Chapter 2

Instructions: Language of the Computer

Chapter 2

- Introduction and computer operations
- Instruction Operands
- Signed and Unsigned numbers
- Representing Instructions (Instruction Format)

Instruction Set

- The collection of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

RISC vs CISC

CISC	RISC
Complex Instruction Set Computers	Reduced Instruction Set Computers
Variable length	Fixed length
Emphasis on hardware	Emphasis on software
Includes multi-clock complex instructions	Single-clock, reduced instruction only
Memory-to-memory: "LOAD" and "STORE" incorporated in instructions	Register to register: "LOAD" and "STORE" are independent instructions
Small code sizes, high cycles per second	Low cycles per second, large code sizes
Ex.: Intel x86, AMD	Ex.: MIPS,ARM,AVR,SPARC

The MIPS Instruction Set

- Used as the example throughout the book
- MIPS commercialized by MIPS Technologies (www.mips.com). Stands for Microprocessor without Interlocked Pipeline Stages
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern ISAs
  - See MIPS Reference Data and Appendices B and E

Categories of MIPS Instructions

- Arithmetic
- Logical
- Data transfer
- Conditional Branch
- Unconditional Jump

Example: MIPS Instructions

- `add a, b, c  # a gets b + c`
  - Three operands: Two sources and one destination
  - All arithmetic/Logic operations have this form

- ***Design Principle 1:* Simplicity favors regularity**
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Example: MIPS Instructions

- C code:
- f = (g + h) - (i + j);

- Compiled MIPS code:
- add t0, g, h # temp t0 = g + h
  add t1, i, j # temp t1 = i + j
  sub f, t0, t1 # f = t0 - t1

Instruction Operands

1) Register Operands

- Arithmetic instructions use register operands
- MIPS has a 32 × 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a “word”

- ***Design Principle 2: Smaller is faster***
  - c.f. main memory: millions of locations

MIPS Register Usage

Name	Register Number	Usage	Preserve on call?
$zero	0	constant 0	n.a.
$at	1	reserved for assembler	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return address	yes

The Constant Zero

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - E.g., move between registers

```
add $t2, $s1, $zero
```

Ex. Arithmetic Instructions

Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three register operands

- C code:

  f = (g + h) - (i + j);

  - f, g, h, i, j in $s0, $s1, $s2, $s3, $s4

- Compiled MIPS code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

2) Immediate Operands

Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three register operands

- C code:

  f = (g +3) - (i + j);

  - f, g, h, i, j in $s0, $s1, $s2, $s3, $s4

- Compiled MIPS code:

```
addi $t0, $s1, 3
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

2) Immediate Operands

- Constant data specified in an instruction

```
addi $s3, $s3, 4
```

- No subtract immediate instruction
  - Just use a negative constant

