Lecture 5: Arithmetic and Logical instructions

There are 10 types of people in the world: those who understand binary, and those who don't.

Topics

- Integer numbers
- MIPS arithmetic and logical instructions
- Bits masking example (lab 7)
- Some textbook references
  - PH Ed3: 3.1, 3.2, 3.3, 3.5
  - PH Ed4: 3.1, 3.2, 3.3, 3.5
  - PH Ed5: 3.1, 3.2, 3.3, 3.5
  - PH Ed6: 3.1, 3.2, 3.3, 3.5

3-bit Binary pattern	Decimal Values		
	Sign Magnitude	1's Complement •if MSB=0, positive value •if MSB=1, invert bits, assume negative	2's Complement •if MSB=0, positive value •if MSB=1, invert bits, add 1, assume negative
000	+0	+0	+0
001	+1	+1	+1
010	+2	+2	+2
011	+3	+3	+3
100	-0	-3	-4
101	-1	-2	-3
110	-2	-1	-2
111	-3	-0	-1

Interpreting bit patterns

- A 32-bit word has no inherent meaning; it can represent various things:
  - ?
  - ?
  - ?
- Bits in a word always are numbered from right to left
  - Least Significant Bit (LSB) - **bit 0** (rightmost)
  - Most Significant Bit (MSB) - **bit 31** (leftmost)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 **0**

Unsigned binary number

- Representation
  - straightforward for natural numbers
- Example
  - 10110 has a decimal value
  - $(1 \times 2^4)+(0x2^3)+(1x2^2)+(1x2^1)+(0x2^0) = 22$

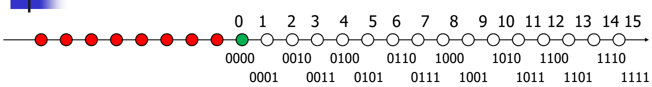
2^4	2^3	2^2	2^1	2^0
1	0	1	1	0

- Given an n-bit number
  - Range: 0 to $\mathbf{2^n - 1}$ (2^n different numbers)
  - Using 3 bits: 0 to 7

000	001	010	011	100	101	110	111
0	1	2	3	4	5	6	7

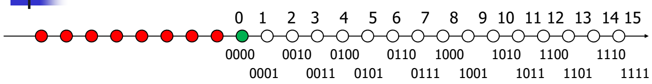
  - $x = x_{n-1} 2^{n-1} + x_{n-2} 2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$

Signed binary number



- We need both positive numbers and negative numbers
- How do we distinguish between them?
  - Turn some UNSIGNED numbers into negative numbers
  - Options? [e.g. +8 as 0? +8 as -1? +15 as 0? +15 as -1? …]
- The obvious solution would be:
  - Reserve one bit for sign, then sign and magnitude representation
  - Symmetry around zero
    - same number of positive and negative numbers represented [0000/1000; 0001/1001; 0010/1010; …]
    - but we have two zeros [1000_2 as -0 in the example above]

Bias representation - 1's complement

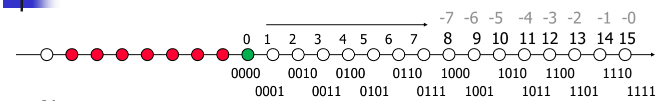


- Give up on symmetry
  - ~~[0000/1000; 0001/1001; 0010/1010; …]~~
- Translation of negative range by adding a distance (**bias**)

$$\text{representation}(x) = \begin{cases} \text{Binary}(x) & \text{if } 0 \le x < 2^{n-1} \\ \text{Binary}(\textbf{bias} - |x|) & \text{if } -2^{n-1} < x < 0 \end{cases}$$

- 1's complement
  - if we select **bias** = $\mathbf{2^n-1}$, we get 1's complement representation

