Integer Representation
Introduction to Digital Logic
Integer Arithmetic & Adder

Representing Numbers: Review

- ***32-bit binary representation of (unsigned) number:***
  - $b_{31} \times 2^{31} + b_{30} \times 2^{30} + \cdots + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$
  - *One billion ($1{,}000{,}000{,}000_{10}$) in binary is*

$0011\ 1011\ 1001\ 1010\ 1100\ 1010\ 0000\ 0000_2$

2^{28} 2^{24} 2^{20} 2^{16} 2^{12} 2^{8} 2^{4} 2^{0}

$= 1\times2^{29} + 1\times2^{28} + 1\times2^{27} + 1\times2^{25} + 1\times2^{24} + 1\times2^{23} + 1\times2^{20} + 1\times2^{19} + 1\times2^{17} + 1\times2^{15}$

$+ 1\times2^{14} + 1\times2^{11} + 1x2^{9}$

$= 536{,}870{,}912 + 268{,}435{,}456 + 134{,}217{,}728 + 33{,}554{,}432 + 16{,}777{,}216 +$

$8{,}388{,}608 + 1{,}048{,}576 + 524{,}288 + 131{,}072 + 32{,}768 + 16{,}384 + 2{,}048 +$

$512 = 1{,}000{,}000{,}000$

What If Too Big?

- **Binary bit patterns are simply representations of numbers.**
- **Numbers really have an infinite number of digits (non-significant zeroes to the left).**
  - with almost all being zero except for a few of the rightmost digits.
  - Don't normally show leading zeros.
- **If result of add (or any other arithmetic operation) cannot be represented by these rightmost hardware bits, overflow is said to have occurred.**
- **Up to Compiler and OS what to do.**

How to Avoid Overflow? Allow It Sometimes?

- **Some languages detect overflow (Ada, Fortran), some don't (C)**
- **MIPS solution is 2 kinds of arithmetic instructions to recognize 2 choices:**
  - add (`add`), add immediate (`addi`), and subtract (`sub`) cause exceptions on overflow
  - add unsigned (`addu`), add immediate unsigned (`addiu`), and subtract unsigned (`subu`) do not cause exceptions on overflow
    - **unsigned integers commonly used for address arithmetic where overflow ignored**
    - **MIPS C compilers always produce addu, addiu, subu**

What If Overflow Detected?

- **If "exception" (or "interrupt") occurs**
  - Address of the instruction that overflowed is saved in a register
  - Computer jumps to predefined address to invoke appropriate routine for that exception
  - Like an unplanned hardware function call

- **Operating System decides what to do**
  - In some situations program continues after corrective code is executed

- **MIPS hardware support: exception program counter (EPC) contains address of overflowing instruction --- (more in Chpt. 5)**

Representing Negative Numbers

Two's Complement

- **What is result for unsigned numbers if subtract larger number from a smaller one?**
  - Would try to borrow from string of leading 0s, so result would have a string of leading 1s
  - With no obvious better alternative, pick representation that made the hardware simple:
    - leading 0s $\Rightarrow$ positive,
    - leading 1s $\Rightarrow$ negative

`000000...`***`xxx`*** $\geq$ **0**

`111111...`***`xxx`*** $<$ **0**

- **This representation is called two's complement**

Two's Complement (32-bit)

$0111 \ldots 1111\ 1111\ 1111\ 1111_{two} = 2{,}147{,}483{,}647_{ten}$

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

$0111 \ldots 1111\ 1111\ 1111\ 1101_{two} = 2{,}147{,}483{,}645_{ten}$

. . .

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

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

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

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

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

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

. . .

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

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

Indicates sign of the integer

Two's Complement Formula, Example

- **Recognizing role of sign bit, can represent positive <u>and negative</u> numbers in terms of the bit value times a power of 2:**

$$-d_{31} \times -2^{31} + d_{30} \times 2^{30} + \cdots + d_2 \times 2^2 + d_1 \times 2^1 + d_0 \times 2^0$$

- **Example (given 32-bit two's comp. number)**

$$\mathbf{1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2}$$

$$= 1 \times -2^{31} + 1 \times 2^{30} + 1 \times 2^{29} + \cdots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$$

$$= -2^{31} + 2^{30} + 2^{29} + \cdots + 2^2 + 0 + 0$$

$$= -2{,}147{,}483{,}648_{10} + 2{,}147{,}483{,}644_{10}$$

$$= -4_{10}$$

Ways to Represent Signed Numbers

(1) Sign and magnitude

– separate sign bit

0001001100101 | 1

(2) Two's (2's) Complement (*n* bit positions)

– *n*-bit pattern $d_{n-1} \ldots d_2 d_1 d_0$ means:

$$-1 \times d_{n-1} \times 2^{n-1} + \cdots + d_2 \times 2^2 + d_1 \times 2^1 + d_0 \times 2^0$$

– also, unsigned sum of *n*-bit number and its negation = 2^n

```
  0001     positive one +
+ 1111     negative one (2's comp)
  -----
 10000        = 2^4 (=zero if only 4 bits)
```

Ways to Represent Signed Numbers

(3) One's (1's) Complement

– unsigned sum of *n*-bit number and its negation = $2^n - 1$

```
 0001    positive one
+1110    negative one (1's comp)
-----
 1111    (2^4 - 1)
```

– better than sign and magnitude but has two zeros (+0=0000 and -0=1111)

– some scientific computers use 1's comp.

