Lecture 5B

Machine Number Representation

- Bits are just bits (have no inherent meaning)
  - conventions define the relationships between bits and numbers
- Binary numbers (base 2) - integers

  0000 → 0001 → 0010 → 0011 → 0100 → 0101 → . . .
  - in decimal from 0 to 2^n-1 for n bits
- Of course, it gets more complicated
  - storage locations (e.g., register file words) are finite, so have to worry about overflow (i.e., when the number is too big to fit into 32 bits)
  - have to be able to represent negative numbers, e.g., how do we specify -8 in

    ```
    addi   $sp, $sp, -8      #$sp = $sp - 8
    ```
  - in real systems have to provide for more that just integers, e.g., fractions and real numbers (and floating point) and alphanumeric (characters)

Possible Representations

Sign Mag.	Two's Comp.	One's Comp.
	1000 = -8	
1111 = -7	1001= -7	1000 = -7
1110 = -6	1010 = -6	1001 = -6
1101 = -5	1011 = -5	1010 = -5
1100 = -4	1100 = -4	1011 = -4
1011 = -3	1101 = -3	1100 = -3
1010 = -2	1110 = -2	1101 = -2
1001 = -1	1111 = -1	1110 = -1
1000 = -0		1111 = -0
0000 = +0	0000 = 0	0000 = +0
0001 = +1	0001 = +1	0001 = +1
0010 = +2	0010 = +2	0010 = +2
0011 = +3	0011 = +3	0011 = +3
0100 = +4	0100 = +4	0100 = +4
0101 = +5	0101 = +5	0101 = +5
0110 = +6	0110 = +6	0110 = +6
0111 = +7	0111 = +7	0111 = +7

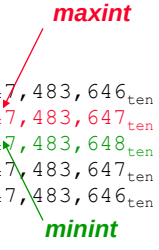
MIPS Representations

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

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +\ 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = +\ 2_{ten}$
...

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +\ 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -\ 2{,}147{,}483{,}646_{ten}$
...

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -\ 3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1_{ten}$



- What if the bit string represented addresses?
  - need operations that also deal with only positive (unsigned) integers

Two's Complement Operations

- Negating a two's complement number – complement all the bits and then add a 1
  - remember: "negate" and "invert" are quite different!
- Converting n-bit numbers into numbers with more than n bits:
  - MIPS 16-bit immediate gets converted to 32 bits for arithmetic
  - sign extend - copy the most significant bit (the sign bit) into the other bits

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

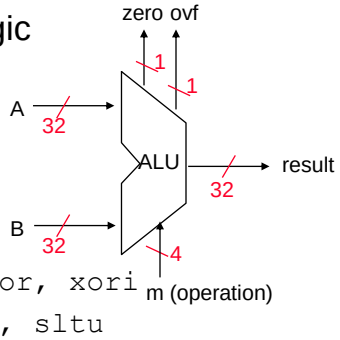
  - sign extension versus zero extend (`lb` vs. `lbu`)

Design the MIPS Arithmetic Logic Unit (ALU)

- Must support the Arithmetic/Logic operations of the ISA

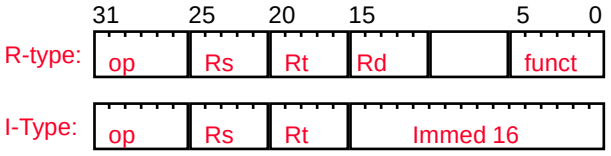
```
add, addi, addiu, addu
sub, subu
mult, multu, div, divu

and, andi, nor, or, ori, xor, xori
beq, bne, slt, slti, sltiu, sltu
```



- With special handling for
  - sign extend – `addi, addiu, slti, sltiu`
  - zero extend – `andi, ori, xori`
  - overflow detection – `add, addi, sub`

MIPS Arithmetic and Logic Instructions



Type	op	funct
ADDI	001000	xx
ADDIU	001001	xx
SLTI	001010	xx
SLTIU	001011	xx
ANDI	001100	xx
ORI	001101	xx
XORI	001110	xx
LUI	001111	xx

Type	op	funct
ADD	000000	100000
ADDU	000000	100001
SUB	000000	100010
SUBU	000000	100011
AND	000000	100100
OR	000000	100101
XOR	000000	100110
NOR	000000	100111

Type	op	funct
	000000	101000
	000000	101001
SLT	000000	101010
SLTU	000000	101011
	000000	101100