Bias representation - 1's complement



- Note
  - no value is mapped to $\pm 2^{n-1}$; there are two 0s
  - pattern of all 1's is commonly referred to as negative zero
  - but we have symmetry
- Decimal Value of a negative number (e.g. 1010)
  - MSB determines the sign
  - Invert all bits, get the value for the positive number
    1010 -> inverted 0101 -> 5
- Problems
  - Arithmetic operations: try (-3) + (-4)

```
  1100
  1011 +
 10111
```


Bias representation - 2's complement

- Translation of negative numbers by a distance (**bias**)

$$\text{representation}(x) = \begin{cases} \text{Binary}(x) & \text{if } 0 \le x < 2^{n-1} \\ \text{Binary}(\textbf{bias} - |x|) & \text{if } -2^{n-1} \le x < 0 \end{cases}$$

- 2's complement
  - if we select bias = 2^n, we get 2's complement representation
- Note
  - we can represent a range from -2^{n-1} to $2^{n-1} - 1$
  - results in the simplest (fastest) hardware
  - universally accepted in all modern computers (also MIPS)

Bias representation - 2's complement

- Decimal Value of a negative number (e.g. 1010)
  - MSB determines the sign
  - Invert all bits, and add one , get the value for the positive number
    1010 -> inverted 0101 (5) -> 5 + 1 = 6
- Advantage
  - Arithmetic operations work naturally: try (-3) + (-4)

```
  1101
  1100 +
 11001
```

  - Sign extension
    - When moving n bits into an n+m bits container, it's safe to extend the sign bit to the leftmost

0 1 0 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1
1 1 0 1	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1

Bias representation - 2's complement

- 2's complement negation
  - Given x -> obtain -x
  - invert the number (turn every 0 to 1, and 1 to 0) ~x
  - Then add 1, that is -x = ~x + 1
- Two's complement operations: Addition & Subtraction
  - addition the same as for unsigned numbers

```
  0101          5
 +1010       + (-6)
 ------      ------
  1111         -1
```

  - subtraction using addition of negative numbers

```
  0101        0101        5
- 1010      + 0110      + 6
------      ------      ---
              1111       11
```


Overflow [Read more from the textbook]

- Overflow (result too large for finite computer word):
  - e.g., adding two n-bit numbers does not yield an n-bit number
  - the computer word is finite
- Arithmetic operations can create a number which cannot be represented

```
  0 111       7       1 111     (-1)
+ 0 001     + 1     + 1 110   + (-2)
  1 000       8       1 101      -3
```

- Two choices:
  - ignore overflow: for example in address arithmetic
  - detect and handle overflow in hardware
    - set a flag (overflow register)
    - or exception in the execution of the program

Handling overflow

- Detecting Overflow
  - No overflow is possible when
    - Addition: a positive and a negative number
    - Subtraction: signs are the same
  - Overflow occurs when the value affects the sign
    - adding two positives yields a negative
    - adding two negatives gives a positive
    - subtract a negative from a positive and get a negative
    - subtract a positive from a negative and get a positive
- Handling overflow
  - Overflow register
    - not in modern RISC architectures (MIPS there is no such a register)
  - An exception is triggered by hardware
    - in MIPS a special purpose register EPC (Exception Program Counter) can be used (details later)

Ignoring overflow

- We don't always want to detect overflow
  - When running unsigned operations
  - MIPS instructions: **addu**, **addiu**, **subu, sltu, ...**
- Note:
  - With **addu**, the "u" means "don't trap overflow"
  - **addiu** and **sltiu** still sign-extend
  - **sltu** for unsigned comparisons

Summary of Representations

3-bit Binary pattern	Decimal Values		
	Sign Magnitude	1's Complement •if MSB=0, positive value •if MSB=1, invert bits, assume negative	2's Complement •if MSB=0, positive value •if MSB=1, invert bits, add 1, assume negative
000	+0	+0	+0
001	+1	+1	+1
010	+2	+2	+2
011	+3	+3	+3
100	-0	-3	-4
101	-1	-2	-3
110	-2	-1	-2
111	-3	-0	-1

Unsigned and signed instructions

- A number can be interpreted by hardware as signed or unsigned
  - A byte may be an ASCII character, or of some other meaning
  - it depends only on the instruction operating on the number
- MIPS provides instructions for signed and unsigned numbers

	Signed	Unsigned
arithmetic	add, addi, sub, mult, div	addu, addiu, subu, multu, divu
comparison	slt, slti	sltu, sltiu
load	lb, lh	lbu, lhu

- Answer these questions:
  - why don't we have two versions of the lw instruction?
  - why don't we have two versions of the store byte sb instruction?

