
A – B	>= 0	< 0	< 0
A – B	< 0	>= 0	>= 0

No overflow when

- signs are different for addition
- signs are the same for subtraction

Binary Number Operations – MIPS

- Overflow detection – 2's complement numbers
  - When adding 2's complement numbers, overflow will occur only if
    - the numbers being added have the same sign
    - the sign of the result is different

$$
\begin{array}{r}
a_{n-1}\ a_{n-2} \ldots a_1\ a_0 \\
+\ b_{n-1}\ b_{n-2} \ldots b_1\ b_0 \\
\text{----------------------} \\
=\ s_{n-1}\ s_{n-2} \ldots s_1\ s_0
\end{array}
$$

$$overflow = a_{n-1} \cdot b_{n-1} \cdot \overline{s_{n-1}} + \overline{a_{n-1}} \cdot \overline{b_{n-1}} \cdot s_{n-1}$$

  - If c_{n-1} and c_n represent the input and output carry signals for the MSB

Operands	Result	c_n	s_{n-1}	a_{n-1}	b_{n-1}	c_{n-1}	event ?
Positive	Positive	0	0	0	0	0	$c_n=c_{n-1}$ ⇔ no overflow
	Negative	0	1	0	0	1	$c_n!=c_{n-1}$ ⇔ overflow
Negative	Positive	1	0	1	1	0	$c_n!=c_{n-1}$ ⇔ overflow
	Negative	1	1	1	1	1	$c_n=c_{n-1}$ ⇔ no overflow

Overflow means → $c_n != c_{n-1}$

Overflow detection → $overflow = CarryOut\ MSB \text{ xor } CarryInMSB$

$$overflow = c_n \otimes c_{n-1}$$

Binary Number Operations – MIPS

- Overflow detection – unsigned numbers
  - Unsigned numbers – overflow → carry out of the most significant bit

$$overflow = c_n$$

```
   1001   = 9
 + 1000   = 8
   - - - -      -
 = 0001   = 1
```

$c_n = 1$

- MIPS architecture
  - Overflow exceptions are signaled for 2's complement arithmetic
    - add, sub, addi
  - Overflow exceptions are not signaled for unsigned arithmetic
    - addu, subu, addiu

Binary Number Operations – MIPS

- Set on Less Than – SLT
  - Signed integers less than condition → SF != OF → sign xor overflow
  - Set-on-Less Than Signed (SLT) instruction

  slt \$rd, \$rs, \$rt // R-type

  If(RF[rs] < RF[rt]) → RF[rd] ← 1 else RF[rd] ← 0

  If(RF[rs] – RF[rt]) → (SF xor OF) = 1 → RF[rd] ← 1 else RF[rd] ← 0
- Set on Less Than Unsigned – SLTU
  - Unsigned integer numbers

  sltu \$rd, \$rs, \$rt // R-type

  If(RF[rs] < RF[rt]) → RF[rd] ← 1 else RF[rd] ← 0

  If(RF[rs] – RF[rt]) → CF = 0 → RF[rd] ← 1 else RF[rd] ← 0
- SLT or SLTU can be accomplished by
  - subtracting \$rt from \$rs
  - setting the least significant bit of the result to ((SF xor OF) or ~CF) and setting all other bits to zero

Arithmetic Logic Unit Design

- The Design Process
  - "To Design Is To Represent"
  - Design Begins With Requirements
    - Functional capabilities
    - Performance characteristics
  - Design Finishes as Assembly
  - Design understood in terms of components and how they have been assembled
    - Top-Down *decomposition* of complex functions (behaviors) into more primitive ones
    - Bottom-up *composition* of primitive building blocks into more complex assemblies

Arithmetic Logic Unit Design

- The ALU should support a subset of arithmetic-logic instructions of MIPS.

Type	opcode	function
addi	001000	xxxxxx
addiu	001001	xxxxxx
slti	001010	xxxxxx
sltiu	001011	xxxxxx
andi	001100	xxxxxx
ori	001101	xxxxxx
xori	001110	xxxxxx
lui	001111	xxxxxx
beq	000100	xxxxxx

Type	opcode	function
add	000000	100000
addu	000000	100001
sub	000000	100010
subu	000000	100011
and	000000	100100
or	000000	100101
xor	000000	100110
nor	000000	100111
slt	000000	101010
sltu	000000	101011

beq – specific for ALU operation

Designing an ALU for the MIPS ISA

- MIPS ALU requirements for a limited subset of instructions
  - add, addu, sub, subu, addi, addiu → 2's complement adder / subtractor with overflow detection (signed arithmetic generates overflow → detection)
  - and, andi, or, ori, xor, xori, nor → Logical AND, OR, XOR, NOR
  - slt, sltu, slti, sltiu → 2's complement adder, check sign/overflow of result
  - beq → zero detector
- MIPS arithmetic-logical instruction formats

31 – 26	25 – 21	20 – 16	15 – 11	10 – 6	5 – 0
opcode	rs	rt	rd	sa	function
6	5	5	5	5	6

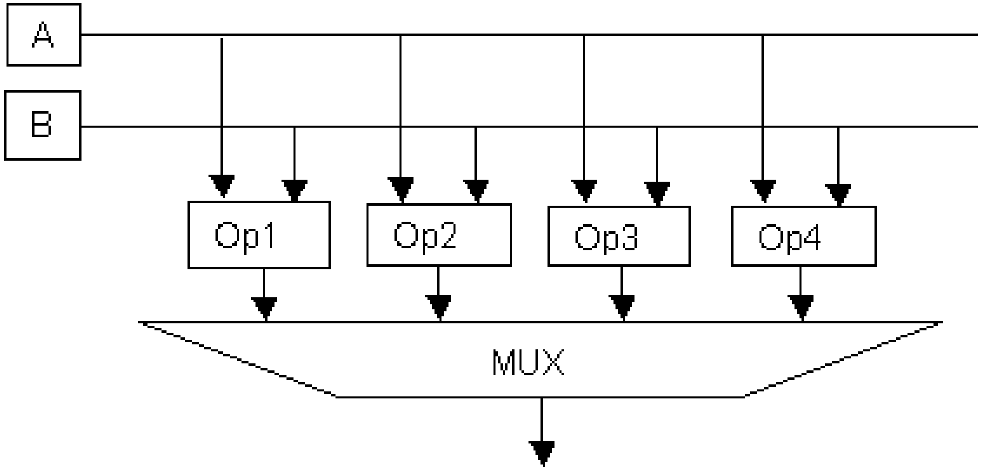
31 – 26	25 – 21	20 – 16	15 – 0
opcode	rs	rt	address / immediate
6	5	5	16

Arithmetic Logic Unit Design

- Design Trick 1: Divide et Impera
  - Break the problem into simpler ones
  - Solve them and glue together the solution
  - Example:
    - Sign / Zero Extended immediates before the ALU
    - No specific ALU ops for Immediates: addi, addiu, executed as add, addu
- Refined requirements (Functional Specification)
  - ALU inputs:
    - 2 x 32-bit operands A, B
    - 4-bit operation code
  - ALU outputs:
    - 32-bit result
    - Sign, Carry, Overflow flags
  - ALU operations:
    - add, addu, sub, subu, and, or, xor, nor, slt, sltu

Arithmetic Logic Unit Design

- Design Trick 2:
  - Take standard digital logic components (AND, OR, +, ...),
  - Connect them conform specification, and select the required operation by MUX (Laboratory version)