Design Trick: Divide & Conquer

- Break the problem into simpler problems, solve them and glue together the solution
- Example: assume the immediates have been taken care of before the ALU
  - now down to 10 operations
  - can encode in 4 bits

0	add
1	addu
2	sub
3	subu
4	and
5	or
6	xor
7	nor
a	slt
b	sltu

Addition & Subtraction

- Just like in grade school (carry/borrow 1s)

```
  0111          0111          0110
+ 0110        - 0110        - 0101
  1101          0001          0001
```

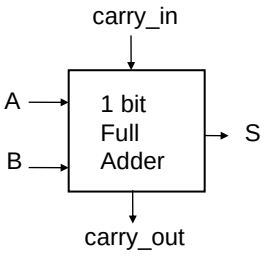
- Two's complement operations are easy
  - do subtraction by negating and then adding

```
  0111    →     0111
- 0110    →   + 1010
  0001          1 0001
```

- Overflow (result too large for finite computer word)
  - e.g., adding two n-bit numbers does not yield an n-bit number

```
  0111
+ 0001
  1000
```

Building a 1-bit Binary Adder

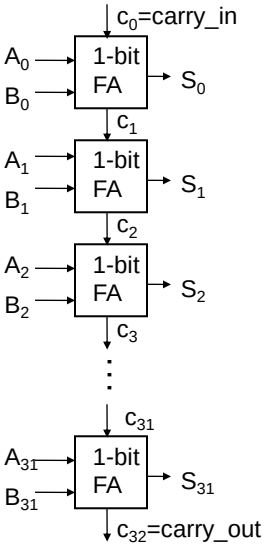


A	B	carry_in	carry_out	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

S = A xor B xor carry_in
carry_out = A&B | A&carry_in | B&carry_in

- How can we use it to build a 32-bit adder?
- How can we modify it easily to build an adder/subtractor?

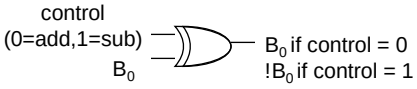
Building 32-bit Adder



- Just connect the carry-out of the least significant bit FA to the carry-in of the next least significant bit and connect . . .
- Ripple Carry Adder (RCA)
  - advantage: simple logic, so small (low cost)
  - disadvantage: slow and lots of glitching (so lots of energy consumption)

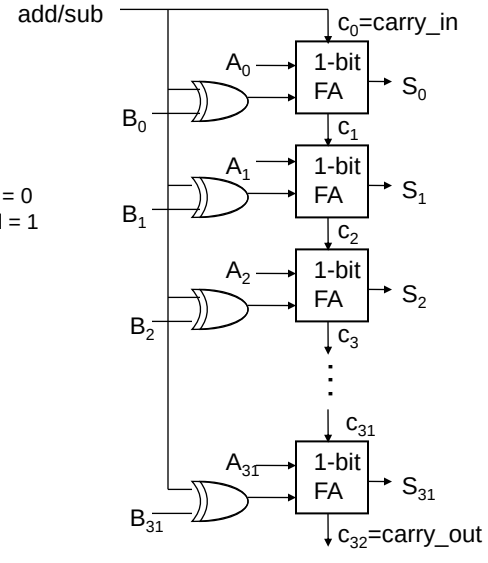
A 32-bit Ripple Carry Adder/Subtractor

- Remember 2's complement is just
  - complement all the bits



  - add a 1 in the least significant bit

```
A    0111   →    0111
B  - 0110   → +  1001
     0001          1
                1 0001
```



Overflow Detection and Effects

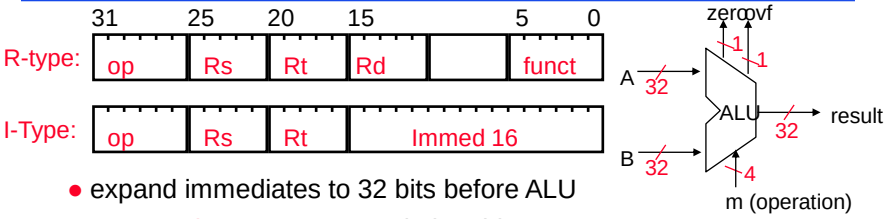
- Overflow: the result is too large to represent in the number of bits allocated
- When adding operands with different signs, overflow cannot occur! Overflow occurs when
  - adding two positives yields a negative
  - or, adding two negatives gives a positive
  - or, subtract a negative from a positive gives a negative
  - or, subtract a positive from a negative gives a positive