Unsigned and signed instructions

- example:
  - \$s0: 1111 1111 1111 1111 1111 1111 1111 1111
  - \$s1: 0000 0000 0000 0000 0000 0000 0000 0001
- Answer these questions:
  - what is the value of \$t0 and \$t1?

```
slt $t0,$s0,$s1    #
sltu $t1,$s0,$s1   #
```


MULTIPLY in MIPS: Instructions

- MIPS registers
  - two special purpose registers **hi** and **lo**
  - **hi**: high-order word of product
  - **lo**: low-order word of product
- MIPS instructions

```
mult rs1, rs2  # (hi, lo) = rs1 * rs2 ;signed
multu rs1, rs2 # (hi, lo) = rs1 * rs2 ;unsigned
mfhi rd        # move from hi to rd
mflo rd        # move from lo to rd
```

- Pseudo instructions

```
mul  $t0,$s1,$s2
mulo  $t0,$s1,$s2
```

DIVIDE in MIPS: Instructions

- all divide instructions put Remainder into hi register, and Quotient into lo register

```
div rs1, rs2    # divide rs1 by rs2; signed
                # quotient in lo, remainder in hi
divu rs1, rs2   # divide rs1 by rs2; unsigned
```

- Overflow and division by 0 are NOT detected by hardware
  - software takes responsibility
  - assembly language programmer or compiler
- Pseudo instructions

```
div $t0,$s1,$s2
```


Logical operations

- we may want to interpret a word
  - as fields of bits of various lengths
  - including a series of single bits

$s1	op	rs	rt	rd	shamt	funct

- instructions for operating on bit fields
  - **shifts** logical operations
  - **bitwise** logical operations

Shifts (Logical shifts, Arithmetic shift)

- Logical shifts
  - move all the bits in the register to the left or to the right filling the empty space with zeros
  - bits "shifted-out" are lost
  - shamt (shift amount): constant
  - Put the result in register rd:

```
sll  rd,rt,shamt    # shamt is a constant

sllv  rd,rt,rs      # Shift left logical variable

srl  rd,rt,shamt    #
srlv  rd,rt,rs      #
```


Shifts (Logical shifts, Arithmetic shift)

- Arithmetic shift
  - shift to the right with sign extension

```
sra  rd,rt,shamt    # shamt is a constant
                    #
srav  rd,rt,rs      # sra by a variable number of bits
```

  - Answer this question:
    - why no arithmetic shift to the left?
- Rotation
  - ror, rol
  - pseudoinstructions to rotate the register to left or right by a number of bits
  - no bits lost, bits "falling off" one end fed into the other end

Arithmetic by shifting

- For a base *n* representation
  - a shift to the left is like multiplying by **n**

    ```
    sll rd, rs, 2
    ```
  - a shift to the right is like dividing by **n**

				1	0	0	1

- PITFALLS
  - **multiplying** numbers by shifting left may result in overflow
    - but can be used with caution for small integers, for example
  - **division** by arithmetic (not logical) right shift
    - positives rounded down

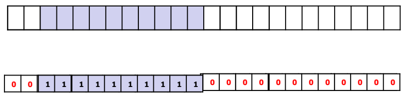
9

1	0	0	1

    - negatives? also rounded down?

