ELE 455/555
Computer System Engineering

Section 2 – The Processor

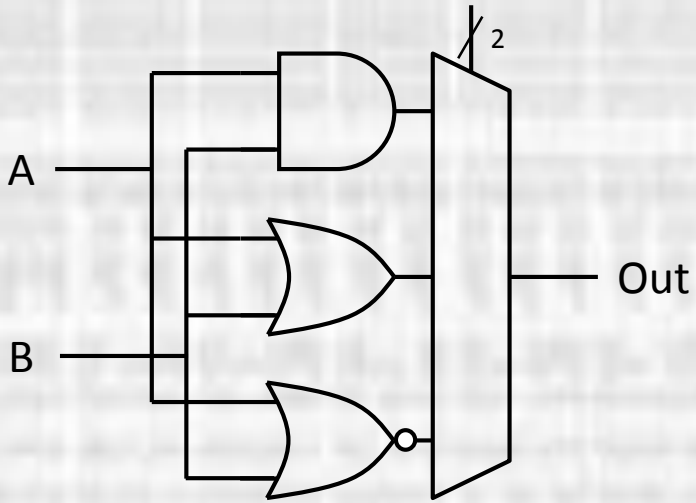
Class 1 – ALU

ALU
Functionality

- MIPS ISA
  - 10 Arithmetic Instructions
    - ADD, ADD Immediate, ADD Imm Unsigned, ADD Unsigned
    - Subtract, Subtract Unsigned
    - Set Less Than, SLT Imm, SLT Imm Unsigned, SLT Unsigned
  - 5 Logical Instructions
    - AND, AND Immediate, NOR, OR, OR Immediate
  - 2 Branch Instructions
    - Branch on Equal, Branch on Not Equal
  - 2 (3) Shift Instructions
    - Shift Left Logical, Shift Right Logical, (Shift Right Arithmetic)

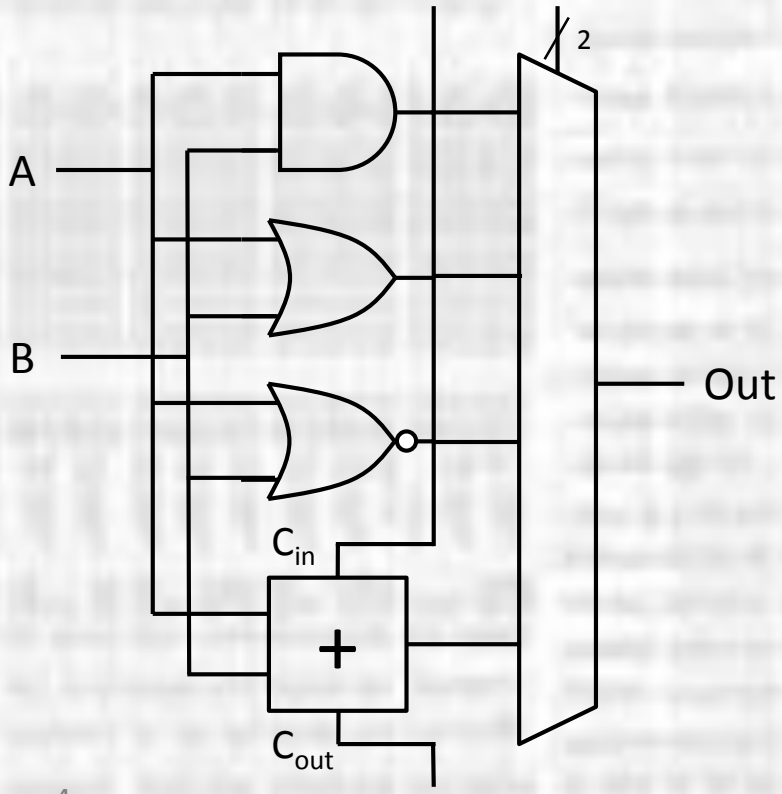
ALU
Functionality

- ALU Implementation
  - 5 Logical Instructions
    - AND, AND Immediate, NOR, OR, OR Immediate
  - 2 inputs A and B



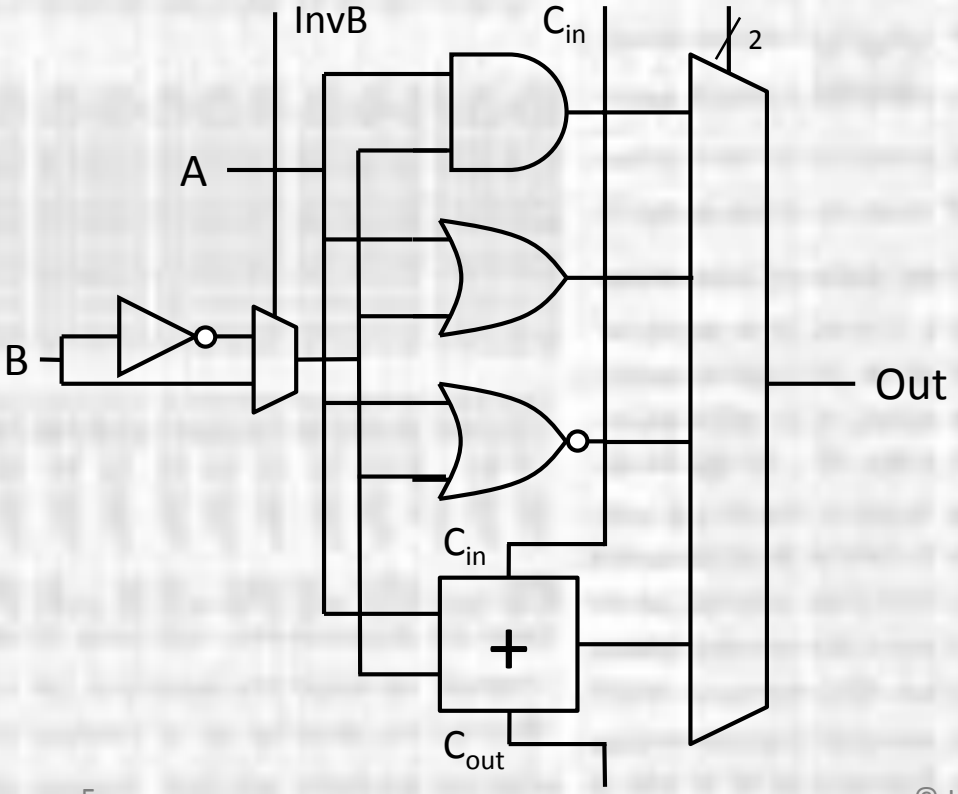
ALU
Functionality

- ALU – Implementation
  - Arithmetic Instructions
    - ADD
    - Inputs: A, B, C_{in}
    - Outputs: Out, C_{out}



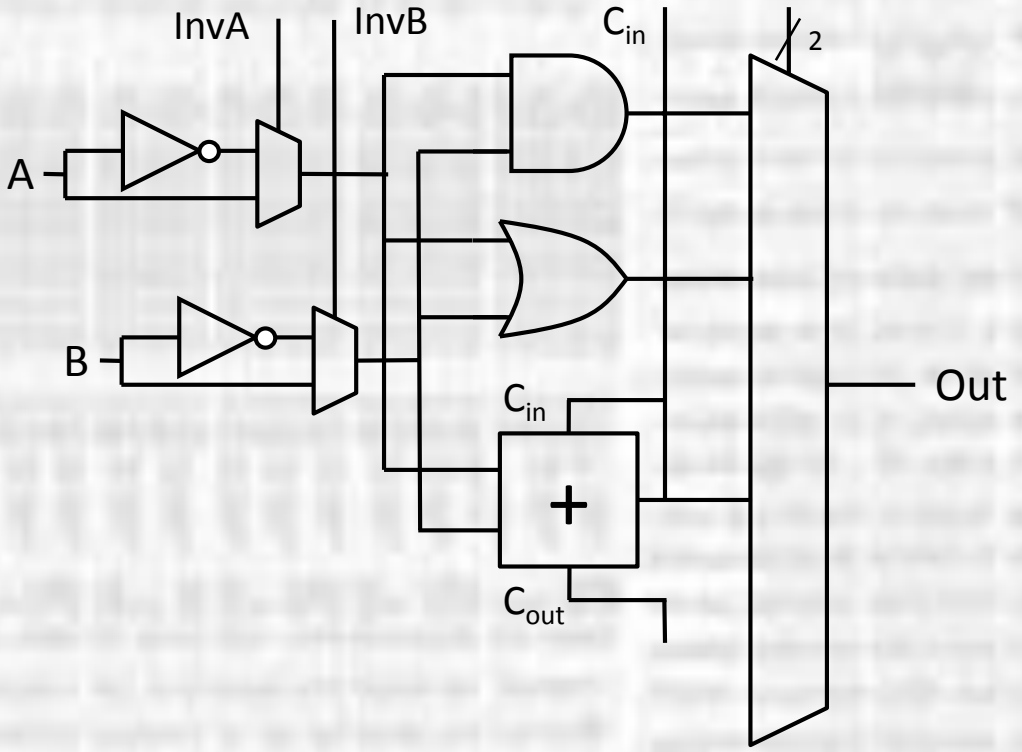
ALU
Functionality

- ALU – Implementation
  - Arithmetic Instructions
    - SUB (2's compliment)
    - $A - B = A + \overline{B} + 1$
    - Invert B and add 1 (C_{inB0}=1)
    - Inputs: A, B, C_{in}
    - Outputs: Out, C_{out}



