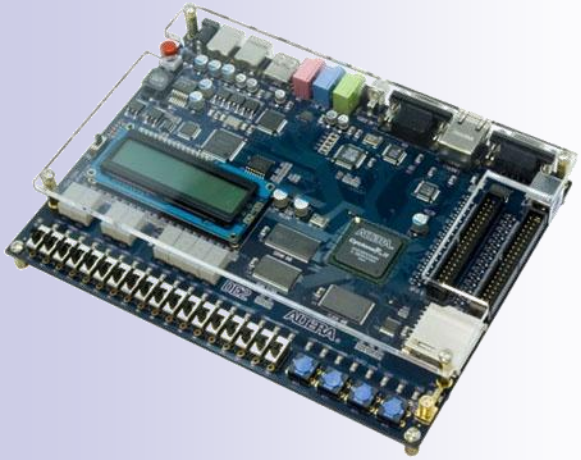
Selection from
Susta:Computer System Structures

Version: 1.1 - March 7th,2017

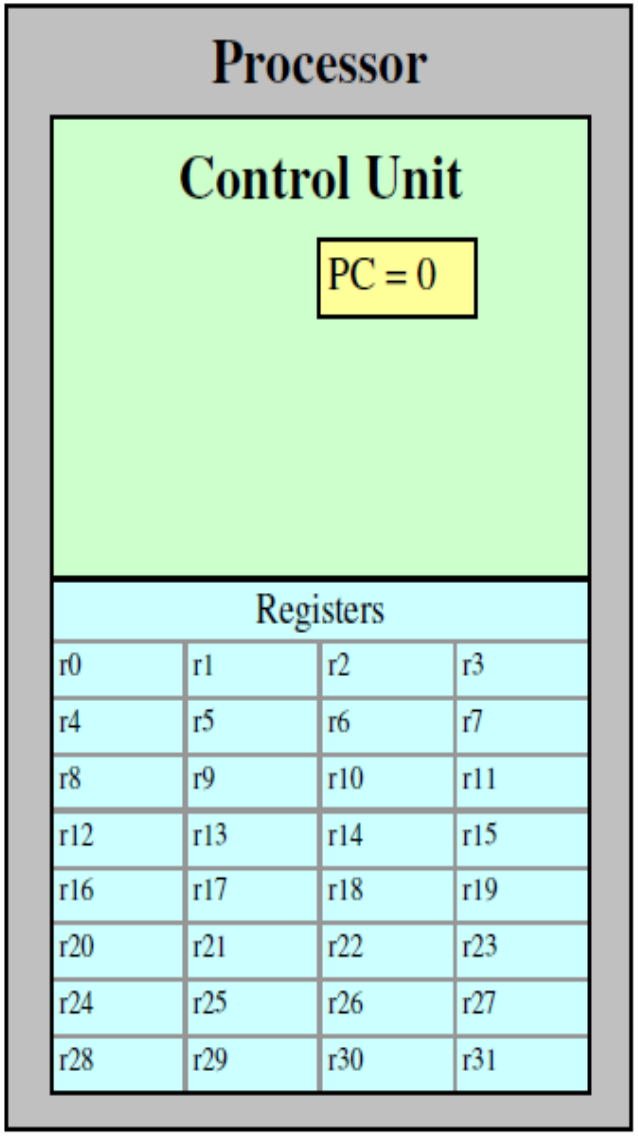


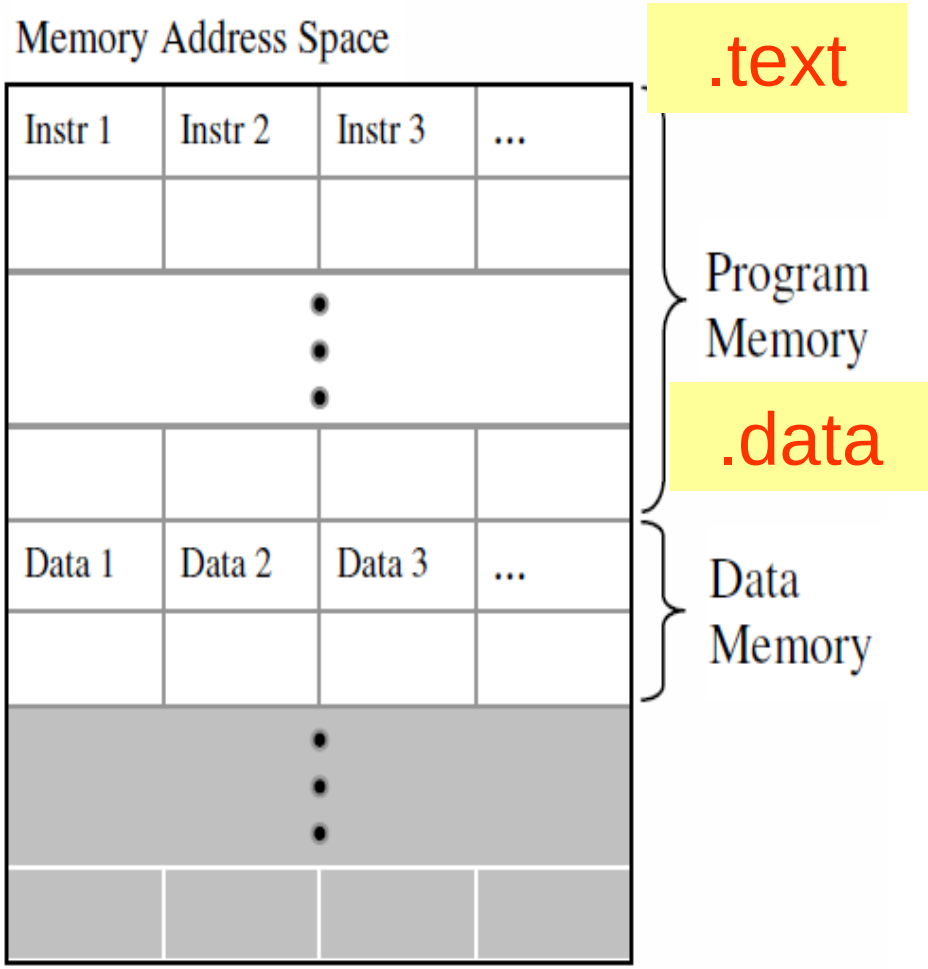
Practile exercise 3rd

ČVUT-FEL in Prague,

Microprocessors

NIOS/MIPS Simplified System





MIPS: Common Register Usage

Reg	Name	Normal usage
$0	**$zero**	0x0000_0000
$1	**$at**	Assembler Temporary
$2		Unsaved function arguments and retun value
$3		
$4		
$5		
$6		
$7		
$8		Unsaved temporaries
$9		
$10		
$11		
$12		
$14		
$14		
$15		

Reg	Name	Normal usage
$16		Saved Temporaries
$17		
$18		
$19		
$20		
$21		
$22		
$23		
$24		
$25		
$26		Global Poiner
$27		
$28	**gp**	Global Poiner
$29	**$sp**	Stack Pointer
$30	**$fp**	Frame Pointer
$31	**$ra**	return Address

Reserved

Some GNU assembler directives

.include	insert another file
.text	this directive tells the assembler to place the following statements at the end of the code section
.global	this directive makes the symbol visible to the program loading instructions into memory.
.data	this directive tells the assembler to place the following statements at the end of the data section.
.equ	this directive set the value of a symbol.
.word	this directive is used to set the content of memory locations to the specified value.
.end	Marks the end of current assembly program.

General Processor Instruction Formats

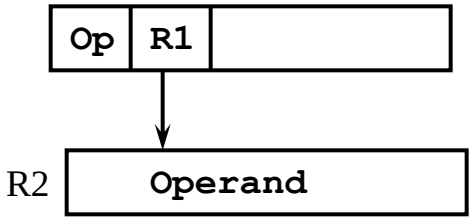
- Three-Address Instructions
  - ADD o1, o2, o3 — o1 ← o2 + o3
- Two-Address Instructions
  - MOV**E** o1, o2 — o1 ← o2
- One-Address Instructions
  - J o1 — PC ← o1
- Zero-Address Instructions
  - NOP — sll $0, $0, 0

- *R-Type - operands o_i are registers only*
- *I-Type - instruction contains an immediate value*
- *C-Type - control instruction*