Logical bitwise operations

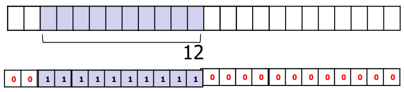
- performed bit by bit, so called bitwise operations
- general format like addition
  - log-op rd, rs1, rs2 # R-type instruction
  - log-opi rd, rs, constant # I-type instruction
- instructions available in MIPS (examples will follow)
  - logical AND
  - logical OR
  - logical NOR
  - logical XOR



exercise: think about this:
...why don't we have unsigned logical instructions?

Masking

- "cutting" out bit fields from a word
- a mask is a word (a constant or register contents)
  - with "1" for bits we want to keep
  - with "0" for bits we want to discard
- a logical AND on the mask and a word
  - leaves only the bits we selected in the mask
  - all other bits are cleared (replaced with zeros)



Extracting fields in an instruction

ASSUME: register $s1 contains a R-type instruction
TASK: extract the register numbers rs, rt, rd used in the instruction and save them in registers $s2, $s3, $s4 respectively

$s1	op	rs	rt	rd	shamt	funct
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Two approaches
  - shift first, mask second
  - mask first, shift second
- We will mask first
  - mask for rs: 0x03e0 0000=0000 0011 1110 0000 0000 0000 0000 0000
  - mask for rt: 0x001f 0000=0000 0000 0001 1111 0000 0000 0000 0000
  - mask for rd: 0x0000 f800=0000 0000 0000 0000 1111 1000 0000 0000
- all masks happen to be 5-bit long, so we can shift masks

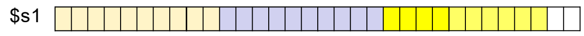
Extracting fields in an instruction

```
addi $t0,$zero, 0xf800  # mask for rd
and  $s4,$s1,$t0        # extract the field
srl  $s4,$s4, 11        # right alignment
sll  $t0,$t0, 5         # mask for rt
and  $s3,$s1,$t0        #
srl  $s3,$s3,16         #
sll  $t0,$t0,5          # mask for rs
and  $s2,$s1,$t0        #
srl  $s2,$s2,21         #
```

$s1	op	rs	rt	rd	shamt	funct
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

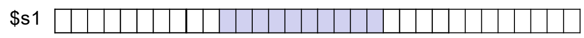
Extracting 2's complement numbers

ASSUME: $s1 contains THREE 10-bit long 2's complement numbers, packed in bits 2 to 31
TASK: let's extract the middle number



- We know:
  - the number is 10-bit long
  - the number starts at bit position 12
- Strategy
  - 10-bit mask (for bits 0-9) is 0x0000 03ff
  - Left-shift 0x0000 03ff by 12 to generate the mask needed
  - Sign extension is needed

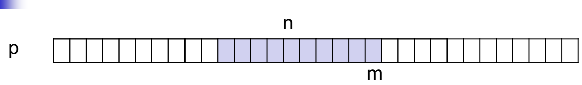
Extracting 2's complement numbers




```
addi $t0,$zero, 0x03ff  # 10-bit mask (for bits 0-9)

sll $t0,$t0, 12         # Left-shift by 12 to generate
and $s2,$s1,$t0         # the mask needed
                        #
sll $s2,$s2, 10         # left most to touch MSB
sra $s2,$s2, 22         # sign extension
```


Enlarged bit patterns from the previous page

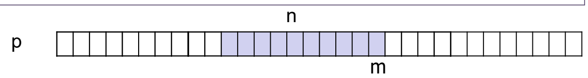


0000 0000 0000 0000 0000 0000 1010 1101	p=173
0000 0000 0000 0000 0000 0000 **1111** **1**000	mask
0000 0000 0000 0000 0000 0000 **1010** **1**000	AND
0000 0000 0000 0000 0000 0000 000**1** **0101**	srl
1010 **1**000 0000 0000 0000 0000 0000 0000	sll
1111 1111 1111 1111 1111 1111 111**1** **0101**	sra

LAB 7 help: write procedure "extract"