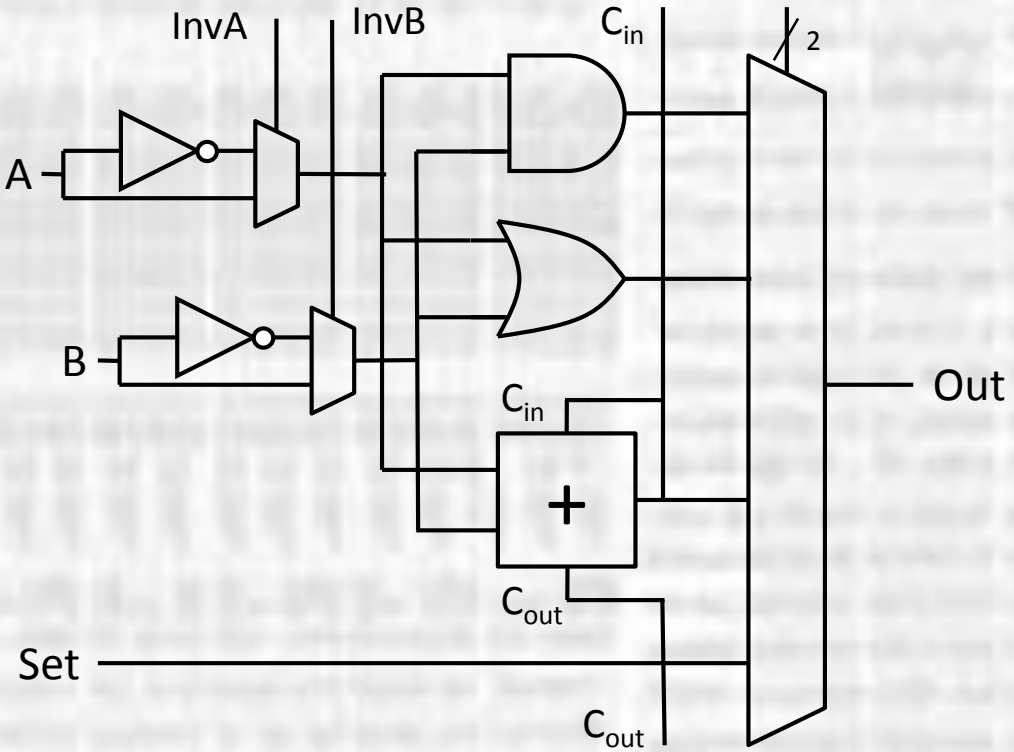
ALU
Functionality

- ALU - Implementation
  - Revisit NOR
    - $\overline{A + B} = \bar{A}\,\bar{B}$



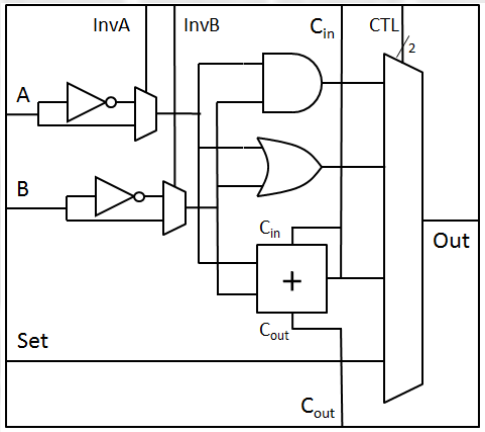
ALU
Functionality

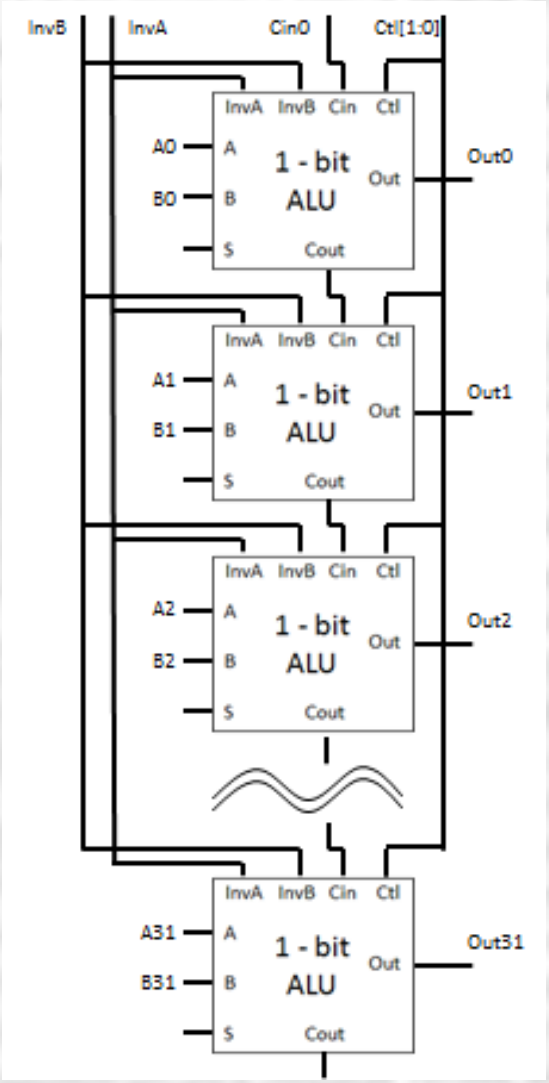
- ALU - Implementation
  - Pre-plan for set function



ALU
Functionality

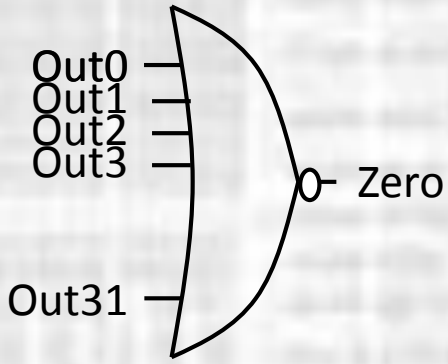
- ALU - Implementation
  - 32 bits





ALU
Functionality

- ALU – Implementation
  - Branches
    - BEQ, BNEQ
    - Need to know if 2 numbers are equal
      - Yes: A – B = 0
      - No: A – B ≠ 0
    - ZERO = NOR of all outputs



ALU
Functionality

- ALU – Implementation
  - Set On Less Than
    - If A < B: Out[0] = 1, Out[31:1] = 0
    - If A ≥ B: Out[31:0] = 0

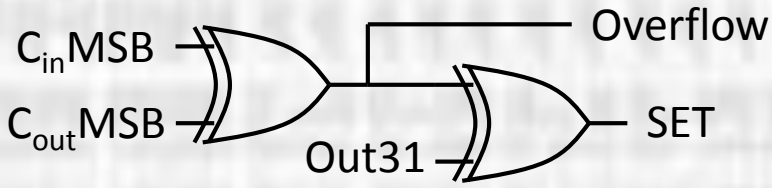
    A < B → (A – B) < 0

    - Subtraction is implemented by addition
      - A – B → A + $\overline{B}$ + 1
    - MSB after subtraction indicates sign
      - MSB = 1 → negative number
      - MSB = 0 → positive number

ALU
Functionality

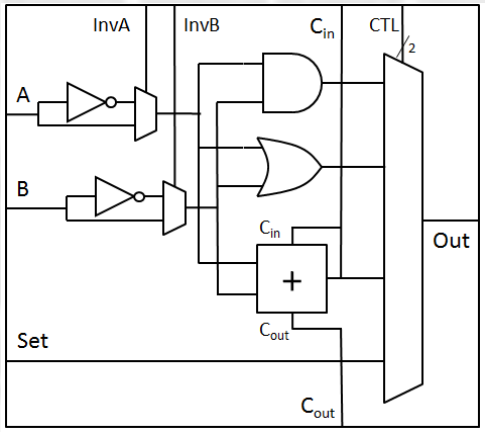
- ALU – Implementation
  - Set On Less Than – cont'd
    - MSB after subtraction indicates sign
      - MSB = 1 → negative number
      - MSB = 0 → positive number
    - Exception: Subtraction (addition) is not valid if overflow occurs
      If overflow occurs, MSB is wrong sign