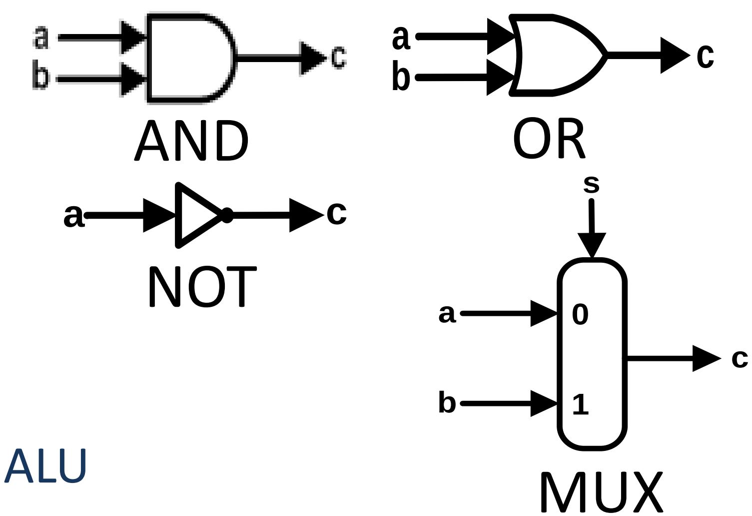


- Design trick 3:
  - Solve part of the problem and extend it
  - Start with AND, OR

Arithmetic Logic Unit Design

- Building a basic Arithmetic Logic Unit
  - Construct an ALU from:
    - logic gates: AND, OR, XOR, etc.
    - inverters
    - multiplexers
  - MIPS word is 32 bits wide
  - → we need a 32-bit-wide ALU
  - Connect 32 x 1-bit ALUs to create the MIPS ALU

Basic ALU Building Blocks

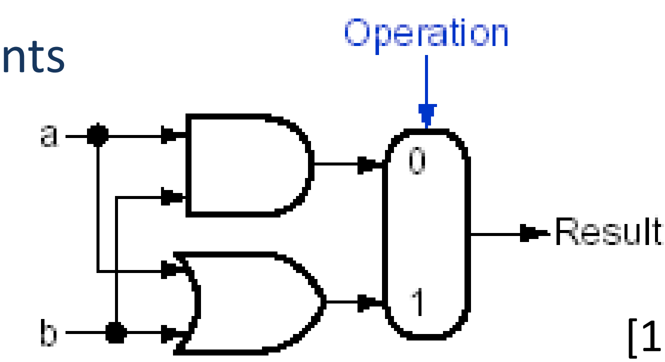


- 1-bit ALU Design
  - Start with logical operations
  - → they map directly onto the Hardware components

1-bit ALU for Logical AND and Logical OR

The multiplexer selects the Results as:

- a AND b
- a OR b

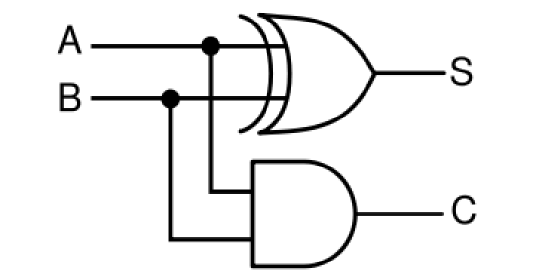


[1]

Arithmetic Logic Unit Design

- The next function to include is <u>addition</u>
  - HALF ADDER / FULL ADDER

S = A xor B
C = A and B

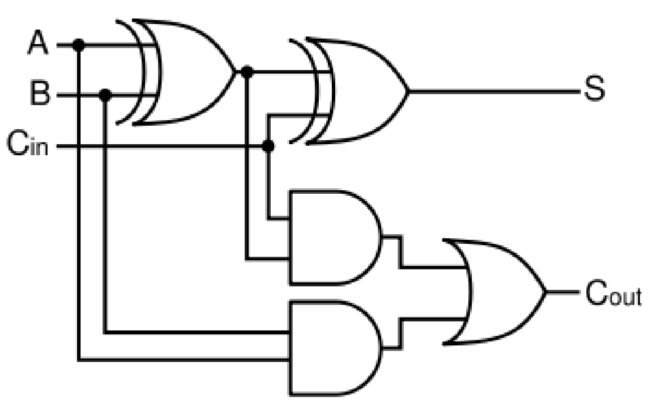


A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

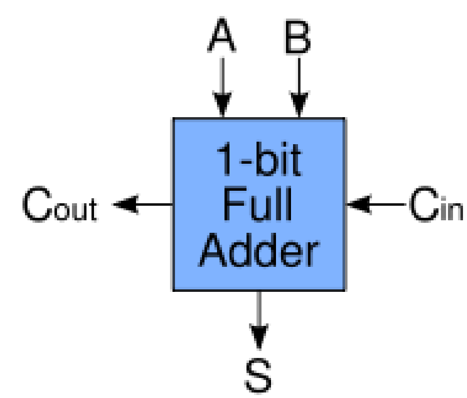
Observation for Full Adder:

C_{in} and (A or B)→C_{in} and (A xor B)

The difference between or/xor is in the A=B=1 case (covered by A and B)



A	B	C_{in}	C_{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

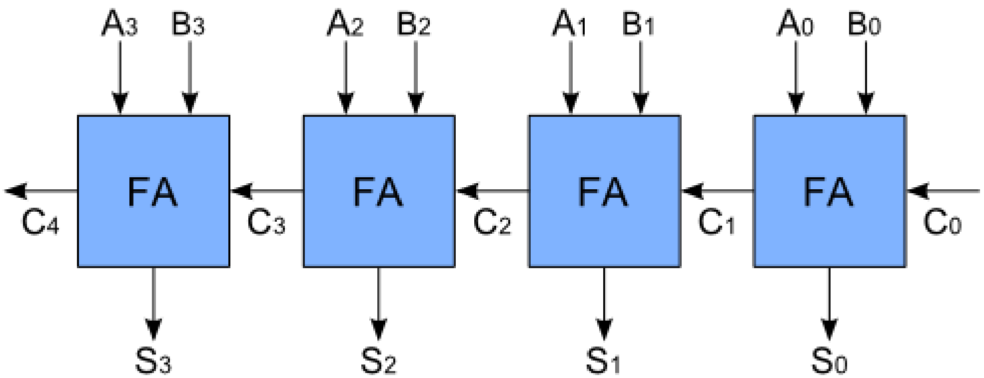


1-bit Full Adder Symbol

S = A xor B xor C_{in}

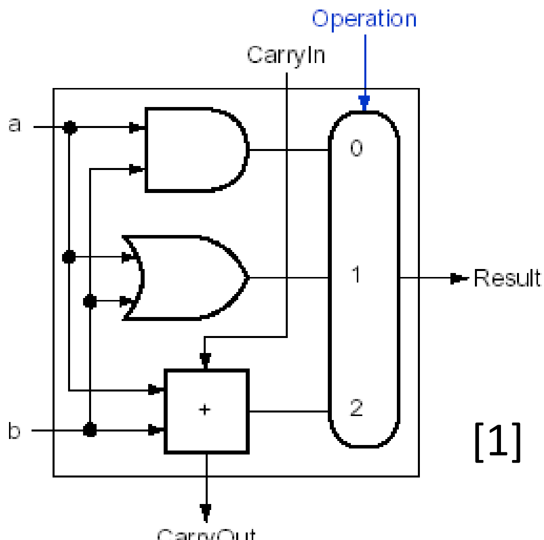
C_{out} = A and B or A and C_{in} or B and C_{in} = A and B or C_{in} and (A xor B)