Refer to maskinghelp.pdf in 'Extra Materials' ribbon on vUWS

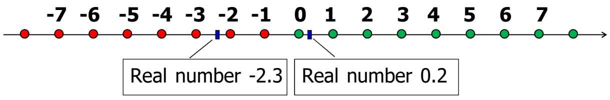
ASSUME: Numbers entered from keyboard are p, m, n
TASK: Write procedure named "extract" which extracts an **n**-bit field starting at bit **m** from a 32-bit value **p**



enter p, for example: 173 (use Windows calculator in Scientific mode for Hex-dec-binary conversions)		
	0000 0000 0000 0000 0000 0000 1010 1101	p is read into a register, say $a2
	0000 0000 0000 0000 0000 0000 **1111** **1**000	we can define this mask in our code
task: extract 5 bits starting from bit 3		
using the mask to extract the bits	0000 0000 0000 0000 0000 0000 **1010** **1**000	$t2 gets logical AND of p and mask
task: display the extracted bits as unsign and signed integers:		
to display as an unsigned integer:	0000 0000 0000 0000 0000 0000 000**1** **0101**	shift right logical, result say into $t3
to display as a signed integer do two shifts:		
first shift	**1010** **1**000 0000 0000 0000 0000 0000 0000	shift left logical up to bit 31
second shift -- final result	1111 1111 1111 1111 1111 1111 111**1** **0101**	shift right arithmetic back to bit 0

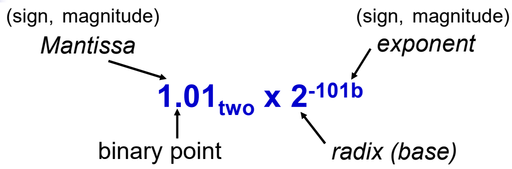
Other Numbers

The rest is for self-study
Also refer to
Lecture05 [Supplement]_fpNumbers.pdf



- What about
  - Very large numbers? (seconds/century)
    $3{,}155{,}760{,}000_{ten}$ ($3.15576_{ten} \times 10^{9}$)
  - Very small numbers? (second / nanosecond)
    0.000000001_{ten} ($1.0_{ten} \times 10^{-9}$)
  - Rationals
    2/3 (0.666666666. . .)
  - Irrationals
    $2^{1/2}$ (1.414213562373. . .)
  - Transcendentals
    e (2.718...), π (3.141...)

Scientific Notation for Binary Numbers



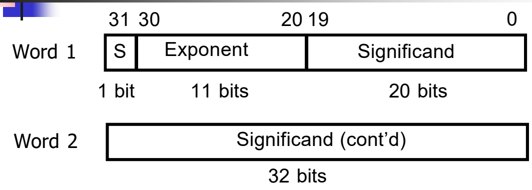
- Normal format: $\mathbf{1.xxxxxxxxxx_{two} * 2^{yyyy}_{two}}$
  leading digit is always **1**, so called 'hidden' or 'implied' **1**, and is implemented in hardware.
  - **1.xxxxxxxxxx**: Mantissa
  - **xxxxxxxxxx**: significand (significant positions)
  - **yyyy**: exponent

IEEE 754 Floating Point Standard

31	30 ... 23	22 ... 0
S	Exponent	Significand
1 bit	8 bits	23 bits

- Word Size (32 bits, 23-bit Significand Single Precision)
- Value: $\mathbf{(-1)^S \times Mantissa \times 2^{Exponent}}$ [broken into 3 parts]
- Range: Represent numbers as small as $\mathbf{2.0 \times 10^{-38}}$ to as large as $\mathbf{2.0 \times 10^{38}}$
  - if result too large? (> 2.0×10^{38}), **Overflow** => Exponent larger than can be represented in 8-bit Exponent field
  - if result too small? (>0, < 2.0×10^{-38}), **Underflow** => Negative exponent larger than can be represented in 8-bit Exponent field
- Issues: increase range (Exponent field) and accuracy (no. of significant positions)