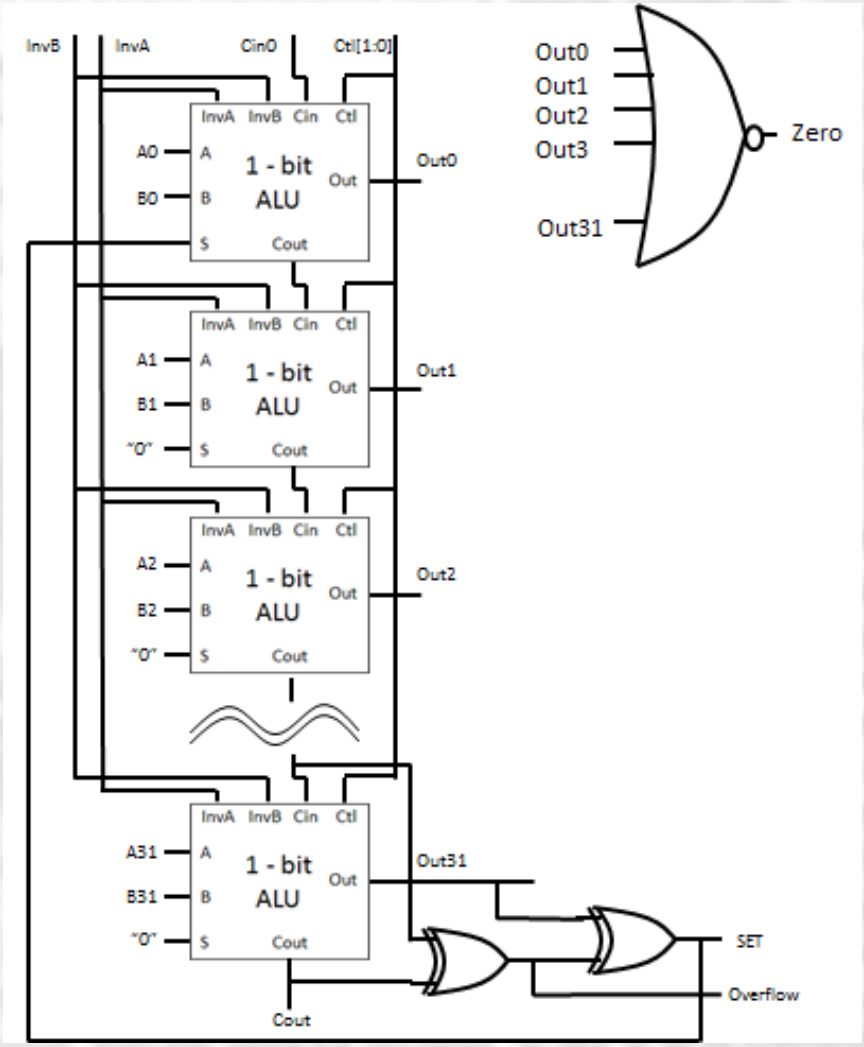
SET: MSB xor OVERFLOW



ALU
Functionality

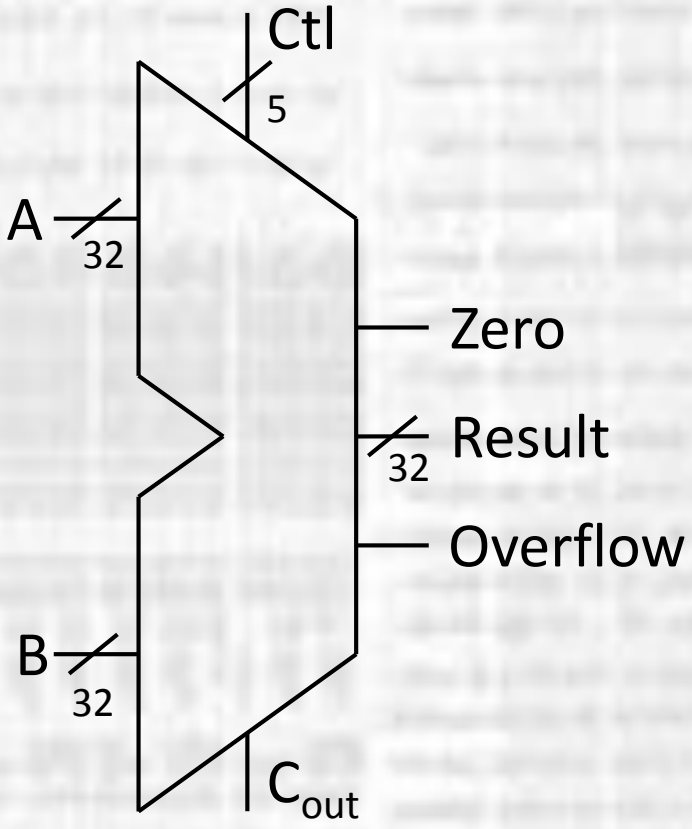
- ALU - Implementation





ALU
Functionality

- ALU – Implementation
  - Control
    - invA
    - invB
    - Cin
    - ctl[1:0]



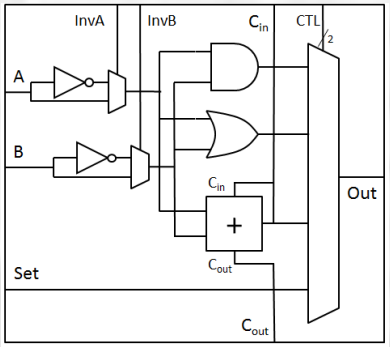
ALU
Functionality

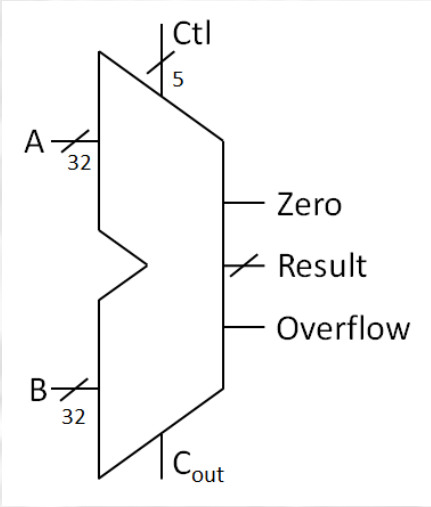
- ALU – Implementation

DeMorgan

Addition

Operation	invA	invB	Cin	ctl[1]	ctl[0]
AND	0	0	x	1	1
OR	0	0	x	1	0
NOR	1	1	x	1	1
ADD	0	0	0	0	1
SUB	0	1	1	0	1
BEQ	0	1	1	0	1
BNE	0	1	1	0	1
SLT	0	1	1	0	0

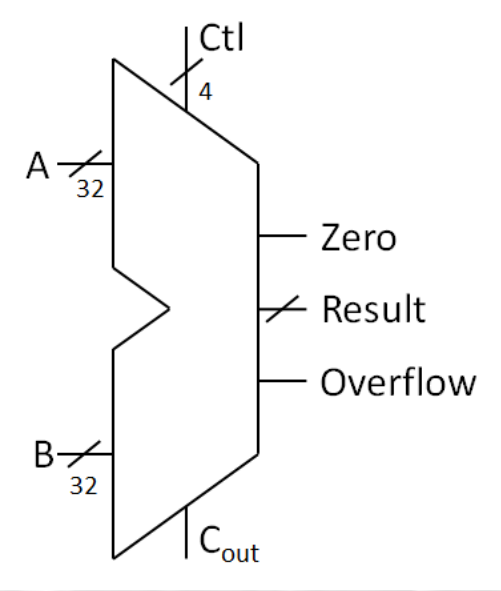




ALU
Functionality

- ALU – Implementation
  - Note: C_{in} and invB are always the same → combine (negB)
    Reduces control lines to 4

Operation	invA	negB	ctl[1]	ctl[0]
AND	0	0	1	1
OR	0	0	1	0
NOR	1	1	1	1
ADD	0	0	0	1
SUB	0	1	0	1
BEQ	0	1	0	1
BNE	0	1	0	1
SLT	0	1	0	0



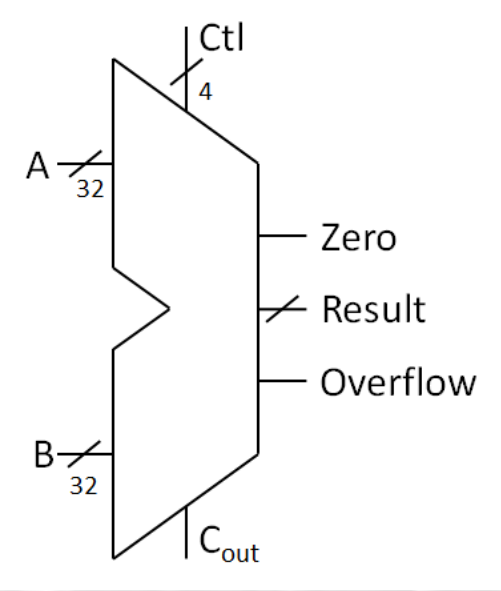
ALU
Functionality

- ALU – Implementation
  - 5 out of 8 instructions involve addition

Current implementation is very slow – why?