(4) Biased notation

– add positive bias B to signed number, store as unsigned; useful in floating point (for the exponent).

– $number = x - B$

Bit-Pattern, $b_3b_2b_1b_0$	Unsigned,	2's Comp,	1's Comp,	Biased
1111	15	-1	0	7
1110	14	-2	-1	6
1101	13	-3	-2	5
1100	12	-4	-3	4
1011	11	-5	-4	3
1010	10	-6	-5	2
1001	9	-7	-6	1
1000	8	-8	-7	0
0111	7	7	7	-1
0110	6	6	6	-2
0101	5	5	5	-3
0100	4	4	4	-4
0011	3	3	3	-5
0010	2	2	2	-6
0001	1	1	1	-7
0000	0	0	0	-8

Bias= 8 (Subtract 8)

Signed *Vs*. Unsigned Comparisons

- **Note: memory addresses naturally start at 0 and continue to the largest address – they are unsigned.**
  - That is, negative addresses make no sense.
- **C makes distinction in declaration.**
  - integer (`int`) can be positive or negative.
  - unsigned integers (`unsigned int`) only positive.
- **Thus MIPS needs two styles of comparison.**
  - Set on less than (`slt`) and set on less than immediate (`slti`) work with signed integers.
  - Set on less than unsigned (`sltu`) and set on less than immediate unsigned (`sltiu`). (Will work with addresses).

Signed *Vs.* Unsigned Comparisons

- **$s0 has**

1111 1111 1111 1111 1111 1111 1111 1100$_2$

- **$s1 has**

0011 1011 1001 1010 1000 1010 0000 0000$_2$

- **What are $t0, $t1 after:**

```
slt     $t0, $s0, $s1     # signed compare
sltu    $t1, $s0, $s1     # unsigned compare
```

- **`$t0`: -4$_{ten}$ < 1,000,000,000$_{ten}$?**
- **`$t1`: 4,294,967,292$_{ten}$ < 1,000,000,000$_{ten}$?**

- ***Key Point: Instructions decide what binary bit-patterns mean***

Two's Complement Shortcut: Negation

- **Invert every 0 to 1 and every 1 to 0, then add 1 to the result**
  - Unsigned sum of number and its inverted representation must be $111...111_2$
  - $111...111_2 = -1_{10}$
  - Let *x´* mean the *inverted representation* of *x*
  - Then $x + x' = -1 \Rightarrow x + x' + 1 = 0 \Rightarrow x' + 1 = -x$

IMPORTANT SLIDE

- **Example: -4 to +4 to -4**

- ***x* :** **$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$**

  ***x´* :** **$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0011_2$**

  +1: **$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0100_2$**

  ( )*x´*: **$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1011_2$**

  +1: **$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$**

Two’s Complement Shortcut

Using Sign extension

- **Convert number represented in *k* bits to more than *k* bits**
  - *e.g.*, 16-bit immediate field converted to 32 bits before adding to 32-bit register in addi
- **Simply replicate the most significant bit (sign bit) of smaller quantity to fill new bits**
  - 2's comp. positive number has infinite 0s to left
  - 2's comp. negative number has infinite 1s to left
  - Finite representation hides most leading bits; sign extension restores those that fit in the integer variable
  - 16-bit -4_{10} to 32-bit:

1111 1111 1111 1100$_2$

1111 1111 1111 1111 1111 1111 1111 1100$_2$

Do It Yourself

- **Convert the two's complement number**

  $1111\ 1111\ 1111\ 1111\ 1111\ 1010_{two}$

  into decimal (base ten):

Do It Yourself

- **Convert the two's complement number**

  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_2$

  into decimal (base ten):

- **Could use conversion formula (hard)**

  $1 \times -2^{31} + 1 \times 2^{30} + \ldots 1 \times 2^{1} + 1 \times 2^{0}$

- **Or, first use negation shortcut (easy)**

```
  0000 0000 0000 0000 0000 0000 0000 0101
+                                        1
  ---------------------------------------
  0000 0000 0000 0000 0000 0000 0000 0110
```

= 6 (therefore, answer: -6)

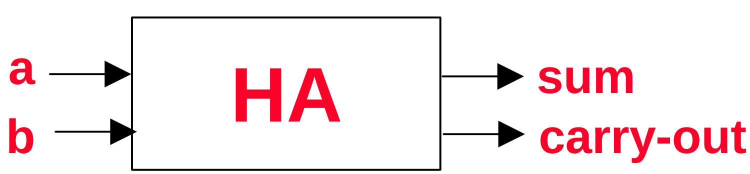
1-bit Binary Addition

- **two 1-bit values gives four cases:**

```
  0      0      1      1
+ 0    + 1    + 0    + 1
---    ---    ---    ---
  0      1      1    (1) 0
```