New MIPS Instructions

Category	Instr	Op Code	Example	Meaning
Arithmetic (R & I format)	add unsigned	0 and 21	addu $s1, $s2, $s3	$s1 = $s2 + $s3
	sub unsigned	0 and 23	subu $s1, $s2, $s3	$s1 = $s2 - $s3
	add imm.unsigned	9	addiu $s1, $s2, 6	$s1 = $s2 + 6
Data Transfer	ld byte unsigned	24	lbu $s1, 25($s2)	$s1 = Mem($s2+25)
	ld half unsigned	25	lhu $s1, 25($s2)	$s1 = Mem($s2+25)
Cond. Branch (I & R format)	set on less than unsigned	0 and 2b	sltu $s1, $s2, $s3	if ($s2<$s3) $s1=1 else $s1=0
	set on less than imm unsigned	b	sltiu $s1, $s2, 6	if ($s2<6) $s1=1 else $s1=0

- Sign extend – `addiu, addiu, slti, sltiu`
- Zero extend – `andi, ori, xori`
- Overflow detected – `add, addi, sub`

Review: MIPS Arithmetic Instructions



- expand immediates to 32 bits before ALU
- 10 operations so can encode in 4 bits

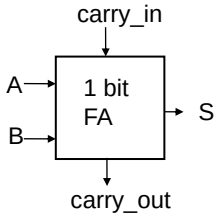
Type	op	funct
ADD	00	100000
ADDU	00	100001
SUB	00	100010
SUBU	00	100011
AND	00	100100
OR	00	100101
XOR	00	100110
NOR	00	100111

Type	op	funct
	00	101000
	00	101001
SLT	00	101010
SLTU	00	101011
	00	101100

0	add
1	addu
2	sub
3	subu
4	and
5	or
6	xor
7	nor
a	slt
b	sltu

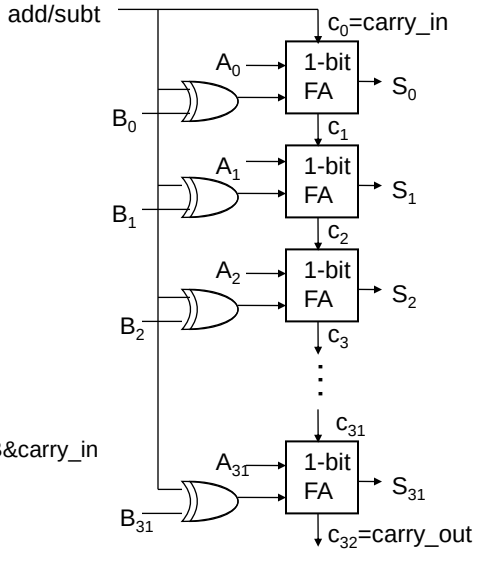
Review: A 32-bit Adder/Subtractor

- Built out of 32 full adders (FAs)



S = A xor B xor carry_in

carry_out = A&B | A&carry_in | B&carry_in
(majority function)

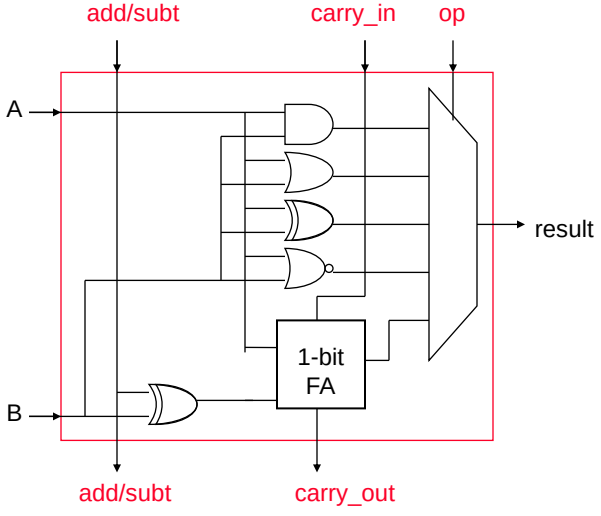


- Small but slow!