```
addi $s2, $s1, -1
```

- ***Design Principle 3: Make the common case fast***
- Immediate operand avoids a load instruction

Ex. Logical Instructions

- Instructions for bitwise manipulation

Operation	C	Java	MIPS
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR	|	|	or, ori
Bitwise NOT	~	~	nor

- Useful for extracting and inserting groups of bits in a word

Ex. Logical Instructions

Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant

Example:

$S0=0000 0000 0000 0000 0000 0000 0000 $1001_{two} = 9_{ten}$

After instruction:

sll $t2,$s0,4 # reg $t2 = reg $s0 << 4 bits

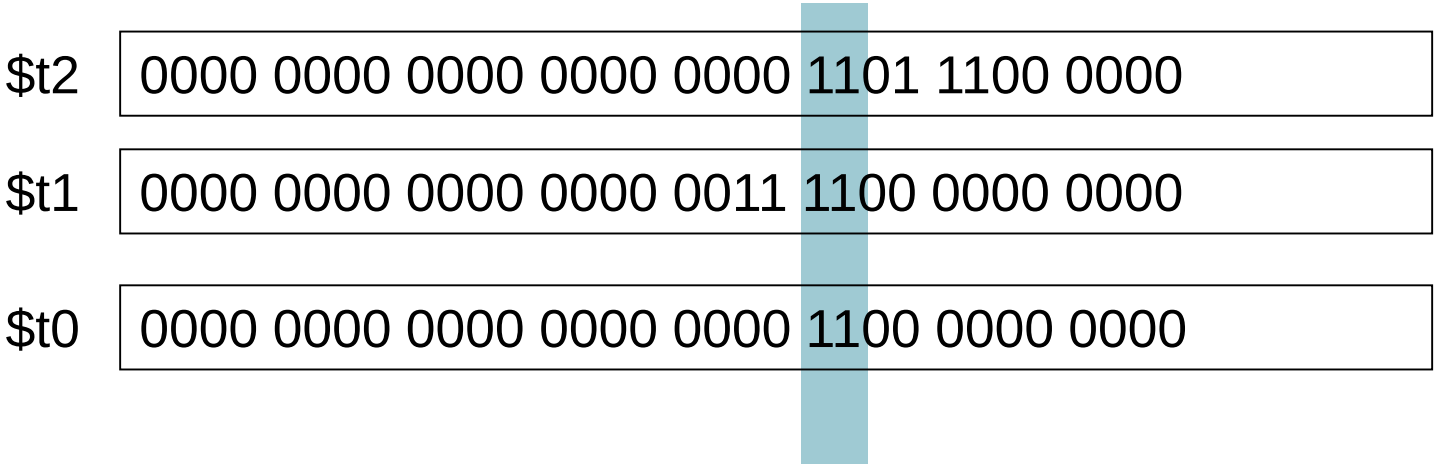
$t2= 0000 0000 0000 0000 0000 0000 1001 0000_{two}

$= 144_{ten}$

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

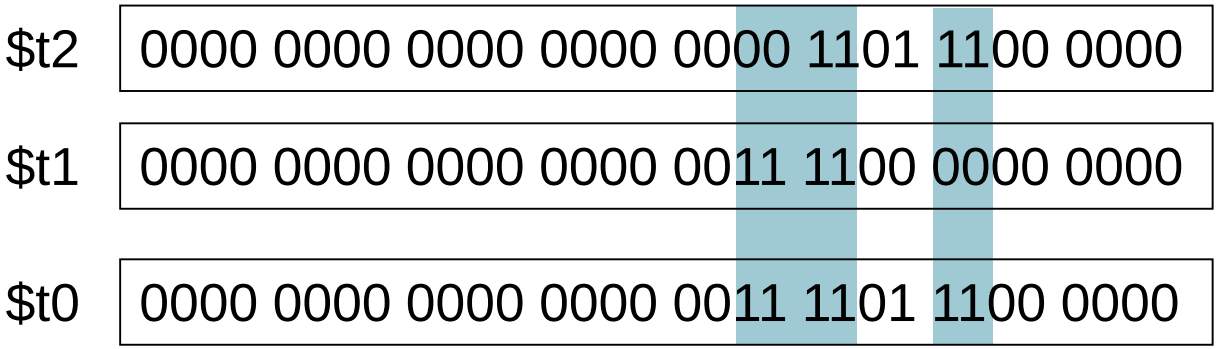
```
and $t0, $t1, $t2
```



OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```
or $t0, $t1, $t2
```



NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
  - a NOR b == NOT (a OR b)

```
nor $t0, $t1, $zero
```