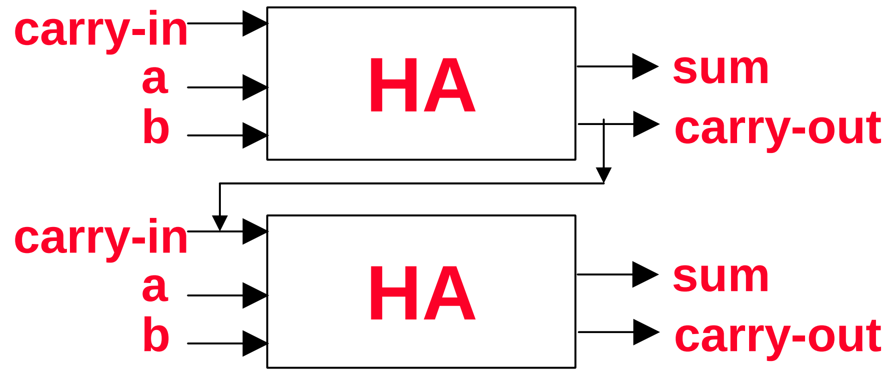
carry-out **1-bit sum**

- **digital logic?: half-adder circuit**



Multi-bit Addition (and Subtraction)

```
00 0111 = 7₁₀
00 0110 = 6₁₀
--------------
00 1101 = 13₁₀
```




```
        (0)   (0)   (1)   (1)   (0)    0    <--- carry-in
         0     0     0     1     1     1    <--- a
 +       0     0     0     1     1     0    <--- b
---------------------------------------------
   (0)   0   (0)0  (0)1  (1)1  (1)0  (0)1

                                  (carries)
```

Subtract? **Simply negate and add!**

Detecting Overflow in 2's Complement?

- **Adding 2 31-bit positive 2's complement numbers can yield a result that needs 32 bits**
  - sign bit set with value of result (1) instead of proper sign of result (0)
  - since need just 1 extra bit, only sign bit can be wrong

Op	A	B	Result
A + B	>=0	>=0	<0
A + B	<0	<0	>=0
A - B	>=0	<0	<0
A - B	<0	>=0	>=0

- **Adding operands with different signs, (subtracting with same signs) overflow cannot occur**

Overflow for Unsigned Numbers?

- **Adding 2 32-bit unsigned integers could yield a result that needs 33 bits**
  - can't detect from "sign" of result
- **Unsigned integers are commonly used for address arithmetic, where overflows are ignored**
- **Hence, MIPS has unsigned arithmetic instructions, which ignore overflow:**
  - addu, addiu, subu
  - Recall that in C, all overflows are ignored, so unsigned instructions are always used (different for Fortran, Ada)

Do It Yourself

- **Add 4-bit signed (2's complement) numbers:**

```
  1111  -1₁₀
+ 1110  -2₁₀
```

- **Did overflow occur?**

Do It Yourself

- **Add 4-bit signed (2's comp.) numbers :**

1111 -1_{10}.
\+ 1110 -2_{10}
11101.

- **Did overflow occur?**
  - overflow in 2's complement only if.
    Negative + Negative → "Positive."
    Positive + Positive → "Negative."
  - overflow = carry-out only if numbers considered to be unsigned.

- **So: addition works same way for both unsigned, signed numbers.**
- **But overflow detection is different.**

Logical Operations

- **Operations on less than full words**
  - Fields of bits or individual bits
- **Think of word as 32 bits *vs*. 2’s comp. integers or unsigned integers**
- **Need to extract bits from a word, insert bits into a word**
- **Extracting via Shift instructions**
  - C operators: << (shift left), >> (shift right)
- **Inserting via And/Or instructions**
  - C operators: & (bitwise AND), | (bitwise OR)

Shift Instructions

- **Move all the bits in a word to the left or right, filling the emptied bits with 0's**

- **Before and after shift left 8 of $s0 ($16):**

0000 0000 0000 0000 0000 0000 0000 1101 two

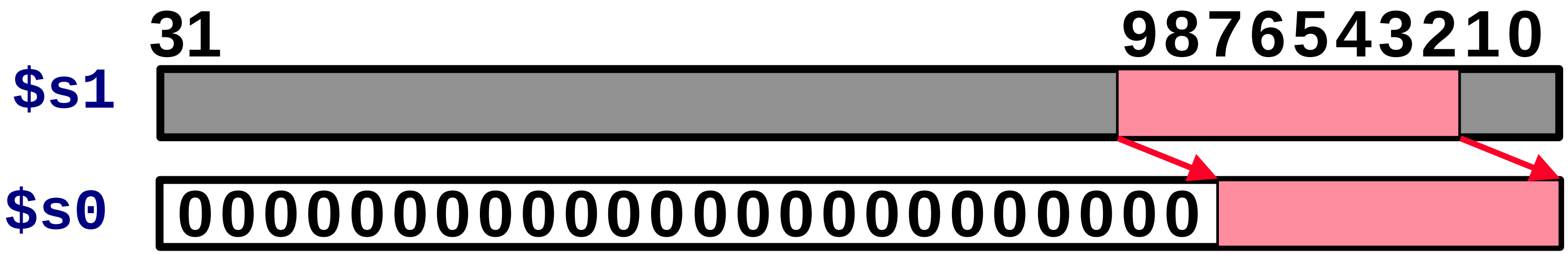
0000 0000 0000 0000 0000 1101 0000 0000 two