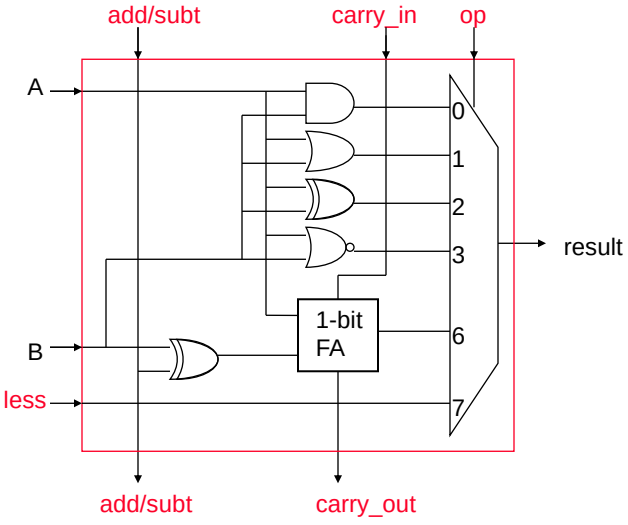
Tailoring the ALU to the MIPS ISA

- Also need to support the logic operations (`and, nor, or, xor`)
  - Bit wise operations (no carry operation involved)
  - Need a logic gate for each function and a mux to choose the output
- Also need to support the set-on-less-than instruction (`slt`)
  - Uses subtraction to determine if (a – b) < 0 (implies a < b)
- Also need to support test for equality (`bne, beq`)
  - Again use subtraction: (a - b) = 0 implies a = b
- Also need to add overflow detection hardware
  - overflow detection enabled only for `add, addi, sub`
- Immediates are sign extended outside the ALU with wiring (i.e., no logic needed)

A Simple ALU Cell with Logic Op Support

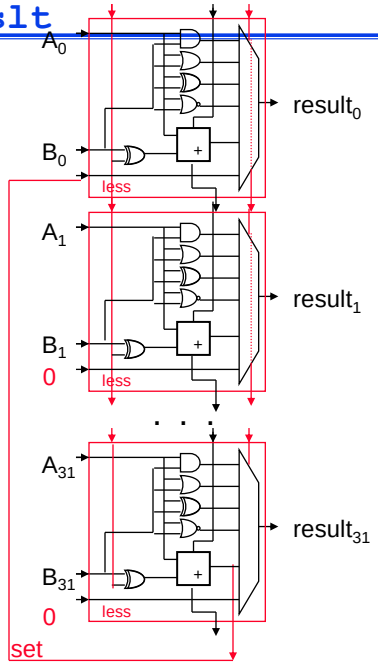


Modifying the ALU Cell for `slt`

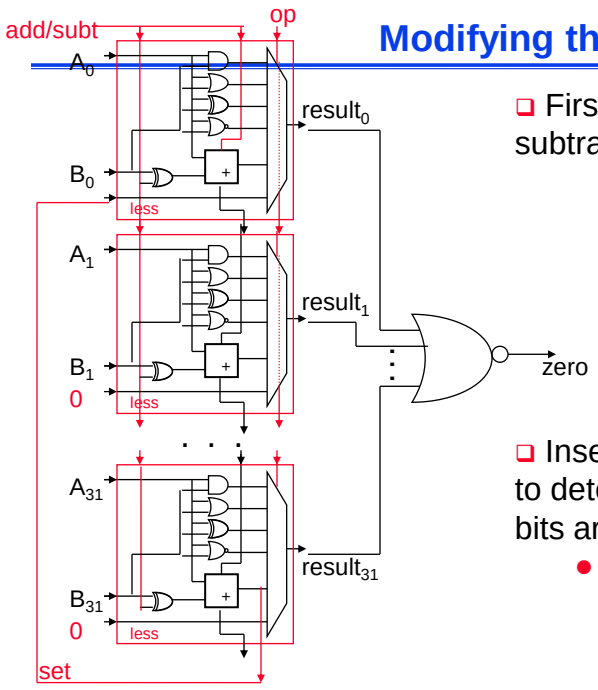


Modifying the ALU for `slt`

- First perform a subtraction
- Make the result 1 if the subtraction yields a negative result
- Make the result 0 if the subtraction yields a positive result
  - tie the most significant sum bit (sign bit) to the low order less input



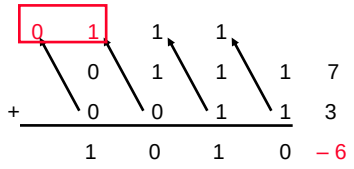
Modifying the ALU for Zero

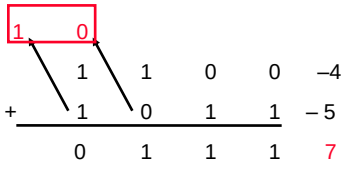


- First perform subtraction
- Insert additional logic to detect when all result bits are zero
  - Note zero is a 1 when result is all zeros