Arithmetic Logic Unit Design



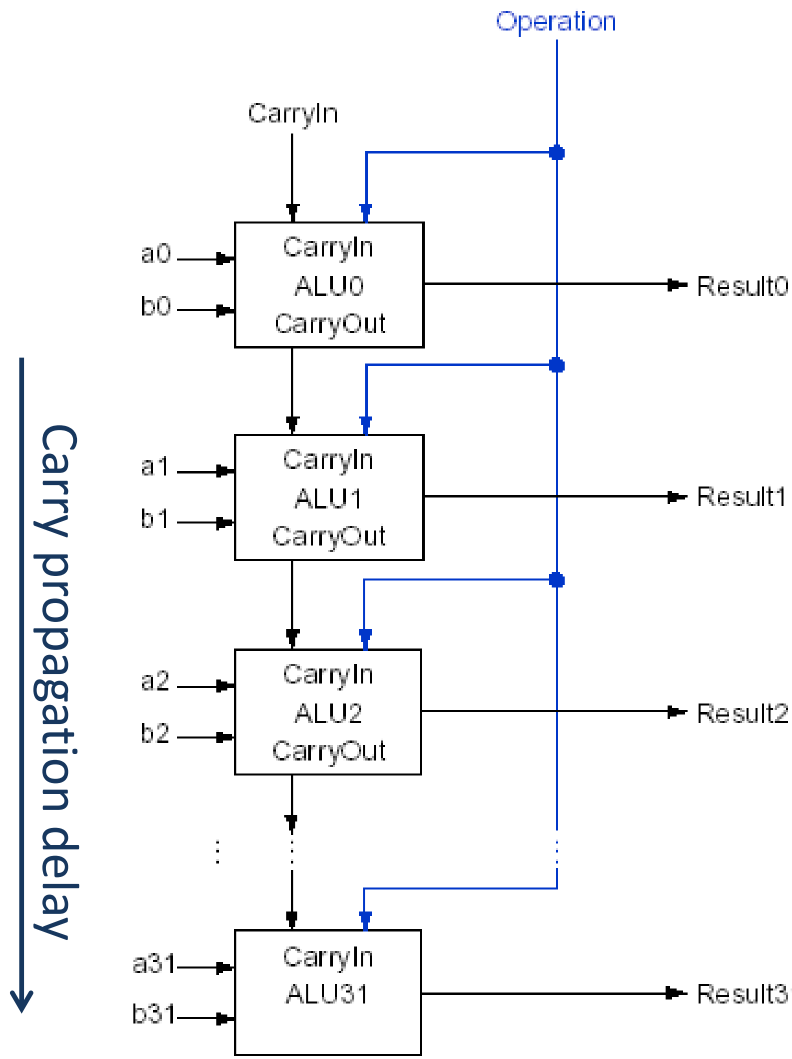
4-bit Ripple Carry Adder:

$C_{in} \leftarrow C_{out}$ between stages



[1]

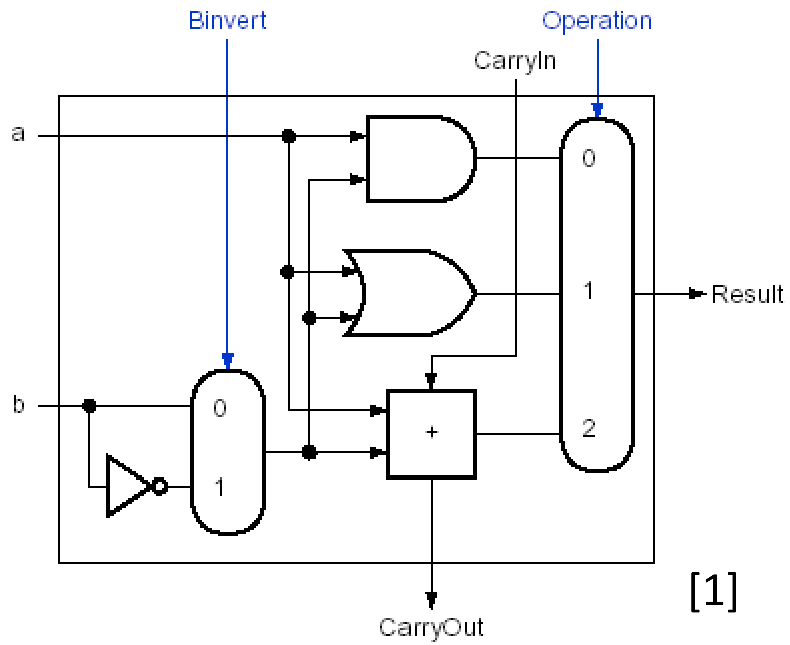
1-bit ALU for AND, OR, ADD; Operation – 2bits



[1]

32-bit ALU built with 32 x 1-bit ALUs

- Additional Operations: subtraction
  - a – b = a + (-b)
  - Subtraction is the same as adding the negative version of an operand
  - 2’s complement negate → invert each bit and then add 1
  - To invert each bit, we simply add a 2:1 mux that chooses between b and $\overline{b}$



[1]

1-bit ALU with subtraction

$$a - b = a + \overline{b} + 1$$

Operation = 2
Binvert = 1
CarryIn = 1

Subtraction ⇔ Adding 2’s complement of b to a

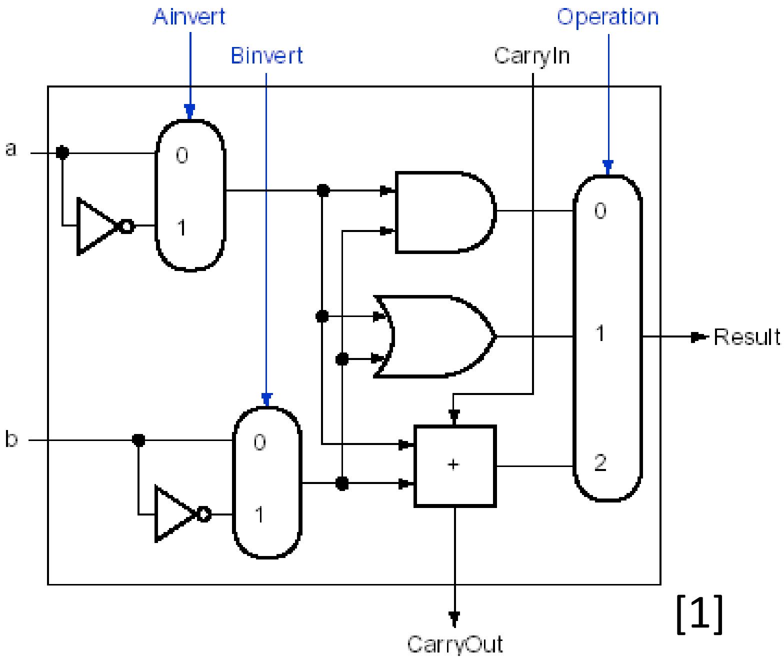
Arithmetic Logic Unit Design

- Additional operations: <u>NOR</u>
  - Instead of adding a separate NOR gate, reuse the hardware already present in the ALU

$$NOR: \overline{a \ OR \ b} = \overline{a} \ AND \ \overline{b}$$

De Morgan's Theorems

  - Because we already have AND and $\overline{b}$, we need to add $\overline{a}$ to the ALU