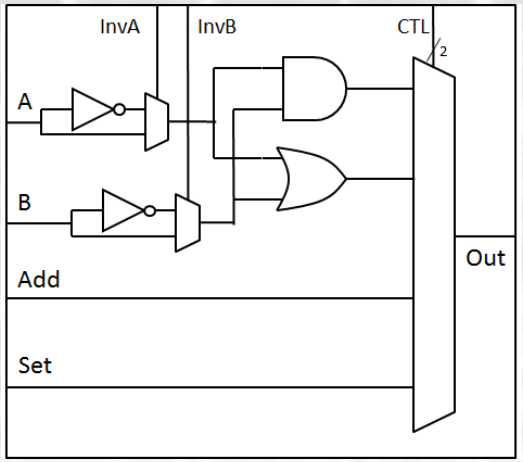
Operation	invA	negB	ctl[1]	ctl[0]
AND	0	0	1	1
OR	0	0	1	0
NOR	1	1	1	1
ADD	0	0	0	1
SUB	0	1	0	1
BEQ	0	1	0	1
BNE	0	1	0	1
SLT	0	1	0	0

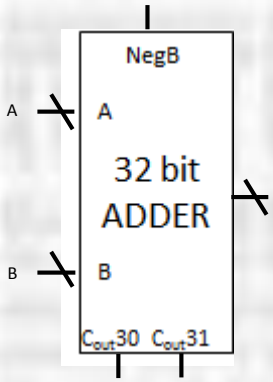
Addition

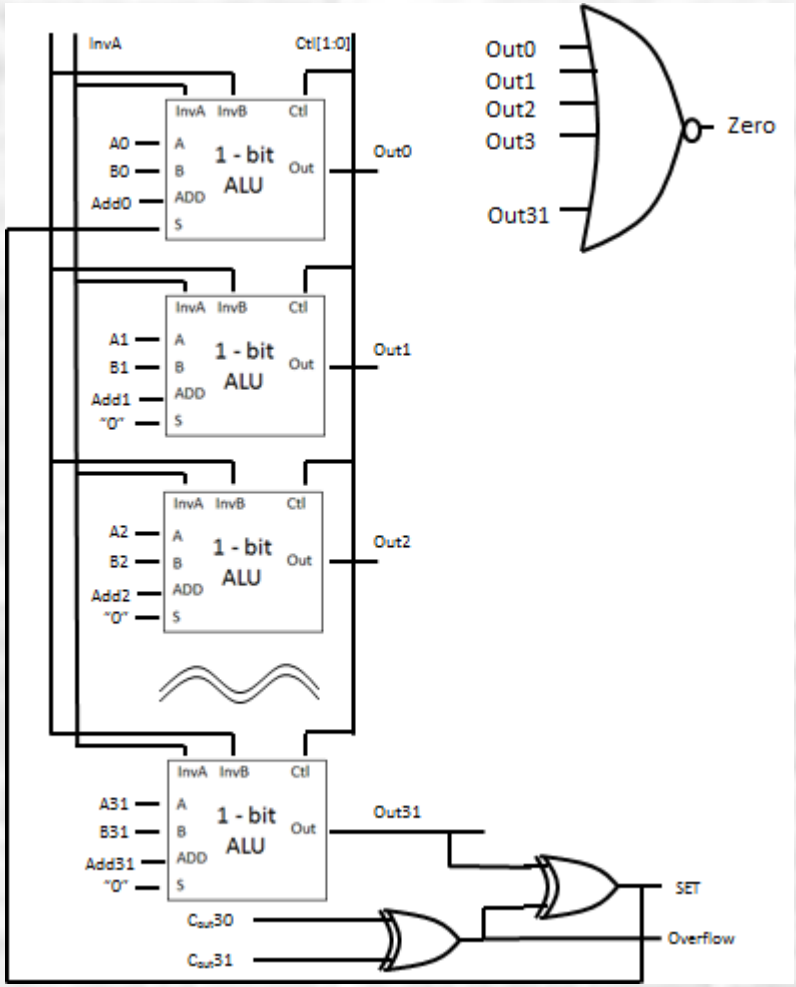


ALU
Functionality

- ALU – Implementation
  - Enhanced Adder

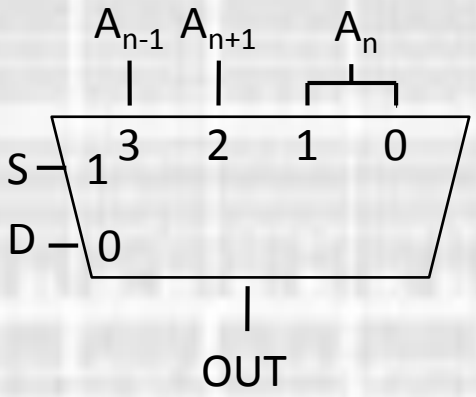






ALU
Functionality

- ALU – Implementation
  - Shift Left Logical, Shift Right Logical
  - Barrel Shifter
    - 32 bits – 5 stages



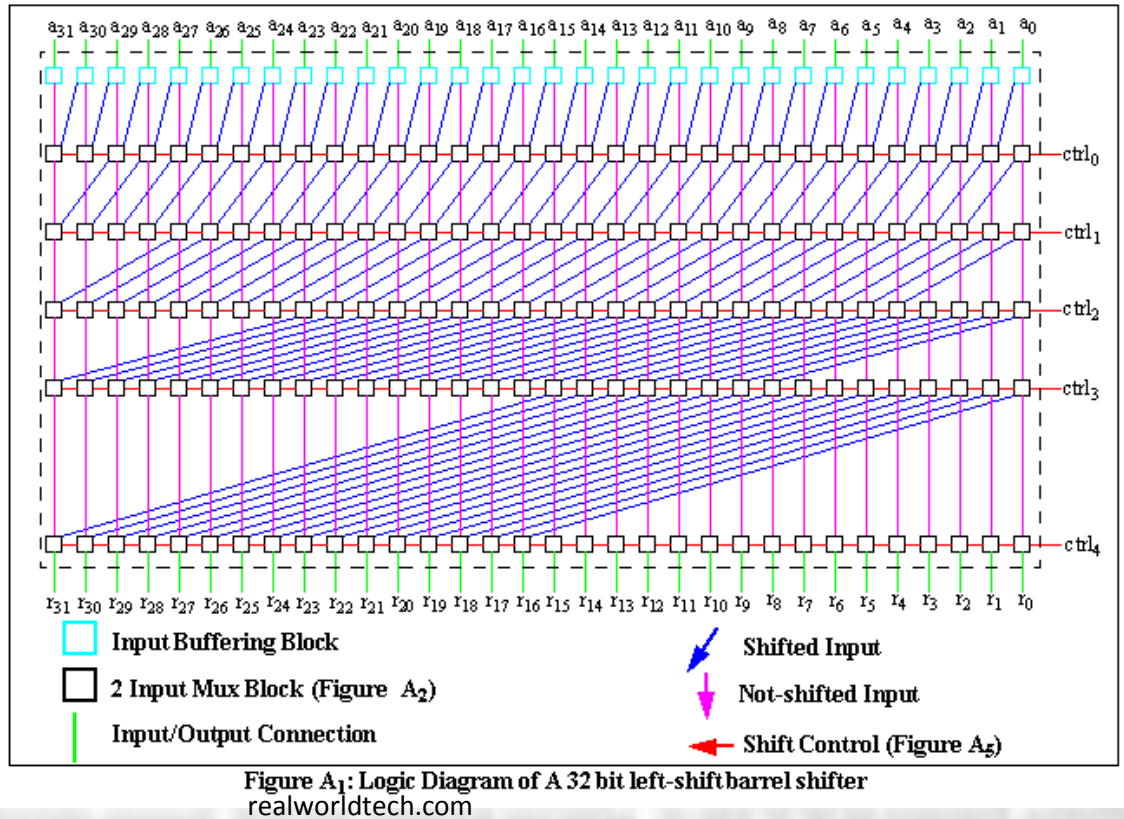
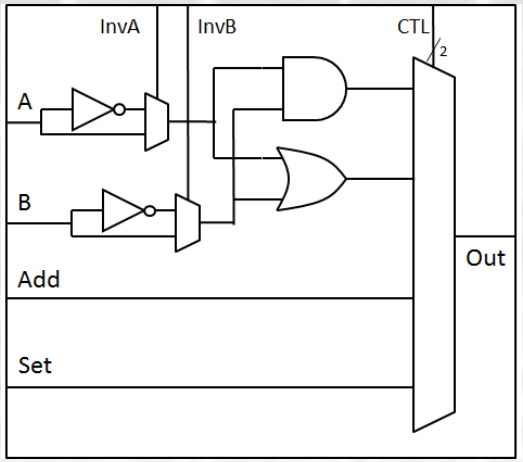
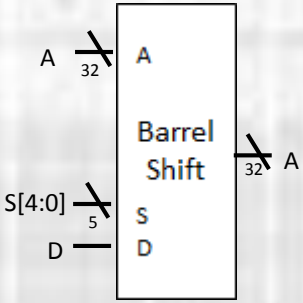