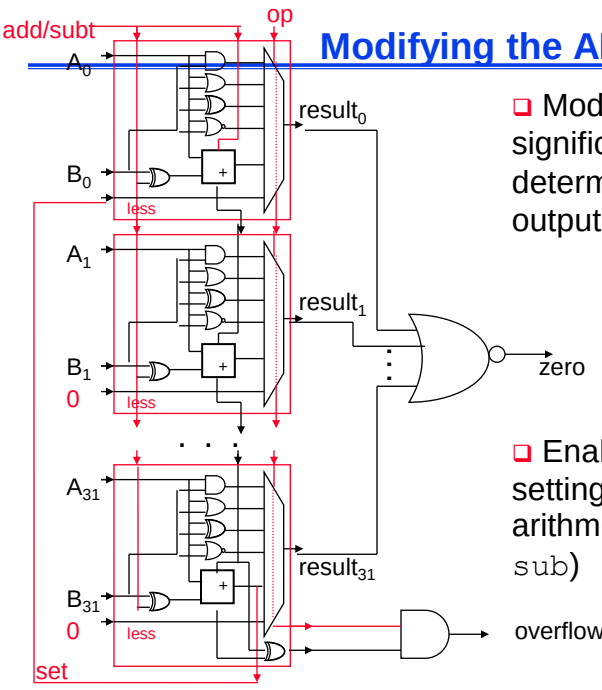
Overflow Detection

- Overflow occurs when the result is too large to represent in the number of bits allocated
  - adding two positives yields a negative
  - or, adding two negatives gives a positive
  - or, subtract a negative from a positive gives a negative
  - or, subtract a positive from a negative gives a positive
- On your own: Prove you can detect overflow by:
  - Carry into MSB xor Carry out of MSB





Modifying the ALU for Overflow



- Modify the most significant cell to determine overflow output setting
- Enable overflow bit setting for signed arithmetic (`add, addi, sub`)

Multiplication

- More complicated than addition
  - Can be accomplished via shifting and adding

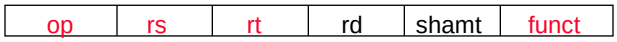
```
   0010   (multiplicand)
 x 1011   (multiplier)
  0010
  0010    (partial product
 0000       array)
0010
0001 0110  (product)
```

- Double precision product produced
- More time and more area to compute

MIPS Multiply Instruction

- Multiply produces a double precision product

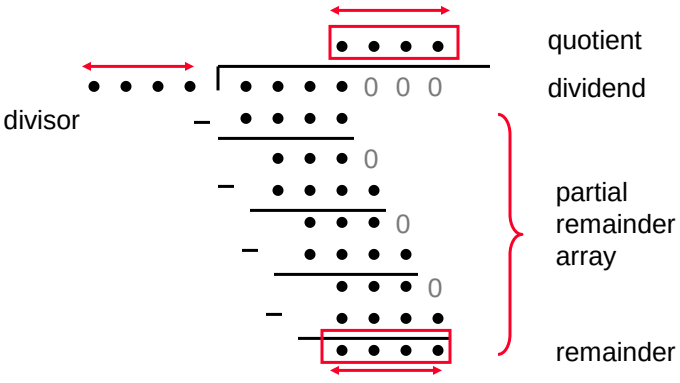
```
mult    $s0, $s1     # hi||lo = $s0 * $s1
```



  - Low-order word of the product is left in processor register `lo` and the high-order word is left in register `hi`
  - Instructions `mfhi rd` and `mflo rd` are provided to move the product to (user accessible) registers in the register file

- Multiplies are done by fast, dedicated hardware and are much more complex (and slower) than adders
- Hardware dividers are even *more* complex and even slower

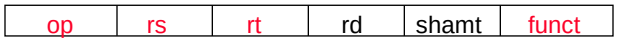
Division



MIPS Divide Instruction

- Divide generates the reminder in `hi` and the quotient in `lo`

```
div   $s0, $s1        # lo = $s0 / $s1
                      # hi = $s0 mod $s1
```

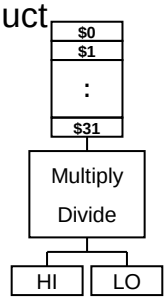


  - Instructions `mflo rd` and `mfhi rd` are provided to move the quotient and reminder to (user accessible) registers in the register file

- As with multiply, divide ignores overflow so software must determine if the quotient is too large. Software must also check the divisor to avoid division by 0.