[1]

1-bit ALU: AND, OR, NOR, ADD, SUB

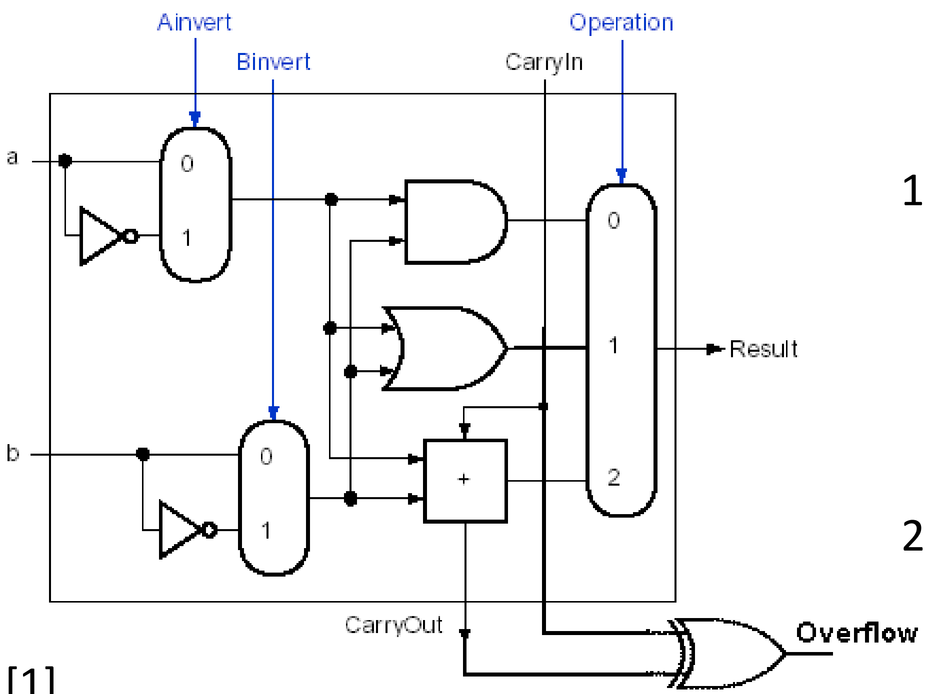
Ainvert = 1
Binvert = 1
Operation = 0

→ we get (a NOR b) instead of (a AND b)

What about NAND? → homework

Arithmetic Logic Unit Design

- Overview Detection $overflow = CarryOut\ MSB \text{ xor } CarryInMSB$



[1]

1-bit ALU for the MSB bit with overflow detection

CPU action at an overflow, two methods:

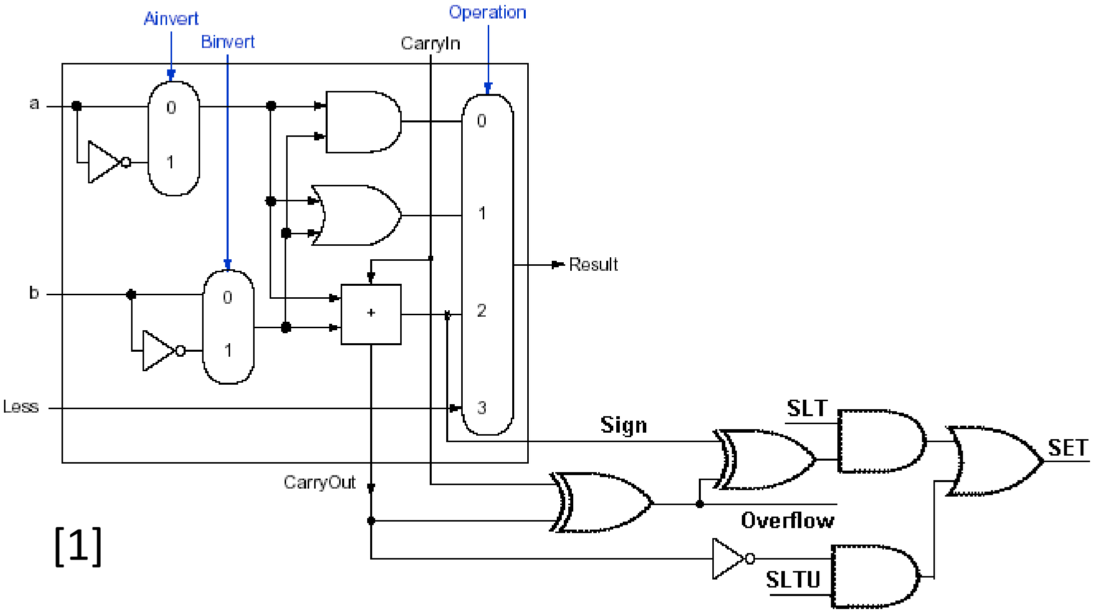
1. Ignore it → MIPS for unsigned instructions
   - Do not detect overflow for
     - addu, addiu, subu
     - addiu still sign-extends!
     - sltu, sltiu for unsigned comparisons

2. Recognize it → MIPS for signed Instructions
   - Generate a trap so that the programmer can try to deal with it
     - An exception (interrupt) occurs
     - Control jumps to predefined address for exception
     - Interrupted address is saved for possible resumption
     - MIPS instructions: add, sub

Arithmetic Logic Unit Design

- Additional operations: set on less than instruction (slt)
  - the slt operation produces 1, if RF[rs] < RF[rt], and 0 otherwise
  - slt will set all bits except the LSB to 0
  - the LSB set according to the comparison RF[rs] < RF[rt]
  - expand the 3-input MUX of the ALU to add a new input for: Less → slt result
  - Connect 0 to the Less inputs: Less[31:1] = 0
  - How to set the Less[0]?
    - Subtract b from a. If the difference is negative, then a < b
    - Analyze Bit 31 of results → Sign Bit
      - Sign = 1 → negative result (a < b) / Sign = 0 → positive result (a > b)
    - Analyze Carry Flag CF
  - SLTU (Unsigned)
    - Less[0] ← Set ← ~CF
  - SLT (Signed)
    - Less[0] ← Set ← Sign xor Overflow

Arithmetic Logic Unit Design



[1]