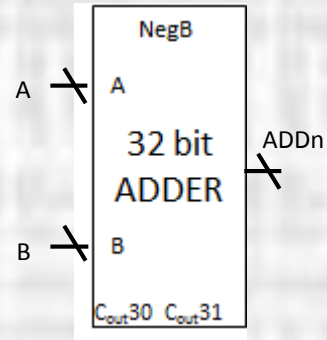
Figure A_1: Logic Diagram of A 32 bit left-shift barrel shifter
realworldtech.com

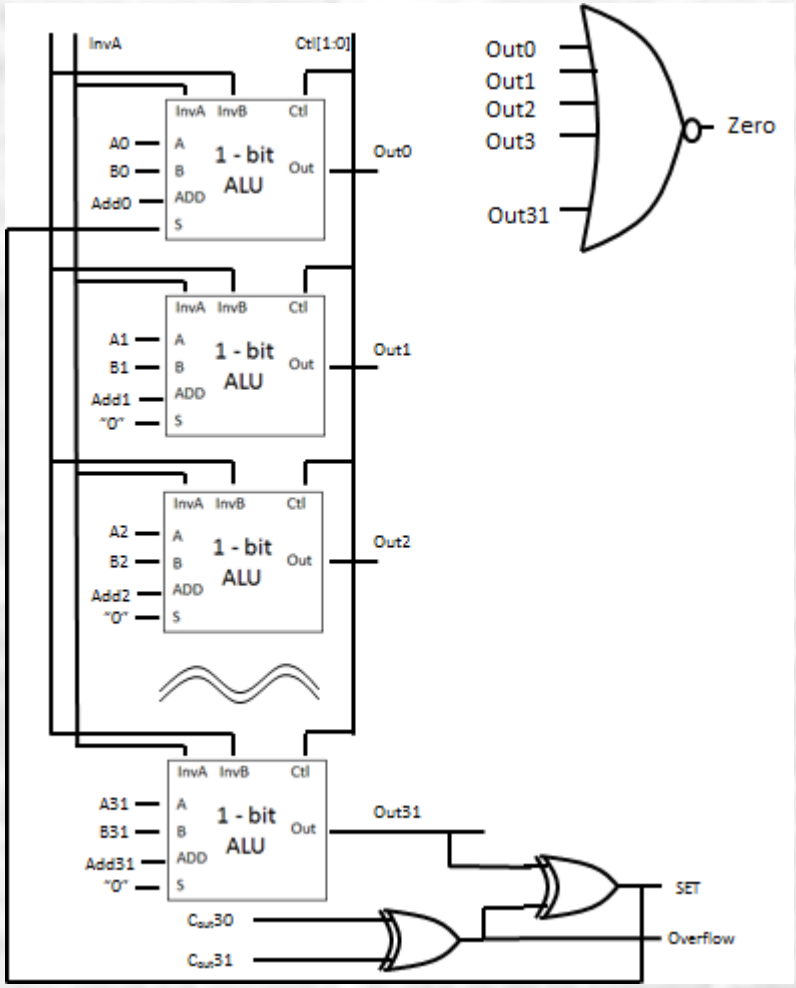
ALU
Functionality

- ALU – Implementation









ALU
Functionality

- Multiplication

```
        0 1 1 1   Multiplicand
      x 0 1 0 1   Multiplier
    -------------
  0 0 1 0 0 0 1 1 Product
```

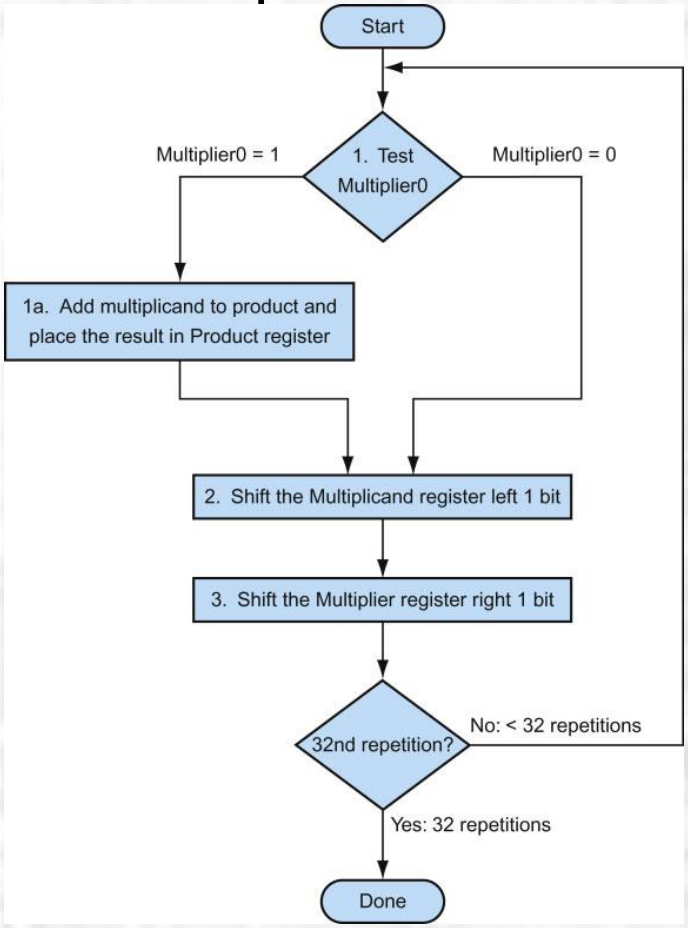
```
                0 0 0 0 0 1 1 1
              x 0 0 0 0 0 1 0 1
                        1 1 1
              -----------------
                0 0 0 0 0 1 1 1
              0 0 0 0 0 0 0 0
            0 0 0 0 0 1 1 1
        + 0 0 0 0 0 0 0 0
          -----------------
          x x x 0 0 1 0 0 0 1 1
```

```
                0 1 1 1
              x 0 1 0 1
              ---------
  0 0 0 0 0 1 1 1
+ 0 0 0 0 0 0 0 0   Add
  ---------------
  0 0 0 0 0 1 1 1   shift

  0 0 0 0 0 0 0
+ 0 0 0 0 0 1 1 1   Add
  ---------------
  0 0 0 0 0 1 1 1   shift

      1 1 1
  0 0 0 1 1 1
+ 0 0 0 0 0 1 1 1   Add
  ---------------
  0 0 1 0 0 0 1 1   shift

  0 0 0 0 0
+ 0 0 1 0 0 1 1 1   Add
  ---------------
  0 0 1 0 0 1 1 1
```

```
                0 1 1 1
              x 0 1 0 1
              ---------
    0 1 1 1
+ 0 0 0 0 0 0 0 0   Add
  ---------------
  0 0 0 0 0 1 1 1   shift

                    SKIP

      1 1 1         shift
    0 1 1 1
+ 0 0 0 0 0 1 1 1   Add
  ---------------
  0 0 1 0 0 0 1 1   shift

                    SKIP
```