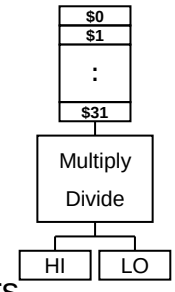
Integer Multiplication in MIPS - revisited

- Multiply instructions
  - **`mult  Rs, Rt`** **Signed multiplication**
  - **`multu Rs, Rt`** **Unsigned multiplication**
- 32-bit multiplication produces a 64-bit Product
- Separate pair of 32-bit registers
  - **HI = high-order 32-bit of product**
  - **LO = low-order 32-bit of product**
- MIPS also has a special **`mul`** instruction
  - **`mul  Rd, Rs, Rt`** **`Rd = Rs × Rt`**
  - Copy **LO** into destination register **Rd**
  - Useful when the product is small (32 bits) and **HI** is not



Integer Division in MIPS

- Divide instructions
  - `div Rs, Rt` **Signed division**
  - `divu Rs, Rt` **Unsigned division**
- Division produces quotient and remainder
- Separate pair of 32-bit registers
  - **HI = 32-bit remainder**
  - **LO = 32-bit quotient**
  - If divisor is 0 then result is **unpredictable**
- Moving data from **HI, LO** to MIPS registers
  - `mfhi Rd` (**Rd = HI**)
  - `mflo Rd` (**Rd = LO**)



Integer Multiply and Divide Instructions

Instruction	Meaning	Format					
`mult Rs, Rt`	`HI, LO = Rs ×s Rt`	`Op = 0`	`Rs`	`Rt`	`0`	`0`	`0x18`
`multu Rs, Rt`	`HI, LO = Rs ×u Rt`	`Op = 0`	`Rs`	`Rt`	`0`	`0`	`0x19`
`mul Rd, Rs, Rt`	`Rd = Rs ×s Rt`	`0x1c`	`Rs`	`Rt`	`Rd`	`0`	`2`
`div Rs, Rt`	`HI, LO = Rs /s Rt`	`Op = 0`	`Rs`	`Rt`	`0`	`0`	`0x1a`
`divu Rs, Rt`	`HI, LO = Rs /u Rt`	`Op = 0`	`Rs`	`Rt`	`0`	`0`	`0x1b`
`mfhi Rd`	`Rd = HI`	`Op = 0`	`0`	`0`	`Rd`	`0`	`0x10`
`mflo Rd`	`Rd = LO`	`Op = 0`	`0`	`0`	`Rd`	`0`	`0x12`
`mthi Rs`	`HI = Rs`	`Op = 0`	`Rs`	`0`	`0`	`0`	`0x11`
`mtlo Rs`	`LO = Rs`	`Op = 0`	`Rs`	`0`	`0`	`0`	`0x13`

$\times_s$ = Signed multiplication, $\times_u$ = Unsigned multiplication

$/_s$ = Signed division, $/_u$ = Unsigned division

Shift Operations

- Shifts move all the bits in a word left or right

```
sll  $t2, $s0, 8 #$t2 = $s0 << 8 bits
srl  $t2, $s0, 8 #$t2 = $s0 >> 8 bits
sra  $t2, $s0, 8 #$t2 = $s0 >> 8 bits
```

- Notice that a 5-bit shamt field is enough to shift a 32-bit value $2^5 - 1$ or 31 bit positions
- Logical shifts fill with zeros, arithmetic left shifts fill with the sign bit
- The shift operation is implemented by hardware separate from the ALU

MIPS Conditional Branch Instructions

- MIPS **compare and branch** instructions:

  `beq Rs, Rt, label` if (**`Rs == Rt`**) branch to **`label`**

  `bne Rs, Rt, label` if (**`Rs != Rt`**) branch to **`label`**

- MIPS **compare to zero & branch** instructions:

  Compare to zero is used frequently and implemented efficiently

  `bltz Rs, label` if (**`Rs < 0`**) branch to **`label`**

  `bgtz Rs, label` if (**`Rs > 0`**) branch to **`label`**

  `blez Rs, label` if (**`Rs <= 0`**) branch to **`label`**

  `bgez Rs, label` if (**`Rs >= 0`**) branch to **`label`**

- **`beqz`** and **`bnez`** are defined as pseudo-instructions.