- **MIPS instructions**
  - shift left logical (`sll`) and shift right logical (`srl`)
  - `sll $s0, $s0, 8 # $s0 = $s0 << 8 bits`
  - R Format, using shamt (shift amount)!

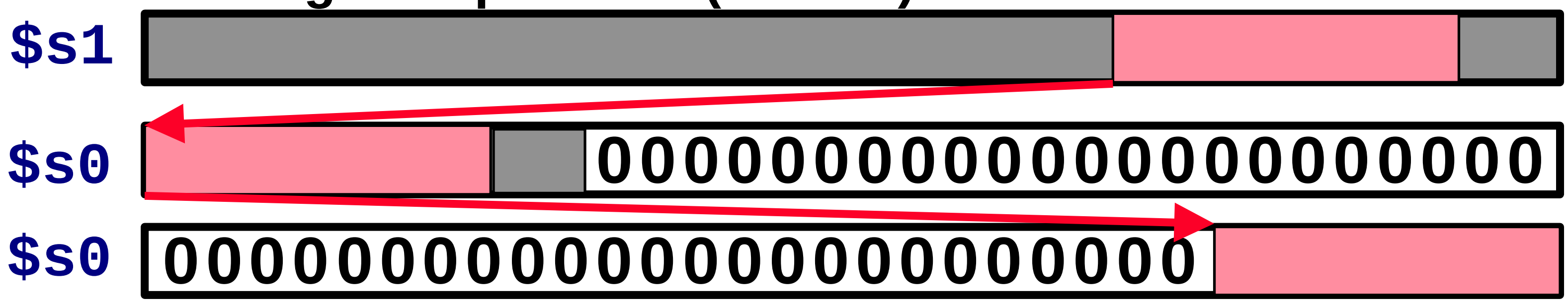
0	0	16	16	8	0
op	rs	rt	rd	shamt	funct

Extracting a Field of Bits

- **Extract bit field from bit 9 (left bit) to bit 2 (size = 8 bits) of register `$s1`, place in rightmost part of register `$s0`**



- **Shift field as far left as possible (31-bit no.) and then as far right as possible (32-size)**




```
sll  $s0, $s1, 22 # 8bits to left end (31-9)
srl  $s0, $s0, 24 # 8bits to right end(32-8)
```

And Instruction

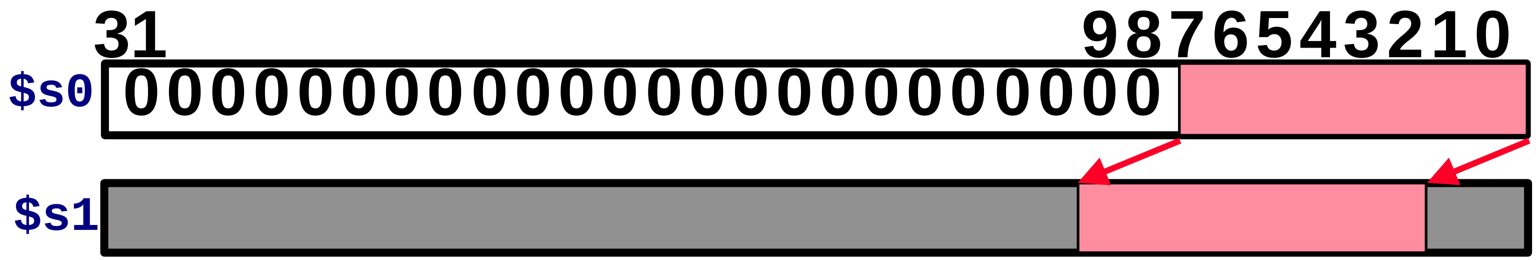
- **AND: bit-by-bit operation leaves a 1 in the result only if both bits of the operands are 1. For example, if registers $t1 and $t2**
  - 0000 0000 0000 0000 0000 1101 0000 0000_2
  - 0000 0000 0000 0000 0011 1100 0000 0000_2
- **After executing MIPS instruction**
  - `and $t0, $t1, $t2 # $t0 = $t1 & $t2`
- **Value of register $t0**
  - 0000 0000 0000 0000 0000 1100 0000 0000_2
- **AND can force 0s where 0 in the bit pattern**
  - Called a "mask" since mask "hides" bits