1-bit ALU for the **MSB bit** with overflow detection and set generation

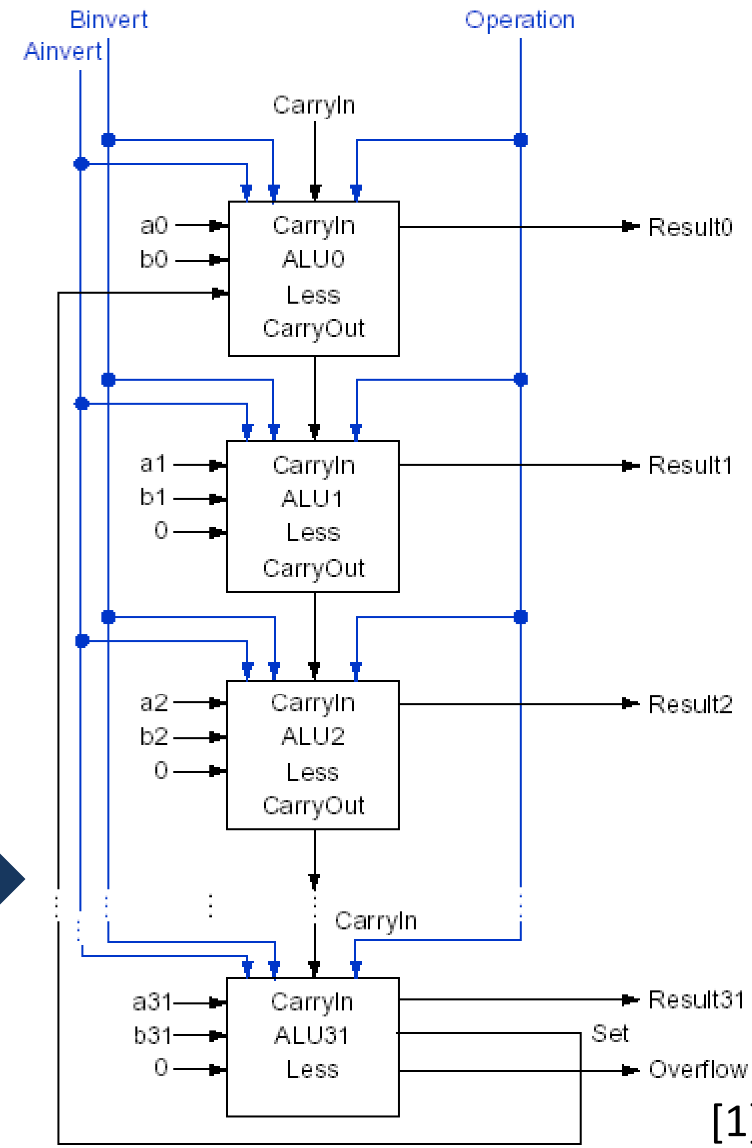
32-bit ALU built with 32 x 1-bit ALUs

Less[31:1] = 0, Less[0] = Set form ALU31

Subtraction operations: CarryIn=Binvert = 1

Addition or logical operations: CarryIn=Binvert = 0

We can simplify the control: CarryIn = Binvert = Bnegate



[1]

Arithmetic Logic Unit Design

- Additional operations: test for Conditional Branch Instructions

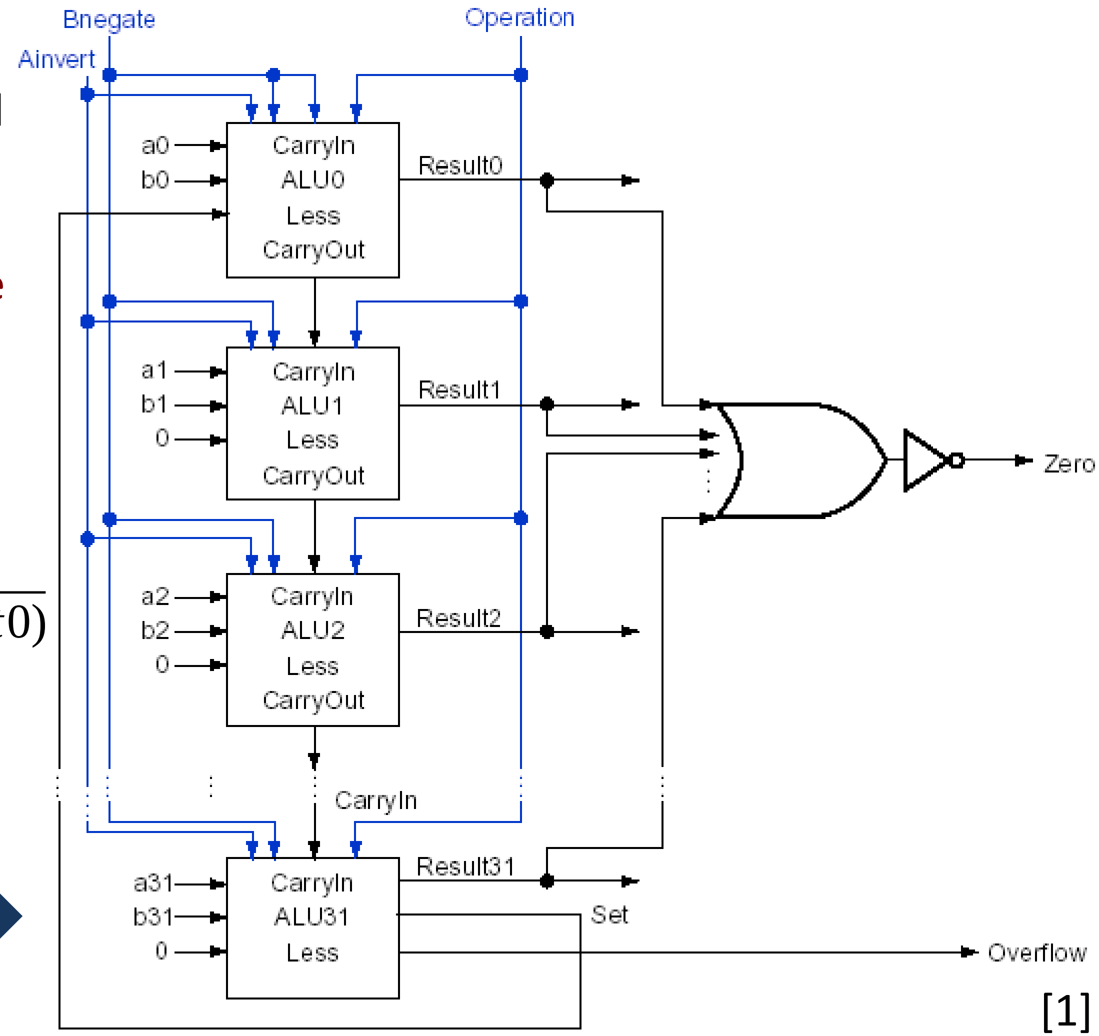
beq – Branch if 2 registers are equal
bne – Branch if 2 registers are not equal

Equality test:
→ subtract b from a and then test if the result is 0

Add specific hardware to test if the result is 0: NOR (negated OR)

$$Zero = \overline{(Result31 or \ldots or\ Result1 or\ Result0)}$$

Final 32-bit ALU with Zero detector, overflow and Set-Less

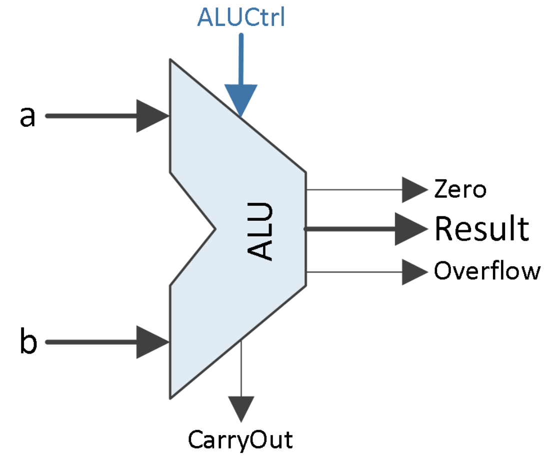


[1]