Branch Instruction Format

❖ Branch Instructions are of the I-type Format:

Instruction	I-Type Format			
`beq Rs, Rt, label`	`Op = 4`	`Rs`	`Rt`	`16-bit Offset`
`bne Rs, Rt, label`	`Op = 5`	`Rs`	`Rt`	`16-bit Offset`
`blez Rs, label`	`Op = 6`	`Rs`	`0`	`16-bit Offset`
`bgtz Rs, label`	`Op = 7`	`Rs`	`0`	`16-bit Offset`
`bltz Rs, label`	`Op = 1`	`Rs`	`0`	`16-bit Offset`
`bgez Rs, label`	`Op = 1`	`Rs`	`1`	`16-bit Offset`

- The branch instructions modify the **PC register** only
- **PC-Relative addressing**:

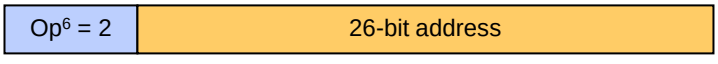
If (branch is taken) **PC = PC + 4 + 4×offset** else **PC =PC+4**

Unconditional Jump Instruction

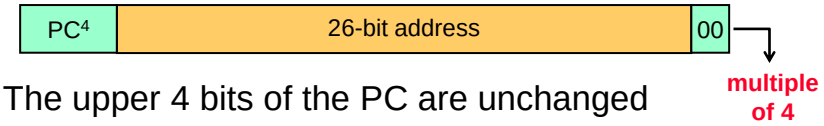
- Unconditional Jump instruction has the following syntax:

```
j   label   # jump to label
  . . .
label:
```

- The jump instruction is **always taken**
- The Jump instruction is of the J-type format:



- The jump instruction modifies the program counter PC:



- The upper 4 bits of the PC are unchanged

Translating an IF Statement

- Consider the following IF statement:

  `if (a == b) c = d + e; else c = d – e;`

  Given that **a, b, c, d, e** are in **`$t0 … $t4`** respectively

- How to translate the above IF statement?

```
        bne   $t0, $t1, else
        addu  $t2, $t3, $t4
        j     next
else:   subu  $t2, $t3, $t4
next:   . . .
```

Logical AND Expression

- Programming languages use **short-circuit evaluation**
- If first condition is **false**, second condition is **skipped**

```
if (($t1 > 0) && ($t2 < 0)) {$t3++;}
```

```
# One Possible Translation ...
   bgtz   $t1, L1        # first condition
   j      next           # skip if false
L1:bltz   $t2, L2        # second condition
   j      next           # skip if false
L2:addiu  $t3, $t3, 1    # both are true
next:
```

Better Translation of Logical AND

```
if (($t1 > 0) && ($t2 < 0)) {$t3++;}
```

Allow the program to **fall through** to second condition

!($t1 > 0) is equivalent to **($t1 <= 0)**

!($t2 < 0) is equivalent to **($t2 >= 0)**

Number of instructions is reduced from **5** to **3**

```
# Better Translation ...
   blez   $t1, next        # 1st condition false?
   bgez   $t2, next        # 2nd condition false?
   addiu  $t3, $t3, 1      # both are true
next:
```

Logical OR Expression

❖ **Short-circuit evaluation** for logical OR

❖ If first condition is **true**, second condition is **skipped**

```
if (($t1 > 0) || ($t2 < 0)) {$t3++;}
```

❖ Use **fall-through** to keep the code as short as possible

```
    bgtz   $t1, L1       # 1st condition true?
    bgez   $t2, next     # 2nd condition false?
L1: addiu $t3, $t3, 1    # increment $t3
next:
```

Compare Instructions

- MIPS also provides **set less than** instructions

```
slt   Rd, Rs, Rt    if (Rs < Rt) Rd = 1 else Rd = 0
sltu  Rd, Rs, Rt    unsigned <
slti  Rt, Rs, imm   if (Rs < imm) Rt = 1 else Rt = 0
sltiu Rt, Rs, imm   unsigned <
```

- **Signed / Unsigned** comparisons compute different results

  Given that: `$t0 = 1` and `$t1 = -1 = 0xffffffff`

```
slt   $t2, $t0, $t1     computes  $t2 = 0
sltu  $t2, $t0, $t1     computes  $t2 = 1
```