Or Instruction

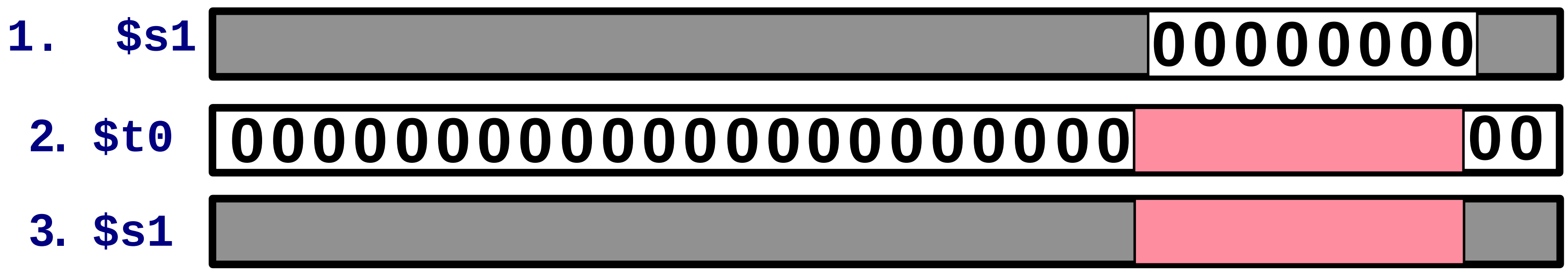
- **OR: bit-by-bit operation leaves a 1 in the result if either bit of the operands is 1. For example, if registers $t1 and $t2**
  - 0000 0000 0000 0000 0000 1101 0000 0000_2
  - 0000 0000 0000 0000 0011 1100 0000 0000_2
- **After executing MIPS instruction**
  - `or $t0, $t1, $t2 # $t0 = $t1 | $t2`
- **Value of register $t0**
  - 0000 0000 0000 0000 0011 1101 0000 0000_2
- **OR can force 1s where 1 in the bit pattern**
  - If 0s in field of 1 operand, can insert new value

Inserting a Field of Bits (Almost OK;-)

- Insert bit field into bits 9—2 (leftmost bit is 9; size = 8 bits) of register $s1 from rightmost part of register $s0 (rest is 0)



- 1. Mask out field; 2. shift left field 2; 3. OR in field




```
andi    $s1, $s1, 0xfc03 # mask out $s1[2..9] = 0
sll     $t0, $s0, 2      # field left 2 $t0[2..9]
or      $s1, $s1, $t0    # OR in field $s1 OR $t0
```

Sign Extension of Immediates

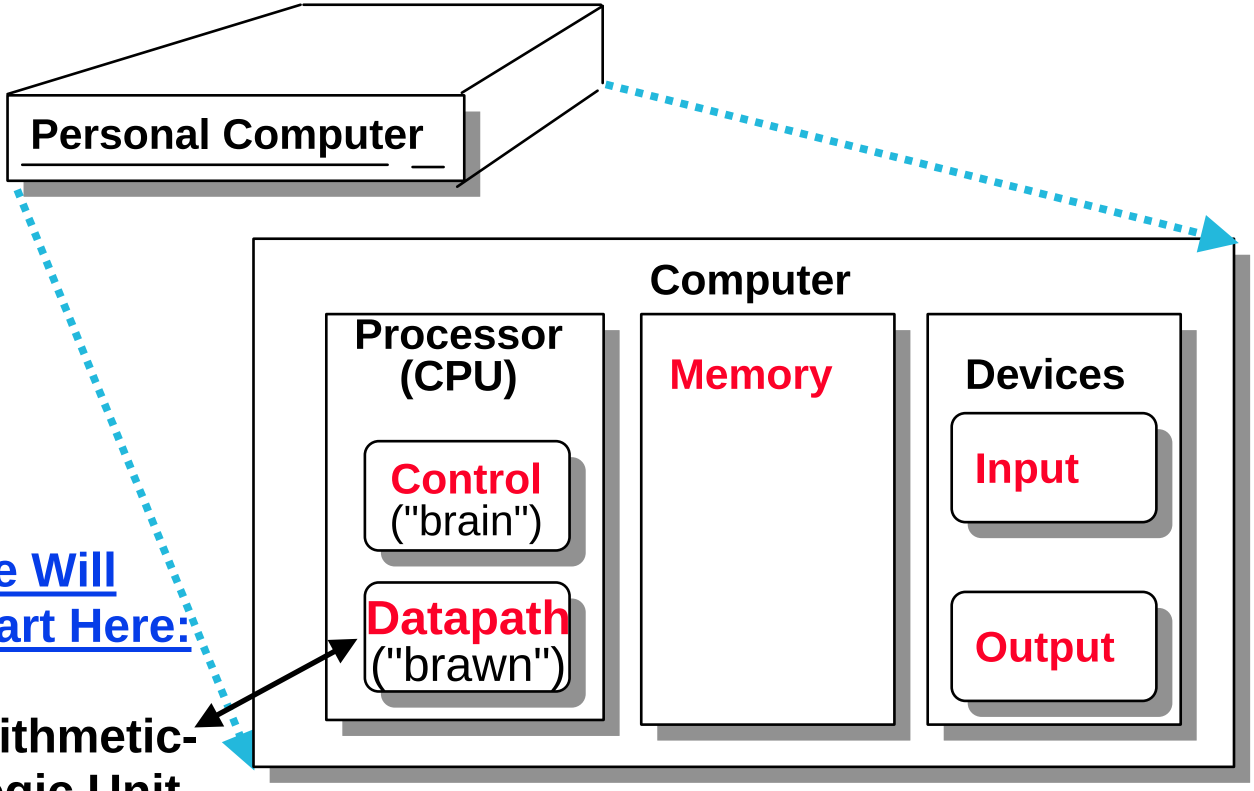
- **`addi` and `slti`: deal with signed numbers, so immediates are sign extended**