IEEE 754 Floating Point Standard



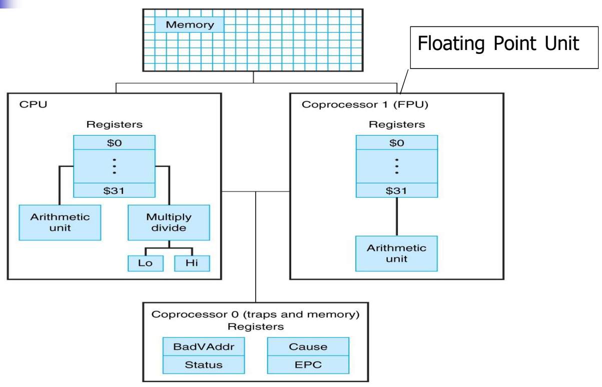
- Multiple of Word Size (64 bits, 52-bit Significand for Double Precision)
- Representing Mantissa: If significand bits left-to-right are $s_1, s_2, s_3, \ldots$ then, Mantissa: $1.s_1s_2s_3\ldots$; the FP value is:

$$(-1)^S \times (\mathbf{1}+(s_1 \times 2^{-1})+(s_2 \times 2^{-2})+(s_3 \times 2^{-3})+\ldots) \times 2^{Exponent}$$

NOTE: $1.s_1s_2s_3\ldots$

2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	...
1	s_1	s_2	s_3	s_4	s_5	...

MIPS FPU



MIPS Floating Point Architecture

- Single Precision, Double Precision versions of add, subtract, multiply, divide, compare
  - Single `add.s, sub.s, mul.s, div.s, c.lt.s`
  - Double `add.d, sub.d, mul.d, div.d, c.lt.d`
- Registers
  - Simplest solution: use existing registers
  - Normally integer and FP operations on different data, for performance could have separate registers
- MIPS provides 32 32-bit FP. reg: $f0, $f1, $f2 ...,
  - Thus need FP data transfers: lwc1, swc1
  - Double Precision? Even-odd pair of registers ($f0#$f1) act as 64-bit register: $f0, $f2, $f4, ...

New MIPS FP arithmetic instructions

```
add.s $f0,$f1,$f2 # $f0=$f1+$f2 FP Add (single)
add.d $f0,$f2,$f4 # $f0=$f2+$f4 FP Add (double)
sub.s $f0,$f1,$f2 # $f0=$f1-$f2 FP Subtract (single)
sub.d $f0,$f2,$f4 # $f0=$f2-$f4 FP Subtract (double)
mul.s $f0,$f1,$f2 # $f0=$f1x$f2 FP Multiply (single)
mul.d $f0,$f2,$f4 # $f0=$f2x$f4 FP Multiply (double)
div.s $f0,$f1,$f2 # $f0=$f1÷$f2 FP Divide (single)
div.d $f0,$f2,$f4 # $f0=$f2÷$f4 FP Divide (double)
c.X.s $f0,$f1     # flag1= $f0 X $f1 FP Compare (single)
c.X.d $f0,$f2     # flag1= $f0 X $f2 FP Compare (double)

# where X is: eq (equal), lt (less than), le (less than
# equal) to tests flag value:
# bc1t - floating-point branch true
# bc1f - floating-point branch false
```


Example with FP Multiply [Exercise - homework]

```
void mm (double x[][], double y[][], double z[][])
{
      int i, j, k;
      for (i=0; i!=32; i=i+1)
          for (j=0; j!=32; j=j+1)
              for (k=0; k!=32; k=k+1)
                  x[i][j] = x[i][j] + y[i][k] * z[k][j];
}
```

- Starting *addresses* are parameters in $a0, $a1, and $a2. Integer *variables* are in $t3, $t4, $t5. Arrays 32 by 32
- Use pseudoinstructions: li (load immediate), l.d/s.d (load/store 64 bits)

MIPS code for first piece: **initilialize, x[ ][ ]**

- Initailize Loop Variables

```
mm: ...
       li $t1, 32    # $t1 = 32
       li $t3, 0     # i = 0; 1st loop
L1:    li $t4, 0     # j = 0; reset 2nd
L2:    li $t5, 0     # k = 0; reset 3rd
```