ALU
Functionality

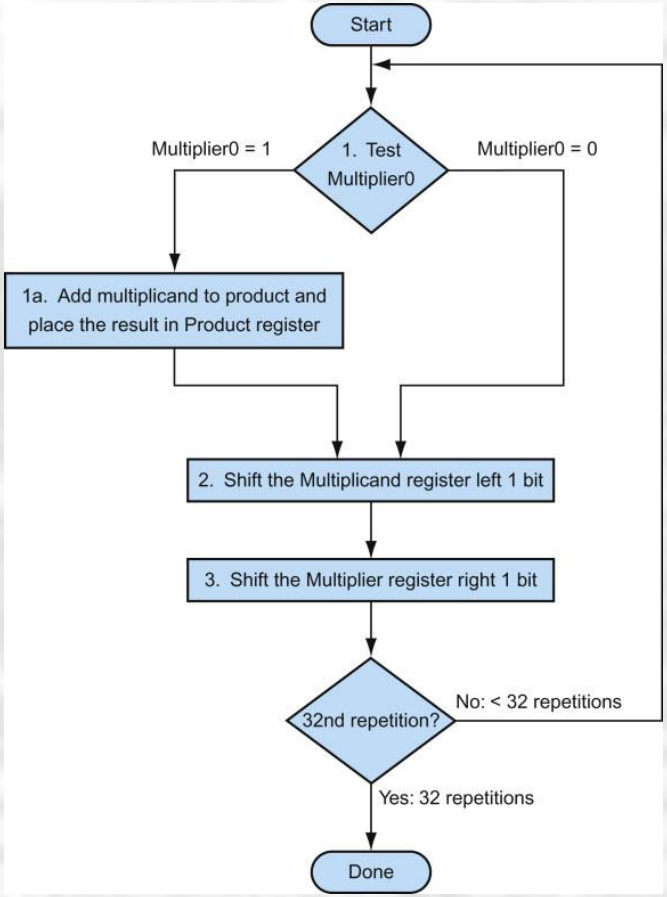
- Multiplication

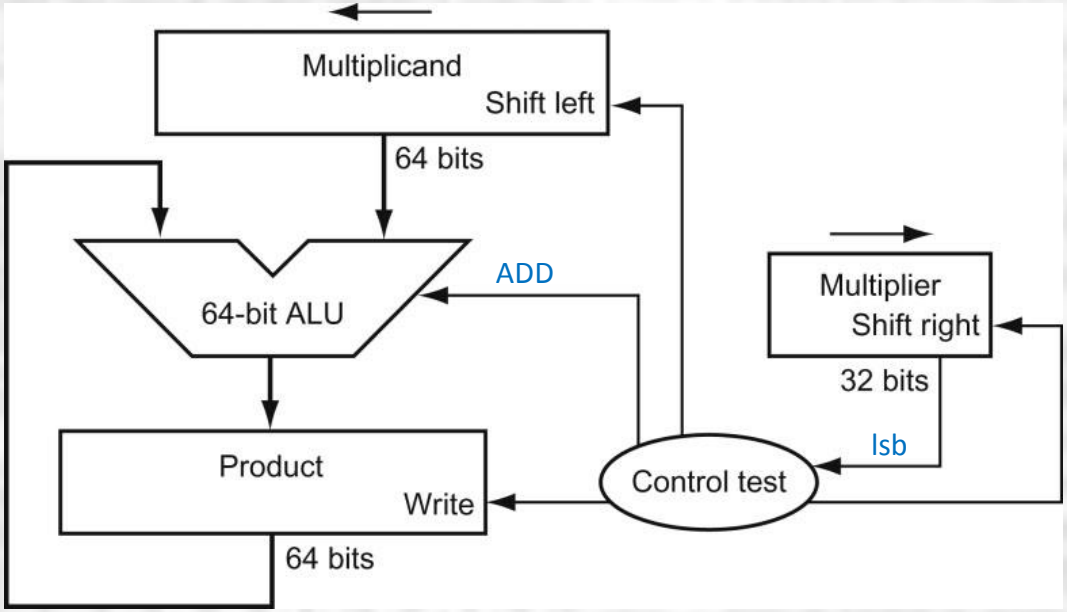


				0	1	1	1	
				x 0	1	0	1	
				0	1	1	1	
0	0	0	0	0	0	0	0	Add
0	0	0	0	0	1	1	1	
			0	1	1	1	0	shift
					0	1	0	
		0	1	1	1	0	0	shift
						0	1	
		1	1	1				Add
		0	1	1	1			
0	0	0	0	0	1	1	1	
0	0	1	0	0	0	1	1	
	0	1	1	1	0	0	0	shift
							0	
0	1	1	1	0	0	0	0	shift
0	0	1	0	0	0	1	1	done

ALU
Functionality

- Multiplication

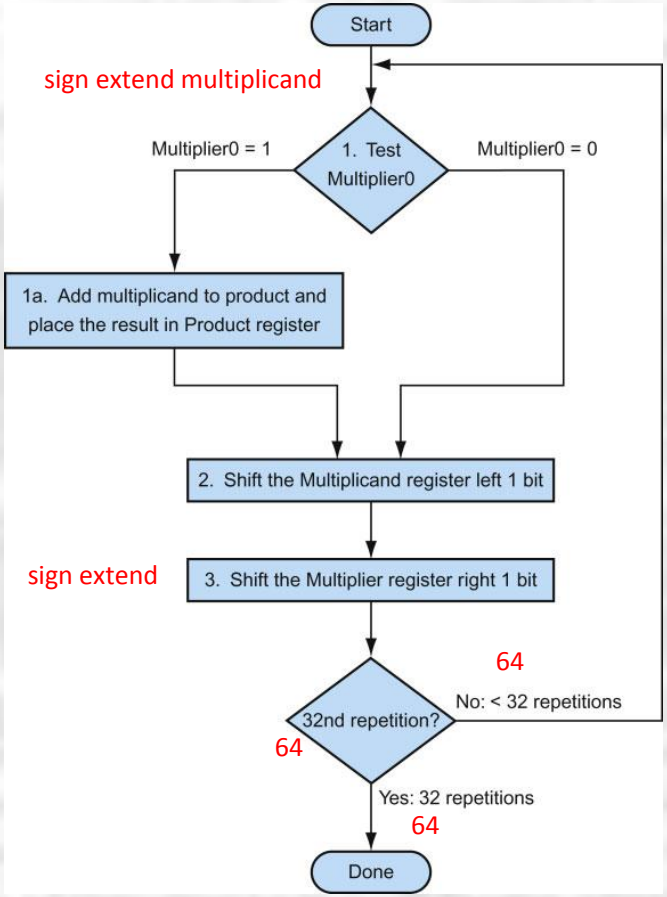


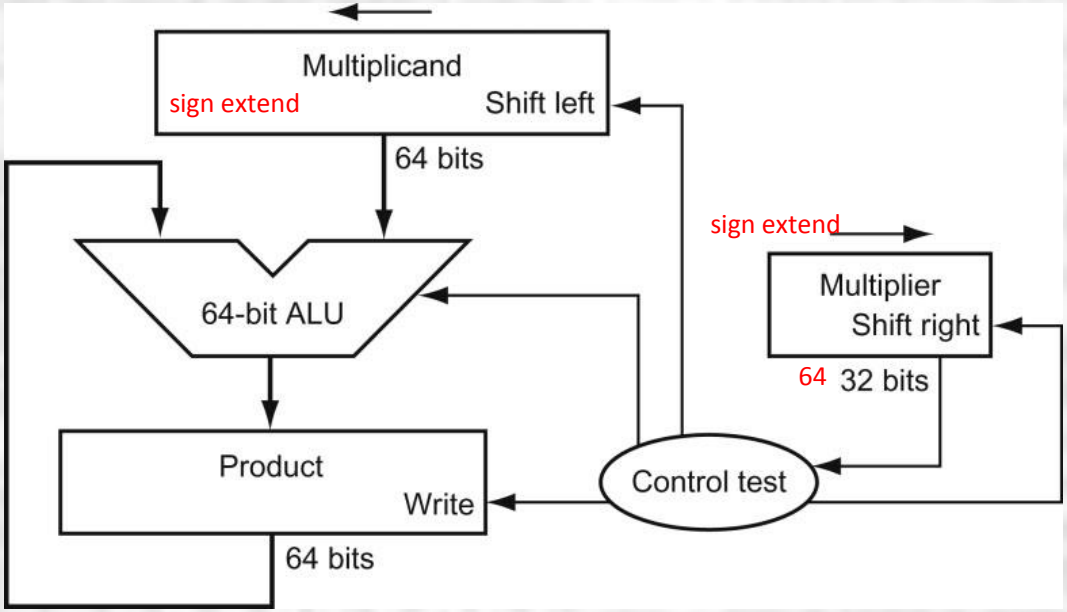


Positive Operands Only

ALU
Functionality

- Multiplication – negative numbers





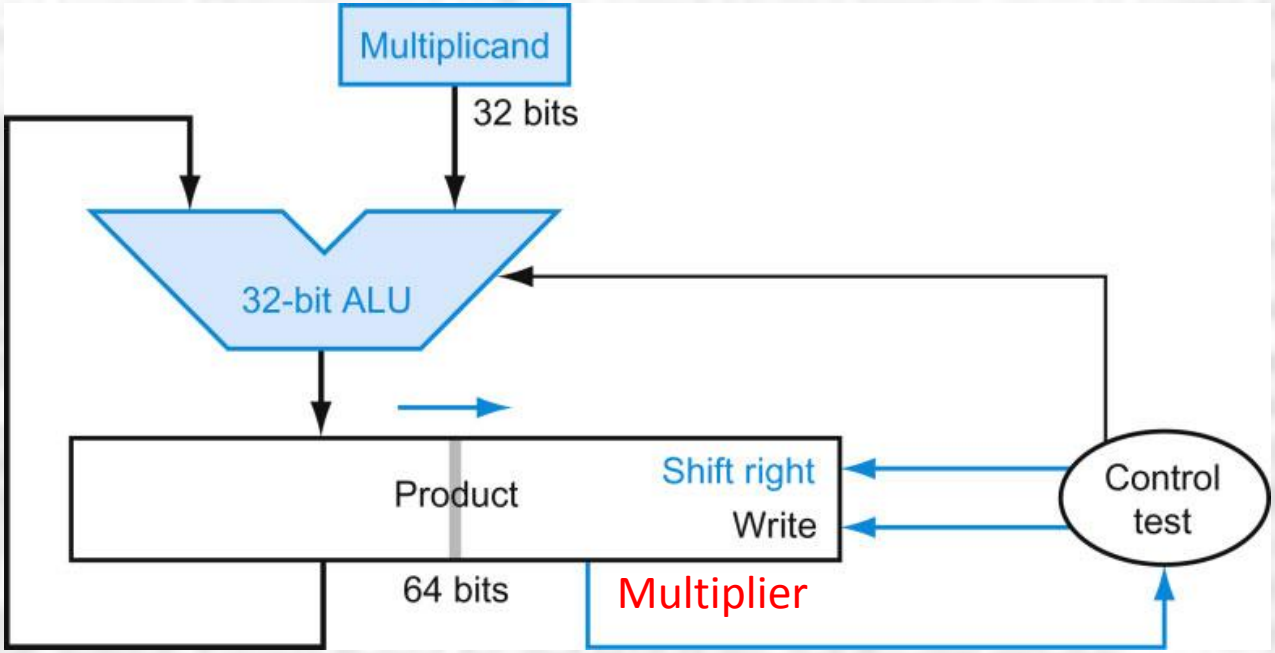
2x timing penalty to support negative numbers