- **Branch and data transfer address fields are sign extended too**

- **`andi` and `ori` work with <u>unsigned</u> integers, so immediates padded with leading 0s**
  - `andi` won't work as a mask in upper 16 bits
  - Use register version instead

```
addiu    $t1, $zero, 0xfc03 # 32b mask in $t1
and      $s1, $s1, $t1      # mask out 9-2
sll      $t0, $s0, 2        # field left 2
or       $s1, $s1, $t0      # OR in field
```

The 5 Components of Any Computer



We Will Start Here:

Arithmetic-Logic Unit (ALU)

Overview: Digital Logic Design

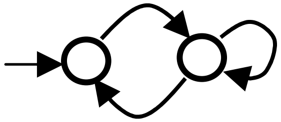
- **Topics we assume you know:**
  - Combinational and Sequential Logic Blocks
  - Boolean Algebra/Logic Equations
  - Truth Tables
  - Logic Gates

- **Appendix B gives review**
  - need B.1 - B.3 for Chapter 4
  - will need B.4 - B.6 for Chapter 5-7

Combinational, Sequential Logic

- **Two kinds of Logic Blocks (Circuits)**
  - Combinational Logic Block
    - described by a logic equation or truth table

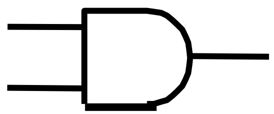
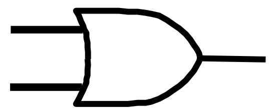
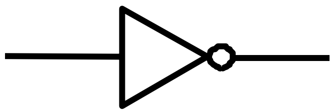
      X = AB + CD
    - no memory: *output* of block depends only on the current *inputs*; no feedback loops



  - Sequential Logic Block
    - described by a finite state machine
    - contains memory (local state); output depends on current inputs *and* stored value; permits feedback loops
  - Will use combinational logic blocks first for the datapath, then sequential logic for the control unit (Chapter 5)

Implementing Logic Blocks

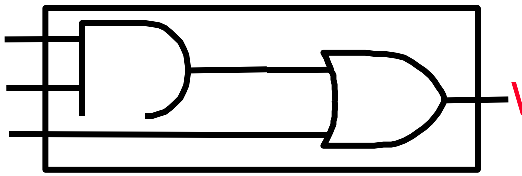
- **Logic Gates : primitives**

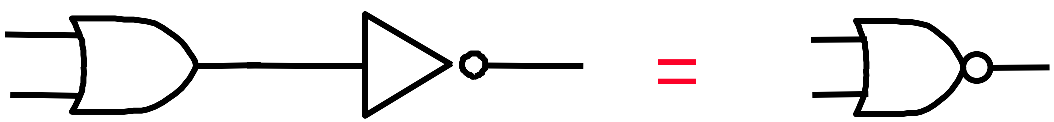
AND **OR** **NOT (inverter)**

- **Combine gates to implement more complex Boolean function:**

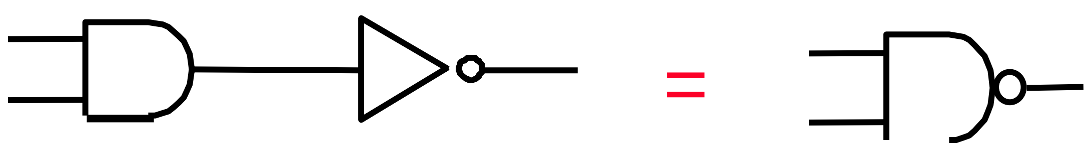
W = X + (YZ)