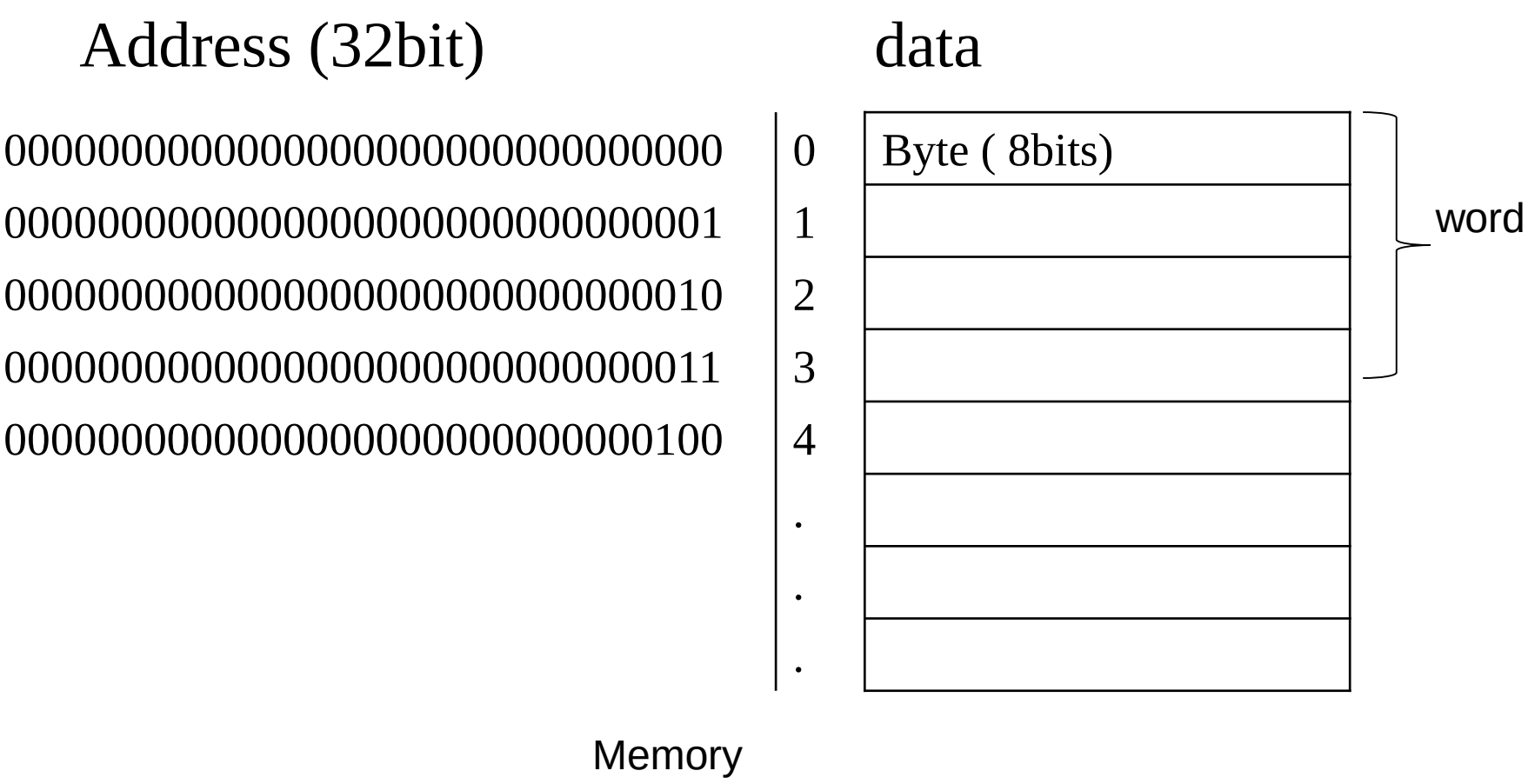
Register 0: always read as zero

$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111

3) Memory Operands

- Accessed only by data transfer instructions
- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4

3) Memory Operands



Ex. Data Transfer Instructions

Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits

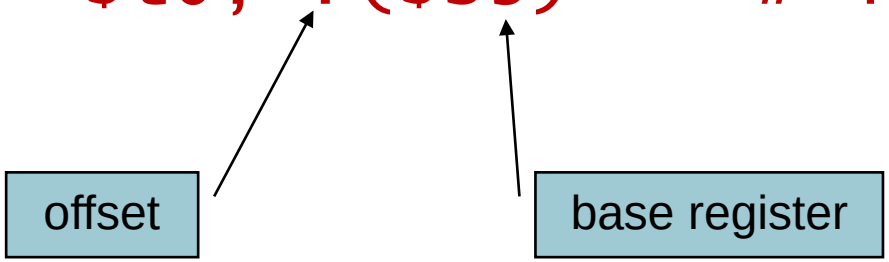
Memory Operand Example 1

- C code, A is array of integers:

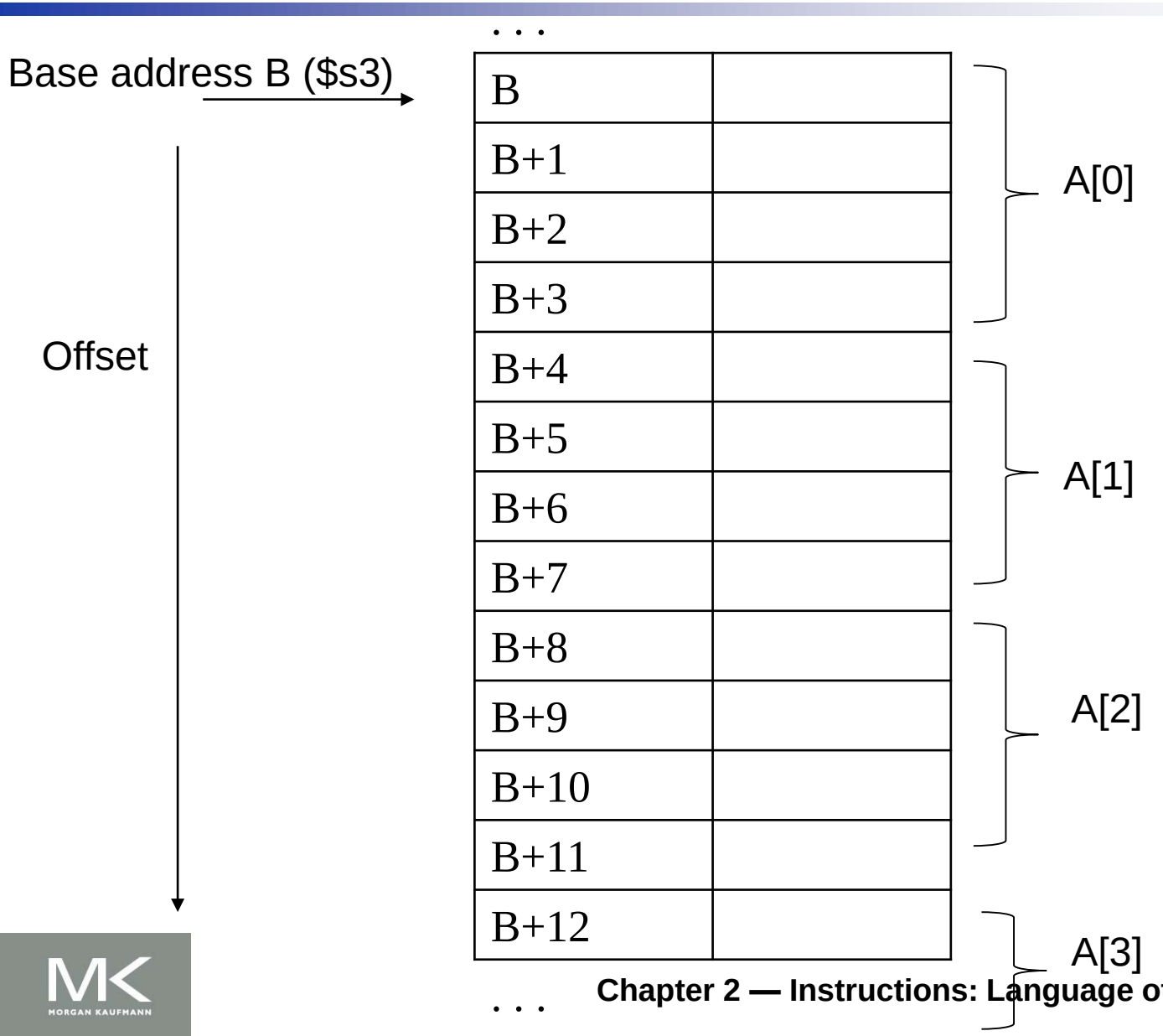
  `g = h + A[3];`

  - g in $s1, h in $s2, base address of A in $s3
- Compiled MIPS code:
  - Index 3 requires offset of 12
    - 4 bytes per word

```
lw  $t0, ?($s3)     # load word
```



Memory Operand Example 1



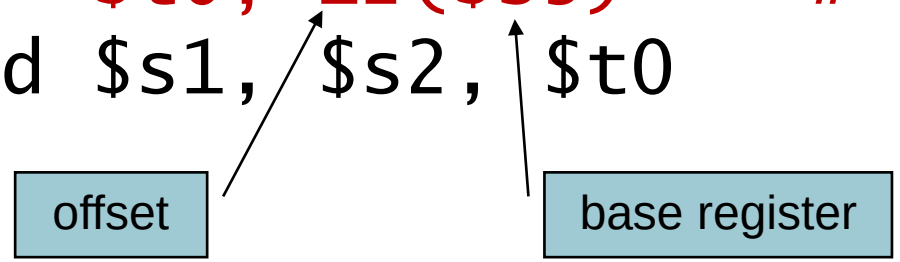
Memory Operand Example 1

- C code, A is array of integers:

  `g = h + A[3];`

  - g in $s1, h in $s2, base address of A in $s3
- Compiled MIPS code:
  - Index 3 requires offset of 12
    - 4 bytes per word

```
lw  $t0, 12($s3)    # load word
add $s1, $s2, $t0
```



Memory Operand Example 2

- C code:

```
A[12] = h + A[8];
```

  - h in $s2, base address of A in $s3

- Compiled MIPS code:
  - Index 8 requires offset of 32

```
lw  $t0, 32($s3)    # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)    # store word
```

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

Signed and Unsigned numbers

Unsigned Binary Integers

- Given an n-bit number

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

- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$

2s-Complement Signed Integers

- Given an n-bit number

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

- Example
  - $0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= 0\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= 2{,}147{,}483{,}644_{10}$
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0 = -4_{10}$
  - Switch the sign bit:
    - (0) → +ve → Multiply coefficient by weight
    - (1)→ -ve →2's Complement → Multiply coefficient

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
- In MIPS instruction set
  - `addi`: extend immediate value
  - `lb`, `lh`: extend loaded byte/halfword
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110

Problems to solve

- 2.1→ 2.5, 2.9,2.10