Arithmetic Logic Unit Control

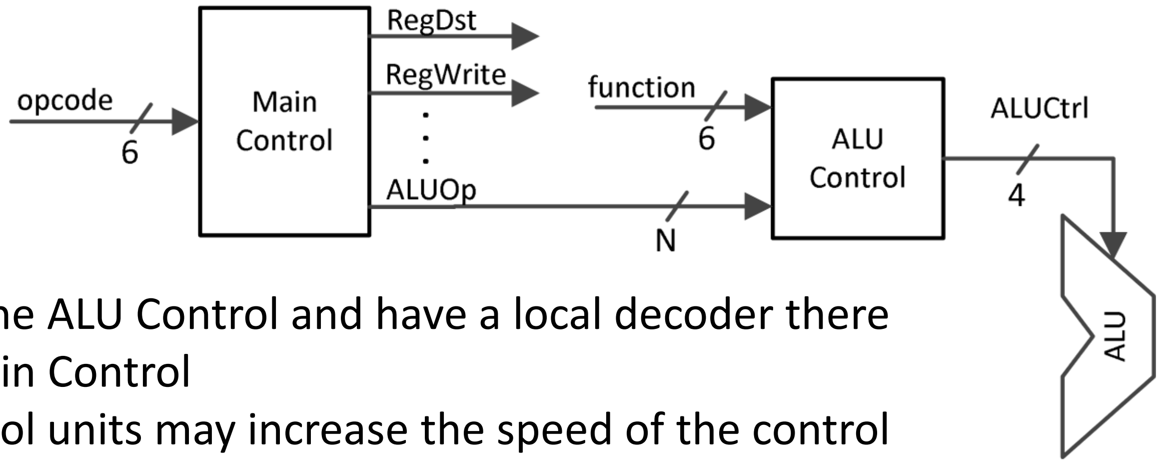
ALU control lines – ALUCtrl				ALU Operation
Ainvert	Bnegate	Operation		
0	0	0	0	AND
0	0	0	1	OR
0	0	1	0	ADD
0	1	1	0	SUBTRACT
0	1	1	1	SET ON LESS THAN
1	1	0	0	NOR

ALU Control Lines and the corresponding ALU operation



ALU Symbol

ALU Control Unit Design



- Multilevel decoding
  - Hierarchical control
  - Pass the function field to the ALU Control and have a local decoder there
  - Reduces the size of the Main Control
  - Using several smaller control units may increase the speed of the control unit

Arithmetic Logic Unit Control

Instruction Opcode	ALUOp	Instruction Operation	Function Field	Desired ALU Operation	ALU Control
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	and	100100	and	0000
R-type	10	or	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

MIPS-lite ALU Control

ALU Control Inputs

- 6-bit function field from the R-type instruction format
- 2-bit ALUOp given by the Main Control, according to opcode field of the instructions. The size of ALUOp can be increased if more MIPS instructions are implemented.

Additional MIPS ALU Requirements

- Multiplication
  - mult, multu – signed and unsigned 32-bit multiplication
  - Paper and pencil unsigned example: 1000 (8) * 1001 (9) = 0100 1000 (72)

Multiplicand					1	0	0	0
Multiplier					1	0	0	1
					1	0	0	0
				0	0	0	0	
			0	0	0	0		
		1	0	0	0			
Product	0	1	0	0	1	0	0	0

! m bits * n bits → m + n bits result

Binary makes it easy:

- Multiplier i bit = 0 → place 0 (0 x multiplicand)
- Multiplier i bit = 1 → place a copy (1 x multiplicand)

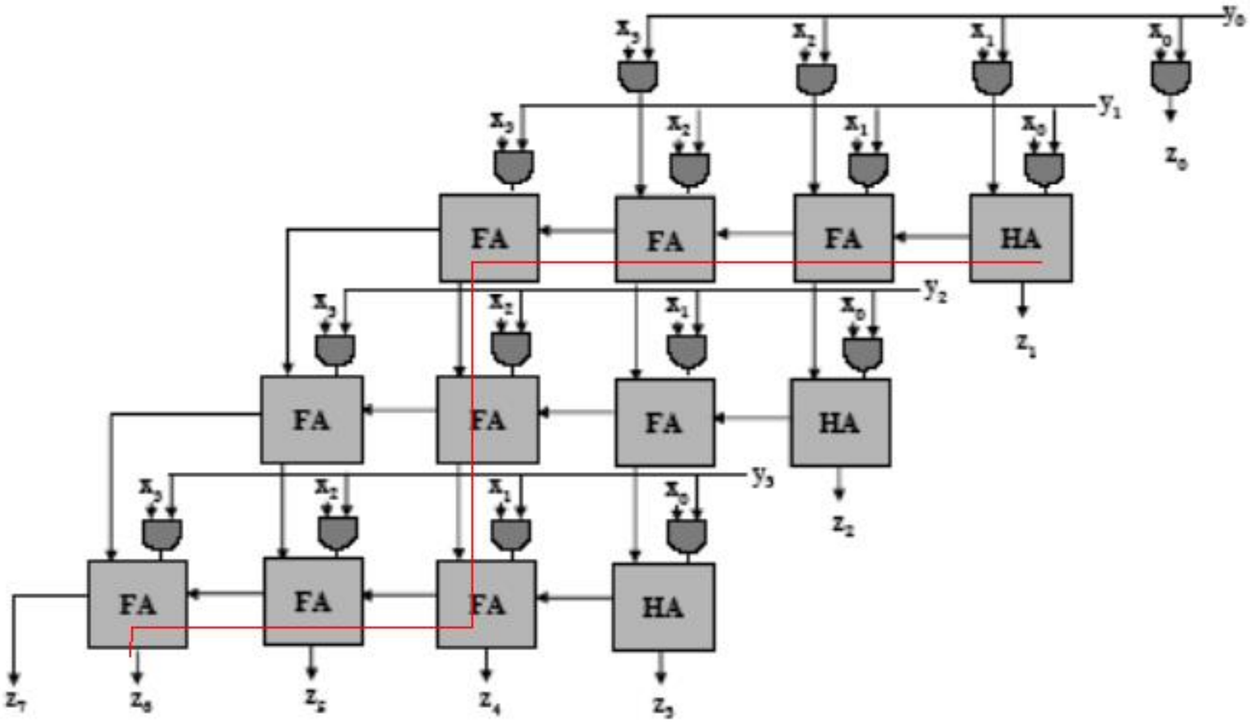
Multiplication is just a lot of additions and shifts!

Multiplier implementations:

- Combinational
- Pipelined
- Multi-Cycle (shift-add cycles)