Compare Instruction Formats

Instruction	Meaning	Format					
`slt Rd, Rs, Rt`	`Rd=(Rs <s Rt)?1:0`	`Op=0`	`Rs`	`Rt`	`Rd`	`0`	`0x2a`
`sltu Rd, Rs, Rt`	`Rd=(Rs <u Rt)?1:0`	`Op=0`	`Rs`	`Rt`	`Rd`	`0`	`0x2b`
`slti Rt, Rs, im`	`Rt=(Rs <s im)?1:0`	`0xa`	`Rs`	`Rt`	`16-bit immediate`		
`sltiu Rt, Rs, im`	`Rt=(Rs <u im)?1:0`	`0xb`	`Rs`	`Rt`	`16-bit immediate`		

- The other comparisons are defined as pseudo-instructions:

`seq, sne, sgt, sgtu, sle, sleu, sge, sgeu`

Pseudo-Instruction	Equivalent MIPS Instructions
`sgt $t2, $t0, $t1`	`slt $t2, $t1, $t0`
`seq $t2, $t0, $t1`	`subu $t2, $t0, $t1` `sltiu $t2, $t2, 1`

Pseudo-Branch Instructions

- MIPS hardware does NOT provide the following instructions:

`blt, bltu`	branch if less than	(signed / unsigned)
`ble, bleu`	branch if less or equal	(signed / unsigned)
`bgt, bgtu`	branch if greater than	(signed / unsigned)
`bge, bgeu`	branch if greater or equal	(signed / unsigned)

-

Pseudo-Instruction	Equivalent MIPS Instructions
`blt $t0, $t1, label`	`slt $at, $t0, $t1` `bne $at, $zero, label`
`ble $t0, $t1, label`	`slt $at, $t1, $t0` `beq $at, $zero, label`

`$at` (`$1`) is the **assembler temporary register**

Using Pseudo-Branch Instructions

- Translate the IF statement to assembly language
- **`$t1`** and **`$t2`** values are **unsigned**

```
if($t1 <= $t2) {
  $t3 = $t4;
}
```

```
   bgtu  $t1, $t2, L1
   move  $t3, $t4
L1:
```

- **`$t3`**, **`$t4`**, and **`$t5`** values are **signed**

```
if (($t3 <= $t4) &&
    ($t4 >= $t5)) {
  $t3 = $t4 + $t5;
}
```

```
   bgt   $t3, $t4, L1
   blt   $t4, $t5, L1
   addu  $t3, $t4, $t5
L1:
```

Conditional Move Instructions

Instruction	Meaning	R-Type Format					
`movz Rd, Rs, Rt`	`if (Rt==0) Rd=Rs`	`Op=0`	`Rs`	`Rt`	`Rd`	`0`	`0xa`
`movn Rd, Rs, Rt`	`if (Rt!=0) Rd=Rs`	`Op=0`	`Rs`	`Rt`	`Rd`	`0`	`0xb`

```
if ($t0 == 0) {$t1=$t2+$t3;} else {$t1=$t2-$t3;}
```

```
    bne   $t0, $0, L1
    addu  $t1, $t2, $t3
    j     L2
L1: subu  $t1, $t2, $t3
L2: . . .
```

```
addu  $t1, $t2, $t3
subu  $t4, $t2, $t3
movn  $t1, $t4, $t0
. . .
```

- Conditional move can eliminate branch & jump instructions

Pseudo-Instructions

- Introduced by the assembler as if they were real instructions
- Facilitate assembly language programming

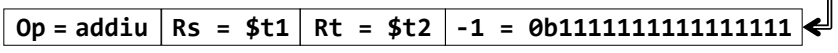
Pseudo-Instruction	Equivalent MIPS Instruction
`move $t1, $t2`	`addu $t1, $t2, $zero`
`not $t1, $t2`	`nor $t1, $t2, $zero`
`neg $t1, $t2`	`sub $t1, $zero, $t2`
`li $t1, -5`	`addiu $t1, $zero, -5`
`li $t1, 0xabcd1234`	`lui $t1, 0xabcd` `ori $t1, $t1, 0x1234`

The MARS tool has a long list of pseudo-instructions

Examples of I-Type ALU Instructions

- Given that registers **$t0, $t1, $t2** are used for **A, B, C**

Expression	Equivalent MIPS Instruction
A = B + 5;	addiu $t0, $t1, 5
C = B – 1;	addiu $t2, $t1, -1
A = B & 0xf;	andi $t0, $t1, 0xf
C = B | 0xf;	ori $t2, $t1, 0xf
C = 5;	addiu $t2, $zero, 5
A = B;	addiu $t0, $t1, 0



No need for **subiu**, because **addiu** has signed immediate

Register **$zero** has always the value **0**