ALU
Functionality

- Multiplication – negative numbers
  - Check signs of multiplier and multiplicand
  - 2's complement anything that is negative
  - Keep track of signs
  - When done:
    - 1 negative → complement the output
    - 0/2 negative → done
  - Penalty
    - no penalty of both positive
    - no penalty if 2 negative → Invert and set Cin=1 for first 2 additions
    - 1 add penalty if 1 negative → Invert and set Cin = 1 for first addition
      → 2's complement output (one additional add)

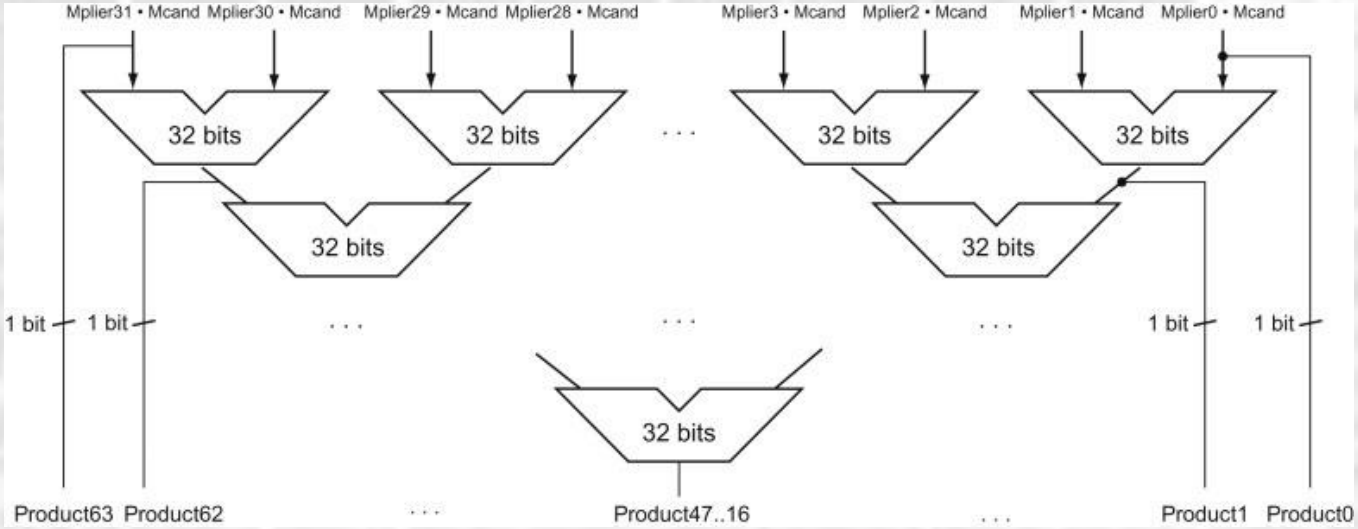
ALU
Functionality

- Multiplication refined



ALU
Extensions

- Multiplication – reality
  - Un-roll the loop
  - Trade HW for speed
    - 31 adders
    - 5 add delays



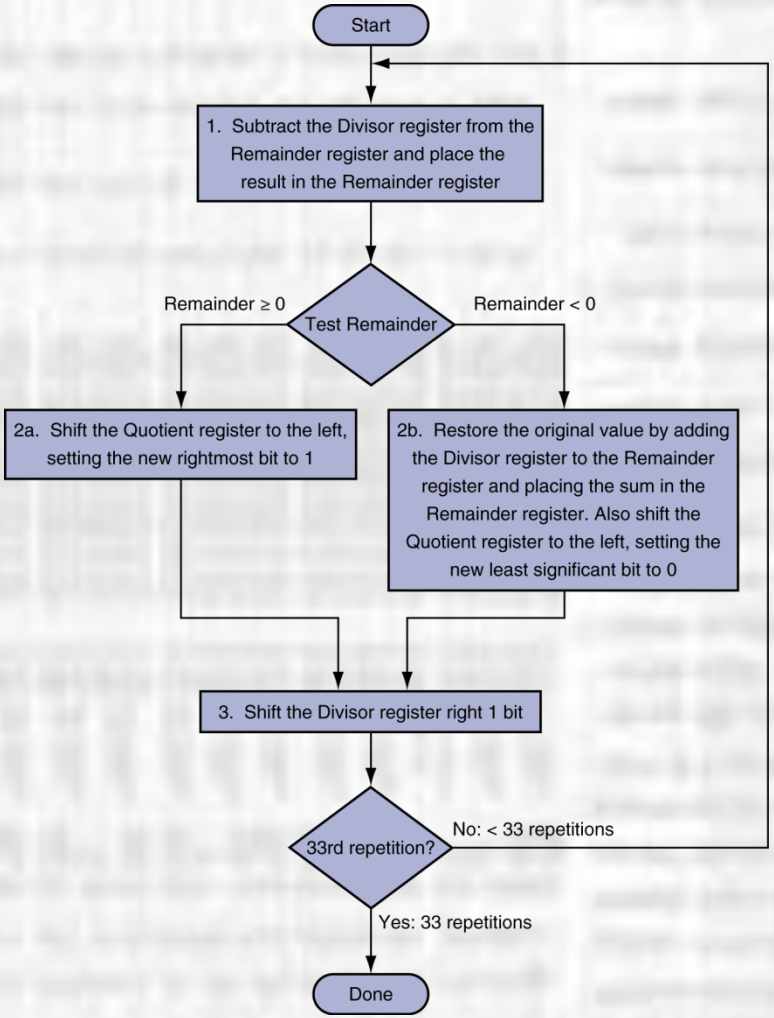
ALU
Functionality

- Division

	0	1	1	1	1	1	1	1	Dividend 0x3F
÷	0	0	0	0	0	1	0	1	Divisor 0x05
	0	0	0	0	1	1	0	0	Quotient
R					0	0	1	1	Remainder

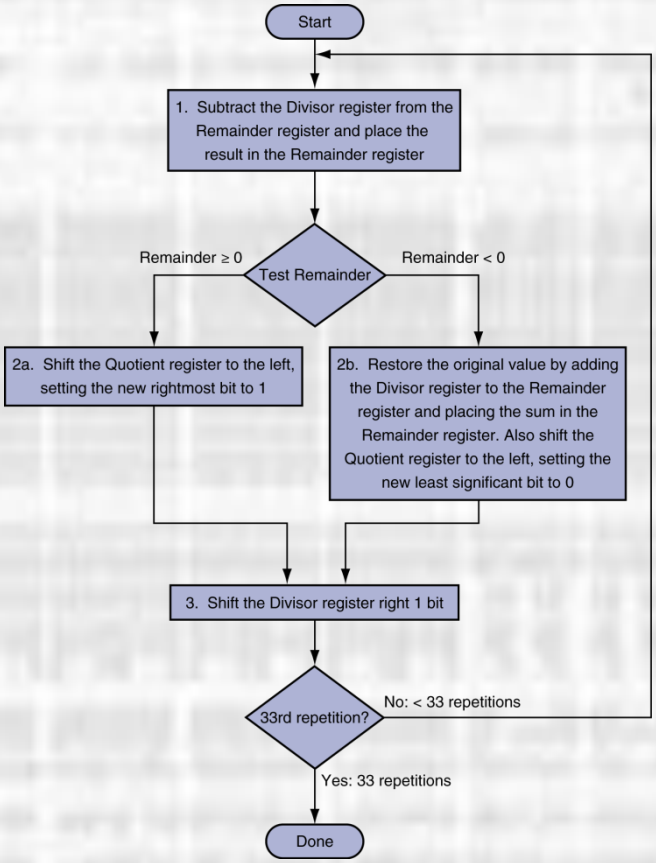
- Restoring Division

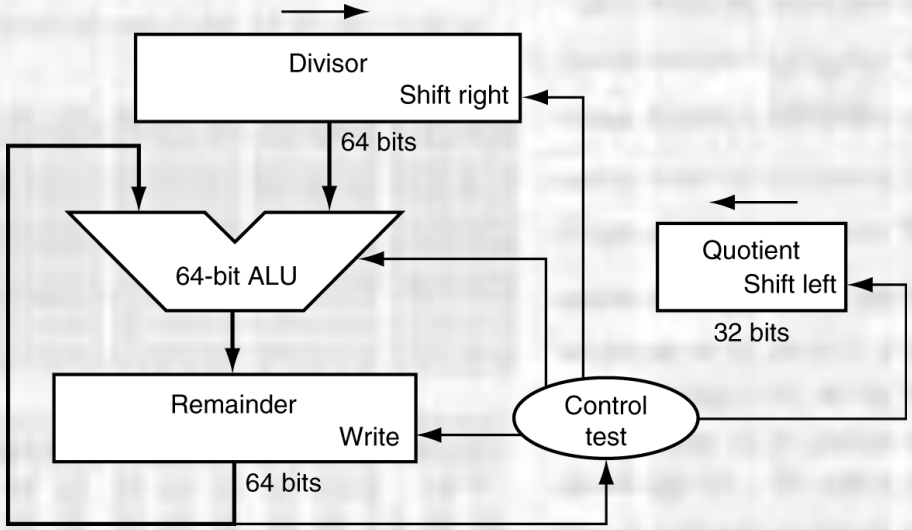
```
                    1  1  0  0   R  0  0  1  1
0  1  0  1 | 0  1  1  1  1  1  1  1
         -     0  1  0  1
         +  1  0  1  1
            0  0  1  0        Positive
               0  1  0  1
            -  0  1  0  1
            +  1  0  1  1
               0  0  0  0        Positive
                  0  0  0  1
               -  0  1  0  1
               +  1  0  1  1
                  1  1  0  0     Negative
               +  0  1  0  1     Correct
                  0  0  0  1
                     0  0  1  1
                  -  0  1  0  1
                  +  1  0  1  1
                     1  1  1  0  Negative
                  +  0  1  0  1  Correct
                     0  0  1  1  Remainder
```