Additional MIPS ALU Requirements

- Unsigned combinational multiplier

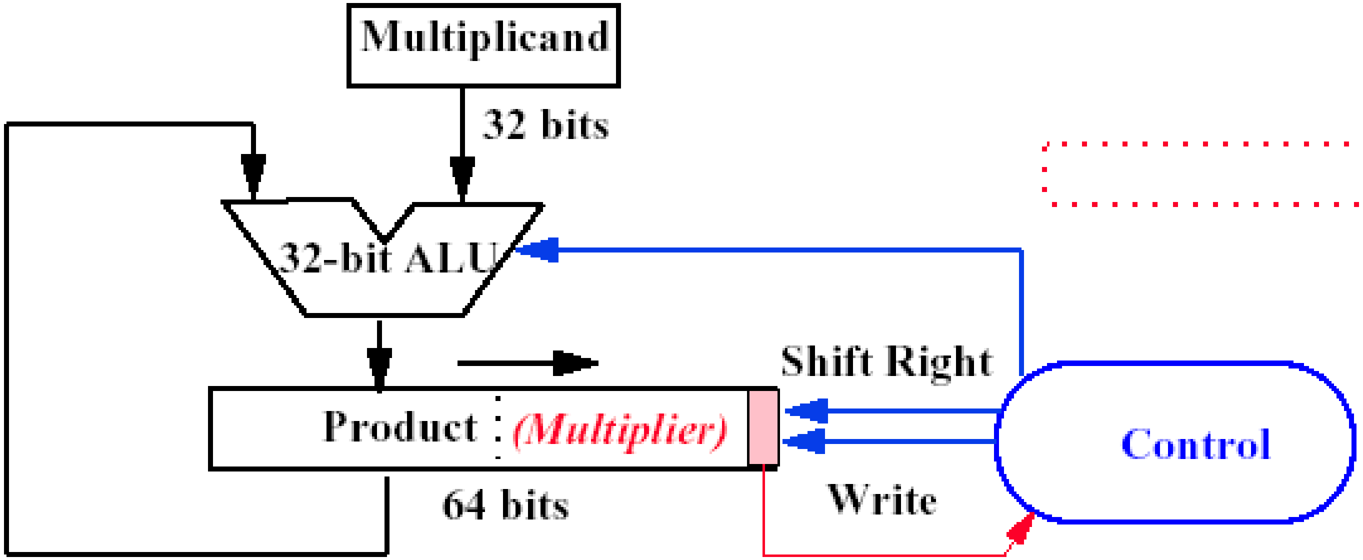


Ripple Carry Array Multiplier

- At each stage shift left X (multiplicand) (<< 1 = x 2)
- Use next bit of Y (multiplier) to determine whether to add in the shifted multiplicand or 0
- Accumulate the partial products
- Critical path is marked in red
- Pipelined versions improve the throughput of the multiplier

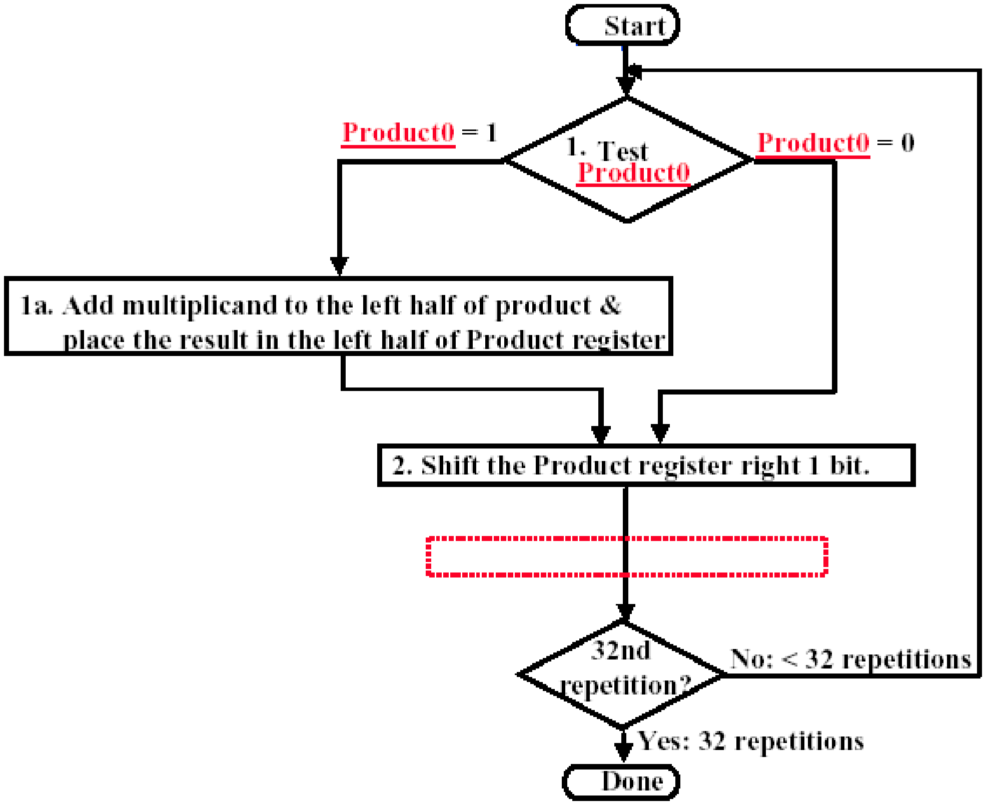
Additional MIPS ALU Requirements

- Multi-cycle (shift-add) Multiplier
  - Implements the multiplication algorithm for binary numbers
  - No separate multiplier register
  - Multiplier placed on right side of 64-bit Product register
  - It has N (number of bits) iterations for summing up the partial products



Multi-cycle (shift-add) Multiplier block diagram

Additional MIPS ALU Requirements



Multiplication Algorithm ASM chart

Observations:

- 2 steps per bit because Shift by Shift register (Product & Multiplier)
- MIPS registers Hi and Lo are left and right half of Product
- MIPS instruction multu places the product in the Hi and Lo registers

Additional MIPS ALU Requirements

Example: 0010 * 0011

Iteration	Step	Multiplicand	Product	Next
0	Init	0010	0000 0011	Product(0)='1' → add
1	1a		0010 0011	Shift Right
	2	0010	0001 0001	Product(0)='1' → add
2	1a		0011 0001	Shift Right
	2		0001 1000	Product(0)='0' → no add
3	1a		0001 1000	Shift Right
	2		0000 1100	Product(0)='0' → no add
4	1a		0000 1100	Shift Right
	2		0000 0110	Done

What about signed multiplication?

1. Easiest solution is to make both positive & remember whether to complement product when done (leave out the sign bit, run for 31 steps).
2. Apply definition of 2's complement: need to sign-extend partial products and subtract at the end.
3. Booth's algorithm.

Additional MIPS ALU Requirements

- Booth's algorithm
  - an elegant way to multiply signed numbers using the same hardware as before and save cycles
  - can handle multiple bits at a time
- Motivation for Booth's Algorithm
  - Booth algorithm gives a procedure for multiplying binary integers in 2's complement representation
  - Originally invented for Speed (when shift was faster than add)
  - Idea: string of 1's …011…10… has as value the sum

$$2^n + 2^{n-1} + \ldots + 2^m = 2^{n+1} - 2^m$$

```
    -1
+10000
------
 01111
```

  - Replace a string of 1s in the multiplier with an initial subtract when we first see a one and then later add after the last one

Current bit	Bit to the right	Explanation	Example	Op
1	0	Begins run of 1s	0001111000	sub
1	1	Middle of run of 1s	0001111000	none
0	1	End of run of 1s	0001111000	add
0	0	Middle of run of 0s	0001111000	none

Additional MIPS ALU Requirements

- Booth's algorithm
  1. Depending on the current and previous bits, do one of the following:

     00: Middle of a string of 0s – no arithmetic operations.