- **Some shorthand:**

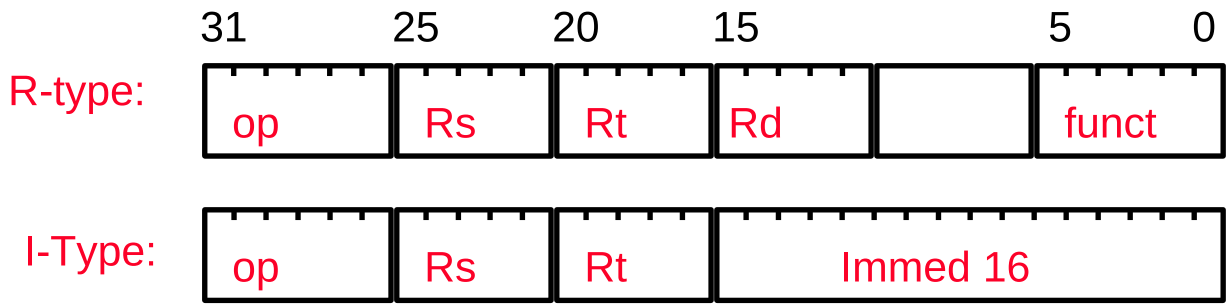


NOR



NAND

MIPS arithmetic instruction format



Type	op	funct
ADDI	10	xx
ADDIU	11	xx
SLTI	12	xx
SLTIU	13	xx
ANDI	14	xx
ORI	15	xx
XORI	16	xx
LUI	17	xx

Type	op	funct
ADD	00	40
ADDU	00	41
SUB	00	42
SUBU	00	43
AND	00	44
OR	00	45
XOR	00	46
NOR	00	47

Type	op	funct
	00	50
	00	51
SLT	00	52
SLTU	00	53

Refined Requirements

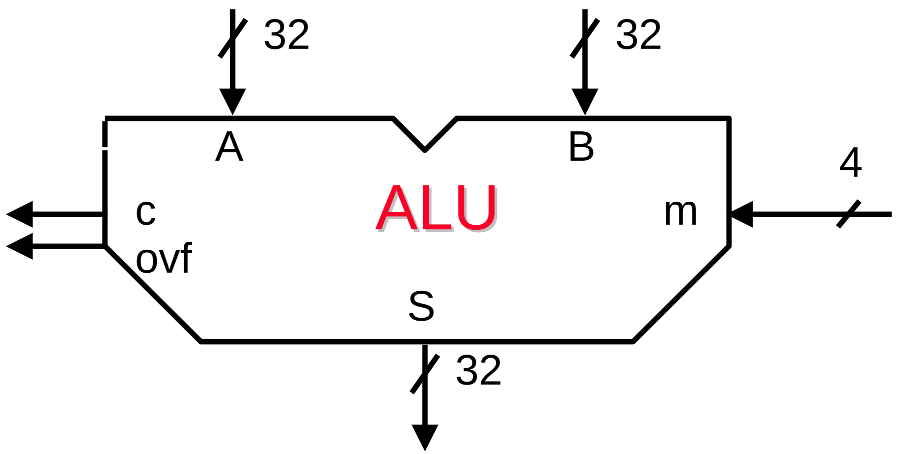
(1) Functional Specification

inputs: 2 x 32-bit operands A, B, 4-bit mode

outputs: 32-bit result S, 1-bit carry, 1 bit overflow

operations: add, addu, sub, subu, and, or, xor, nor, slt, sltU

(2) Block Diagram (powerview symbol, VHDL entity)



Gates, Truth Tables and Logic Equations

- **Digital Electronics: Circuits that operate with only two voltages of interest.**
- **"High" and "Low" voltage, corresponding to logic values. Other values occur only during transitions.**
- **Example.**
  - *"High"* $\in$ [5.0*V*, 3.5*V*]; "Low" $\in$ [0.0*V*, 1.5*V*];
- ***Associate Logic* 1 with High and Logic 0 with Low.**
- ***We will talk about logic signal values, instead of voltage levels.***
  - *Signal "asserted"* $\leftrightarrow$ 1; "de-asserted" $\leftrightarrow$ 0.

Combinational Circuits & Truth Tables

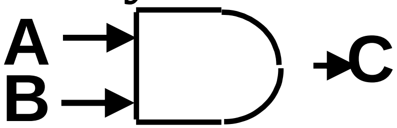
- *Combinational logic blocks have no memory and can be fully described by truth tables.*
- *Each function with n inputs → 2^n entries.*
- *Let Z = G (A, B, C).*
- *A Truth Table describes the behaviour of G.*

A	B	C	Z	D	E	F.
0	0	0	z_{000}	0	0	0.
0	0	1	z_{001}	1	0	0.
0	1	0	z_{010}	1	0	0.
0	1	1	z_{011}	1	1	0.
1	0	0	z_{100}	1	0	0.
1	0	1	z_{101}	1	1	0.
1	1	0	z_{110}	1	1	0.
1	1	1	z_{111}	1	0	1.