ALU
Extensions

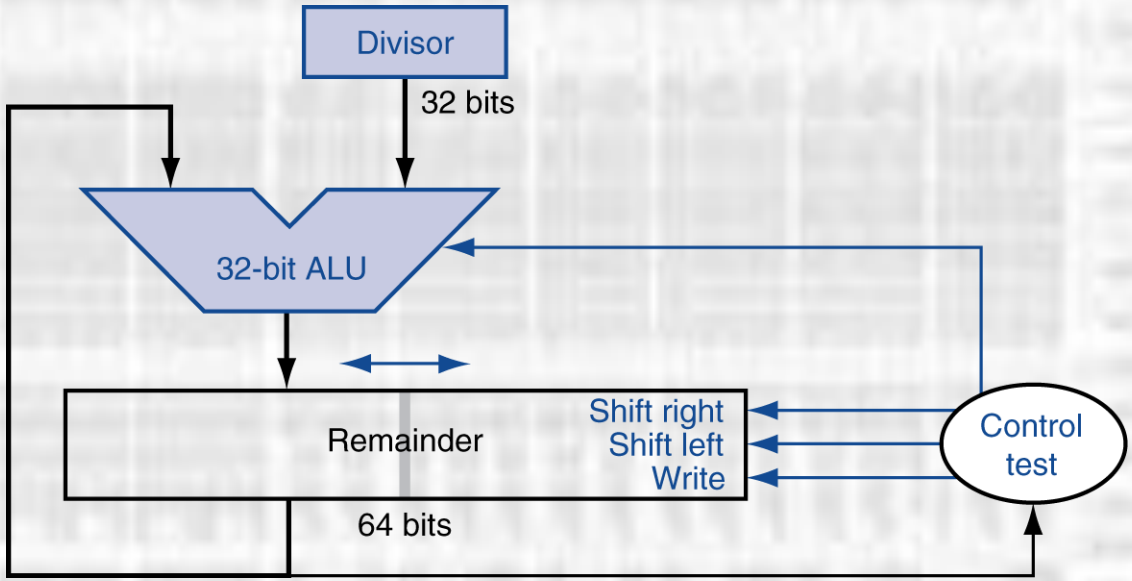
- Division





ALU
Extensions

- Division
  - Optimized



ALU

Extensions

- Floating Point Arithmetic
  - Addition

Consider a 4-digit binary example

$1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)

1. Align binary points

   Shift number with smaller exponent

   $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$

2. Add significands

   $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-}1 = 0.001_2 \times 2^{-1}$

3. Normalize result & check for over/underflow

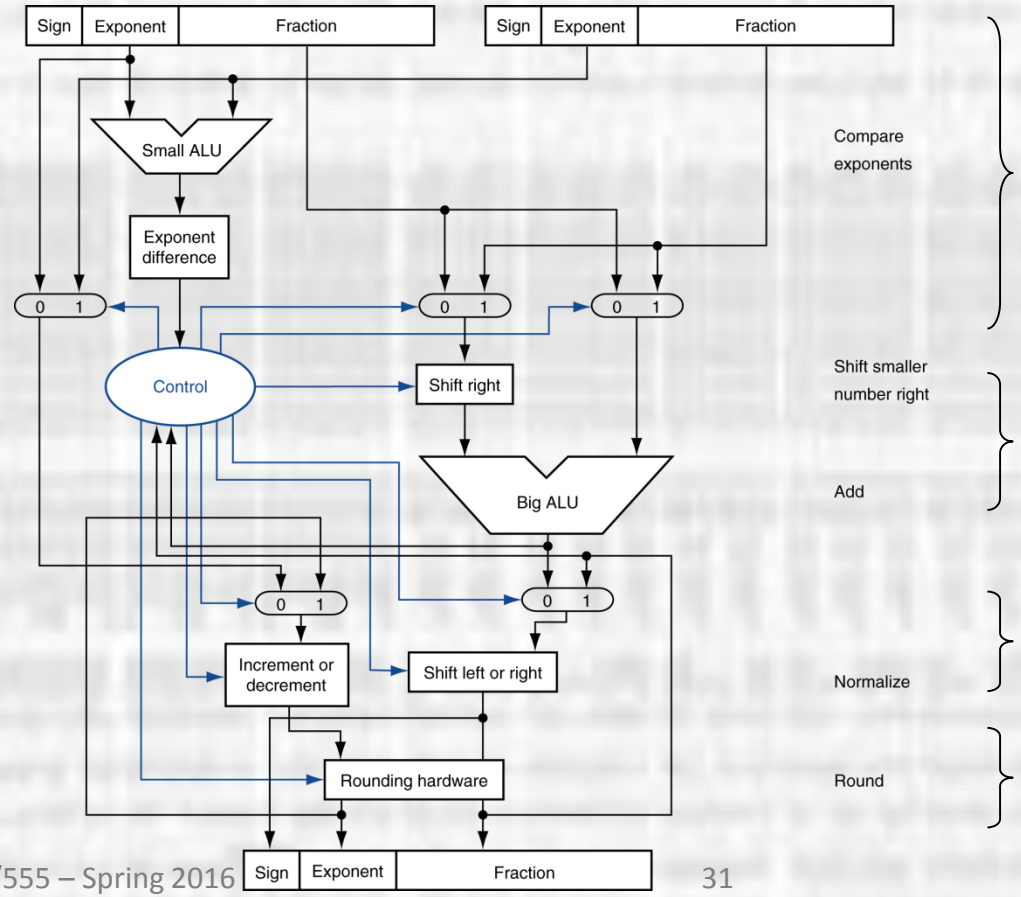
   $1.000_2 \times 2^{-4}$, with no over/underflow

4. Round and renormalize if necessary

   $1.000_2 \times 2^{-4}$ (no change) = 0.0625

ALU
Extensions

- Floating Point Arithmetic
  - Addition



Step 1 - align binary points

Step 2 – add significands

Step 3 – normalize and check for overflow

Step 4 – round and re-normalize

ALU
Extensions

- Floating Point Arithmetic
  - Multiplication

Consider a 4-digit binary example

$1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5 × –0.4375)

1. Add exponents
   Unbiased: –1 + –2 = –3
   Biased: (–1 + 127) + (–2 + 127) = –3 + 254 – 127 = –3 + 127
2. Multiply significands
   $1.000_2 \times 1.110_2 = 1.1102 \Rightarrow 1.110_2 \times 2^{-3}$
3. Normalize result & check for over/underflow
   $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
4. Round and renormalize if necessary
   $1.110_2 \times 2^{-3}$ (no change)
5. Determine sign: +ve × –ve ⇒ –ve
   $-1.110_2 \times 2^{-3} = -0.21875$

ALU
Extensions

- Floating Point Arithmetic
  - Multiplication
  - FP multiplier is of similar complexity to FP adder
    - But uses a multiplier for significands instead of an adder
  - FP arithmetic hardware usually does
    - Addition, subtraction, multiplication, division, reciprocal, square-root
    - FP $\leftrightarrow$ integer conversion
  - Operations usually takes several cycles
    - Can be pipelined

ALU
Extensions

- Floating Point Arithmetic
  - FP hardware is coprocessor 1
    - Adjunct processor that extends the ISA
  - Separate FP registers
    - 32 single-precision: $f0, $f1, … $f31
    - Paired for double-precision: $f0/$f1, $f2/$f3, …
      - Release 2 of MIPs ISA supports 32 × 64-bit FP reg's
  - FP instructions operate only on FP registers
    - Programs generally don't do integer ops on FP data, or vice versa
    - More registers with minimal code-size impact
  - FP load and store instructions
    - `lwc1`, `ldc1`, `swc1`, `sdc1`
      - e.g., `ldc1 $f8, 32($sp)`