     01: End of a string of 1s – add the multiplicand to the left half of the product.
     10: Beginning of a string of 1s – subtract the multiplicand from the left half of the product.
     11: Middle of a string of 1s – no arithmetic operation
  2. As in the previous algorithm, shift the Product register right (arithmetic) 1 bit
- Booth Multiply
  - Modify Step 1 of the Shift/Add Multiply algorithm to consider 2 bits of the multiplier:
  - Instead of two alternatives, now there are four
    - The current bit and the bit to the right (i.e., the current bit of the previous step)
  - Modify Step 2 of Shift/Add Multiply algorithm to sign extend when the product is shifted right (arithmetic right shift, rather than logical right shift) because the product is a signed number.
  - Shift/Add Multiply algorithm and Booth share the same hardware, except Booth requires one extra flip-flop to remember the bit to the right of the current bit in the product register – which is the bit pushed out by the preceding right shift

Additional MIPS ALU Requirements

- Booth Example 1: (2 * 7)

  Multiplicand m = 0010

  Product p = 0000 0111

Iteration	Multiplicand	Product	LSB-1	Next
0. Init	0010	0000 0111	0	10 → sub
1a. P = P – m	-m = 1110	1110 0111	0	Shift Right Arithmetic
1b.	0010	1111 0011	1	11 → nop, shift
2.	0010	1111 1001	1	11 → nop, shift
3.	0010	1111 1100	1	01 → add
4a.	0010	0001 1100	1	Shift
4b.	0010	0000 1110	0	Done

0000 1110 = 14

Additional MIPS ALU Requirements

- Booth Example 2: (2 * -3)

  m = 0010 3=0011, -3 = 1101

  p = 0000 1101

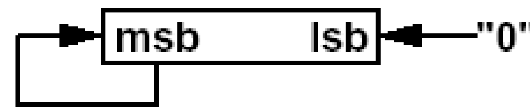
Iteration	Multiplicand	Product	LSB-1	Next
0. Init	0010	0000 1101	0	10 → sub
1a. P = P – m	-m = 1110	1110 1101	0	Shift Right Arithmetic
1b.	0010	1111 0110	1	01 → add
2a.		0001 0110	1	Shift Right Arithmetic
2b.		0000 1011	0	10 → sub
3a. P = P – m	-m = 1110	1110 1011	0	Shift Right Arithmetic
3b.	0010	1111 0101	1	11→ nop
4a.	0010	1111 0101	1	Shift Right Arithmetic
4b.	0010	1111 1010	1	Done

1111 1010 = -6

Additional MIPS ALU Requirements

- Shifters
  - sll, srl, sra MIPS instructions – shift left/right/right arithmetic by 0 to 31 bits

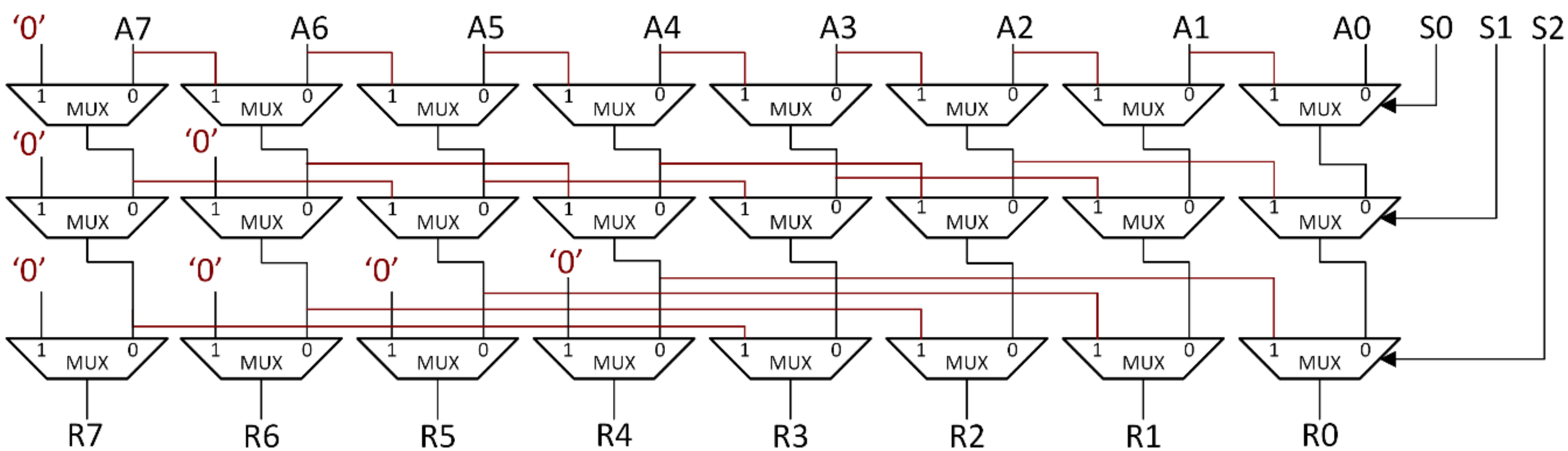
Logical: value shifted in is always "0"



Arithmetic: on right shift the sign bit is extended

Note: these are single bit shifts. Instructions can request 0 to 31 bits to be shifted

Combination shifter implemented with MUXs

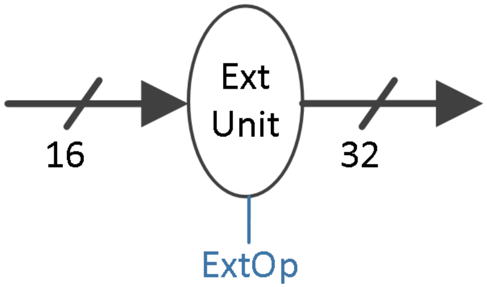


How many levels for a 32-bit shifter?

If we added Right-to-left connections we could support Rotate operations (not implemented in MIPS)

Additional MIPS ALU Requirements

- Sign / Zero Extender
  - MIPS instructions use:
    - Zero Extended (logic) operands   ori: RF[rt] ← RF[rs] | Z_Ext(imm)
    - Sign Extended (arithmetic) operands   lw: RF[rt] ← M[RF[rs] + S_Ext(imm)]



  - Z_Ext (zero extension):
    - The Extender takes a 16-bit number, imm[15:0] and extends it with 0's (extension in unsigned form) if the control line ExtOp = 0
    - Z_Ext(imm16) = $0_{31}.....0_{16}||imm_{15}....imm_0$
  - S_Ext (sign extension):
    - The Extender takes a 16-bit number, imm[15:0] and extends it with the imm_{15} bit if the control line ExtOp = 1
    - S_Ext(imm16) = $0_{31}.....0_{16}||imm_{15}....imm_0$ if $imm_{15} = 0$
    - S_Ext(imm16) = $1_{31}.....1_{16}||imm_{15}....imm_0$ if $imm_{15} = 1$

Problems – Homework

- Design an 8-bit ALU for the following operations: A + B, A – B, IncrA, DecrA, PassA and NegateA. Use a single adder circuit. Show the schematic with control signals and a table with control signal values for the required operations.

- Design an Add/Subtract unit that can work with 8, 16 and 32-bit data. You are given 32x1-bit full adders and the necessary auxiliary circuits. Show the block diagram and the control signals for 4x8-bit, 2x16-bit, 1x32-bit Add/Subtract operations.

- Describe the Booth multiplication method. Give a numerical example.

References

1. D. A. Patterson, J. L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, 5th edition, ed. Morgan–Kaufmann, 2013.
2. D. A. Patterson and J. L. Hennessy, “Computer Organization and Design: A Quantitative Approach”, 5th edition, ed. Morgan-Kaufmann, 2011.
3. MIPS32™ Architecture for Programmers, Volume I: “Introduction to the MIPS32™ Architecture”.
4. MIPS32™ Architecture for Programmers Volume II: “The MIPS32™ Instruction Set”.