Hardware Building Blocks

AND Gate

Symbol

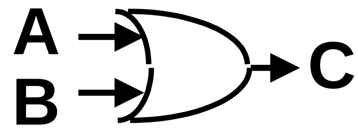


Definition

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

OR Gate

Symbol

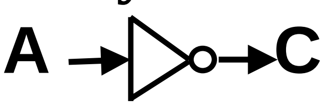


Definition

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

Inverter

Symbol

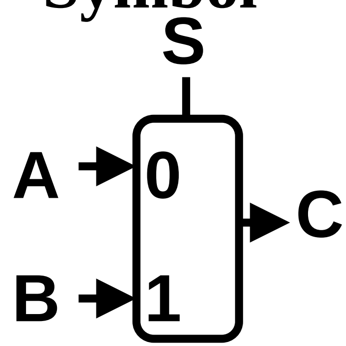


Definition

A	C
0	1
1	0

Multiplexor

Symbol



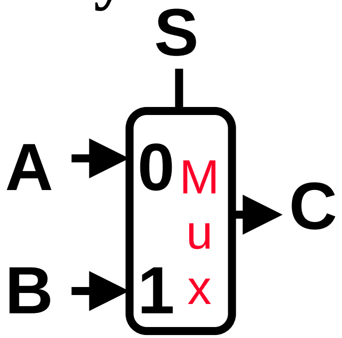
Definition

S	C
0	A
1	B

Multiplexors

- **AND, OR, Inverter (NOT) are the logic *primitives* (smallest logic elements)**
- **Multiplexors, *e.g.*, Selector, Mux, can be *constructed* from primitives:**

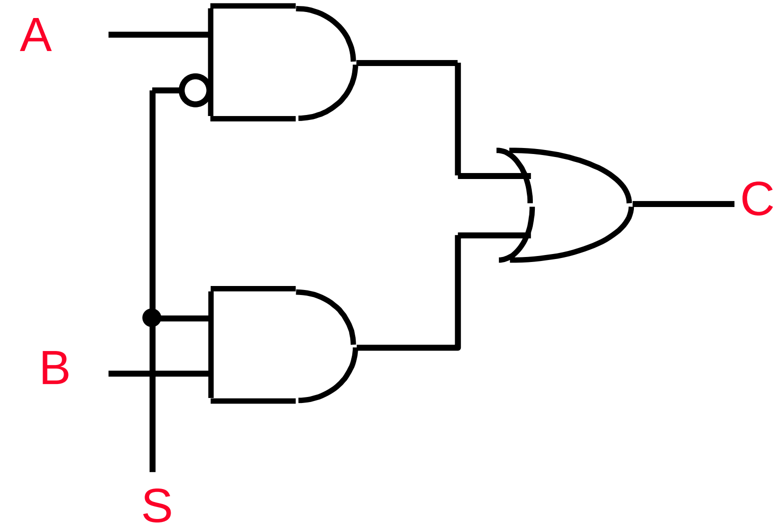
Symbol



Definition

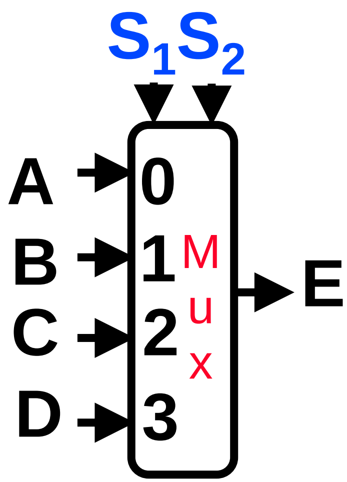
S	C
0	A
1	B

$C = SB + S'A$



Multiplexors

- **Larger muxes: need multiple "select" inputs: interpret as binary number**



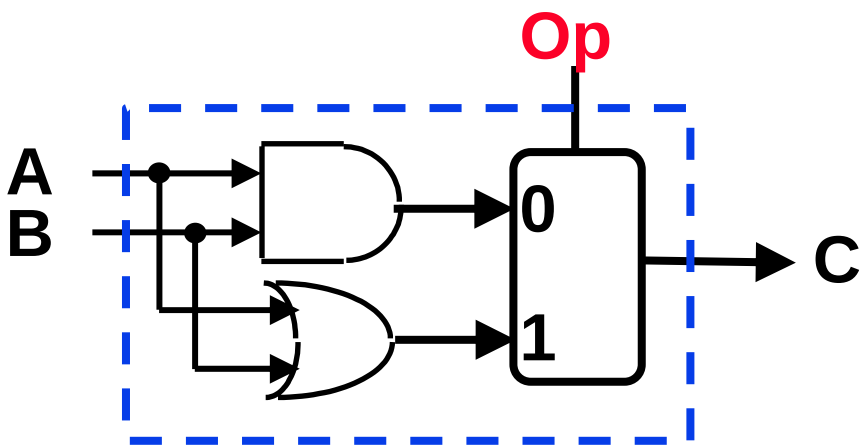
S_1	S_2	E
0	0	A
0	1	B
1	0	C
1	1	D

Appendix B.3 for more details

  - Can implement directly with gates, or
  - use decoder (see B.3) to enable a single input, or
  - combine several 2-input muxes

Arithmetic Logic Unit (ALU)

- **MIPS ALU is 32 bits wide**
- **Start with 1-bit ALU, then connect 32 1-bit ALUs to form a 32-bit ALU in a "bit slice" manner**
- **Since hardware building blocks include an AND gate and an OR gate, and since AND and OR are two of the operations of the ALU, start here:**



Definition

Op	C
0	A and B
1	A or B