- To fetch x[i][j], skip i rows (i*32), add j

```
       sll   $t2,$t3,5     # $t2 = i * 2^5
       addu  $t2,$t2,$t4   # $t2 = i*2^5 + j
```

- Get byte address (8 bytes), load x[i][j]

```
       sll $t2,$t2,3       # i,j byte addr.
       addu $t2,$a0,$t2    # @ x[i][j]
       l.d $f4,0($t2)      # $f4 = x[i][j]
```


MIPS code for second piece: **z[ ][ ], y[ ][ ]**

- Like before, but load z[k][j] into $f16

```
L3:    sll $t0,$t5,5       # $t0 = k * 25
       addu $t0,$t0,$t4    # $t0 = k*25 + j
       sll $t0,$t0,3       # k,j byte addr.
       addu $t0,$a2,$t0    # @ z[k][j]
       l.d $f16,0($t0)     # $f16 = z[k][j]
```

- Like before, but load y[i][k] into $f18

```
       sll $t0,$t3,5       # $t0 = i * 25
       addu $t0,$t0,$t5    # $t0 = i*25 + k
       sll $t0,$t0,3       # i,k byte addr.
       addu $t0,$a1,$t0    # @ y[i][k]
       l.d $f18,0($t0)     # $f18 = y[i][k]
```

- Summary: $f4:x[i][j], $f16:z[k][j], $f18:y[i][k]

MIPS code for last piece: add/mul, loops

- Add y*z to x

```
       mul.d $f16,$f18,$f16 # y[][]*z[][]
       add.d $f4, $f4, $f16 # x[][]+ y*z
```

- Increment k; if end of inner loop, store x

```
       addiu $t5,$t5,1     # k = k + 1
       bne $t5,$t1,L3      # if(k!=32) goto L3
       s.d $f4,0($t2)      # x[i][j] = $f4
```

- Increment j; middle loop if not end of j

```
       addiu $t4,$t4,1     # j = j + 1
       bne $t4,$t1,L2      # if(j!=32) goto L2
```

- Increment i; if end of outer loop, return

```
       addiu $t3,$t3,1     # i = i + 1
       bne $t3,$t1,L2      # if(i!=32) goto L1
       jr $ra
```

Revision quiz

- A binary pattern `1010` in 2's complement has equivalent decimal value:
  1) `-6` 2) `10` 3) `16`

- Is the following statement correct?
  A 32-bit word, without specifying a context, has no inherent meaning. That is, it can represent various things.

- `sll $s2, $s1, 1` has the same effect as
  1) `add $s2, $s1, $s1`
  2) `sub $s2, $s1, $s1`
  3) `muli $s2, $s1, 1`

Recommended readings

General Data	UnitOutline | LearningGuide | Teaching Schedule | Aligning Assessments |
Extra Materials	ascii_chart.pdf | bias_representation.pdf | HP_AppA.pdf | instruction decoding.pdf | masking help.pdf | PCSpim.pdf | PCSpim Portable Version | Library materials

PH6, §3.1, §3.2, §3.3, §3.5: MIPS Arithmetic; MIPS FP Architecture
PH5, §3.1, §3.2, §3.3, §3.5 [p211-p217 of §3.5]: MIPS Arithmetic; MIPS FP Architecture
PH4, §3.1, §3.2, §3.3, §3.5 [p259-p265 of §3.5]: MIPS Arithmetic; MIPS FP Architecture

HP_AppA.pdf -> A-51: Arithmetic and Logical Instructions

Text readings are listed in Teaching Schedule and Learning Guide

PH6 (PH5 & PH4 also suitable): check whether eBook available on library site

PH6: companion materials (e.g. online sections for further readings)

https://www.elsevier.com/books-and-journals/book-companion/9780128201091

PH5: companion materials (e.g. online sections for further readings)
http://booksite.elsevier.com/9780124077263/?ISBN=9780124077263