NIOS/MIPS Addressing Modes

(a) Register direct addressing

Register contains the operand




```
move R2,R1
R2  ← R1
```

(b) Immediate addressing

Instruction contains the operand

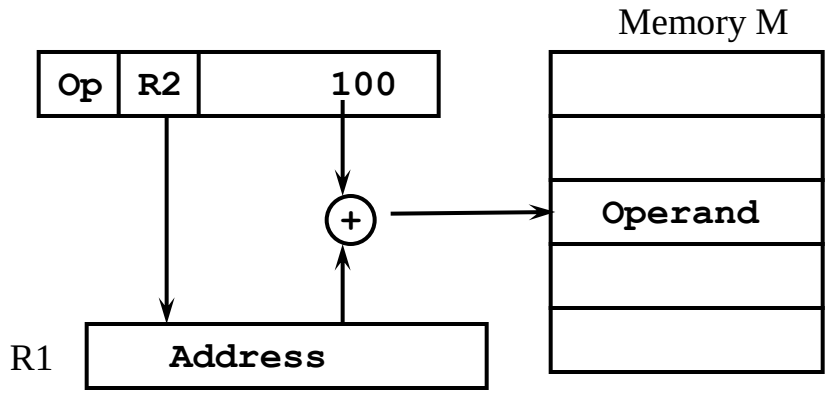

```
addi r2, r3, -20
```

```
sw R2, byte_offset(R1)
```

(c) Displacement (or offset) addressing

Address of operand = register + constant

stw R2, 100(R1) : **R2** → M[**R1**+100]



RISC vs CISC - Pentium Addressing Modes

Only for self-study

Mode	*Algorithm*
Immediate	Operand = A
Register	LA = R
Displacement	LA = (SR) + A
Base	LA = (SR) + (B)
Base with Displacement	LA = (SR) + (B) + A
Scaled Index with Displacement	LA = (SR) + (I) × S + A
Base with Index and Displacement	LA = (SR) + (B) + (I) + A
Base with Scaled Index and Displacement	LA = (SR) + (I) × S + (B) + A
Relative	LA = (PC) + A

LA=linear address ~ EA

[Source: Intel co.]

MIPS Load/Store

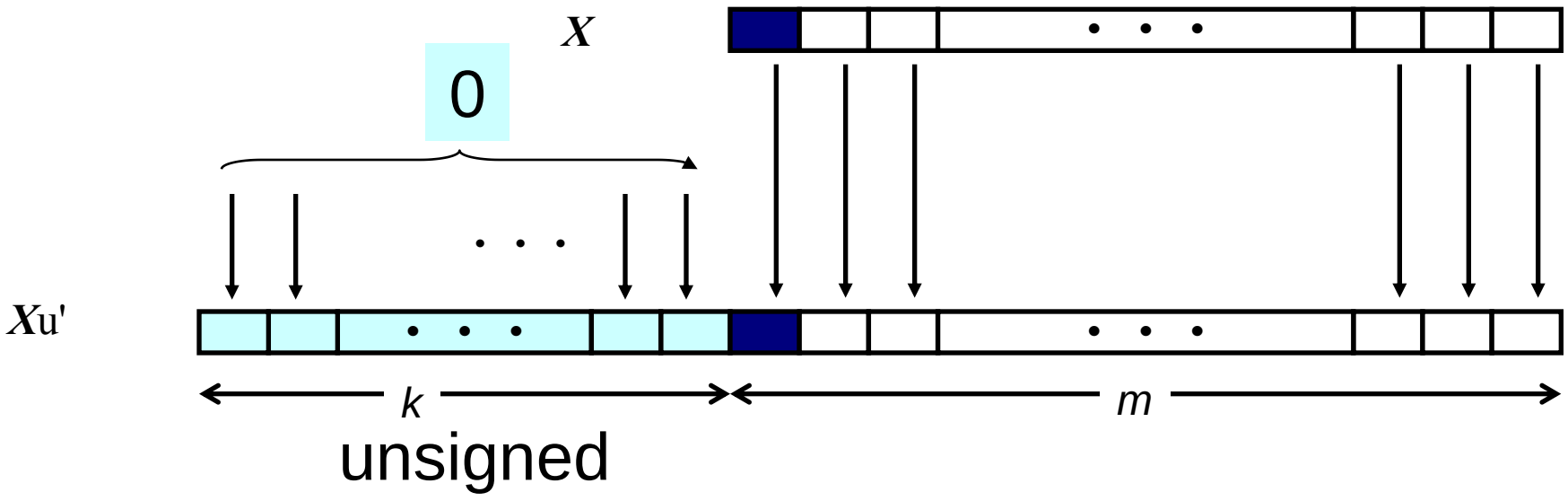
ldw (load 32-bit word from memory)

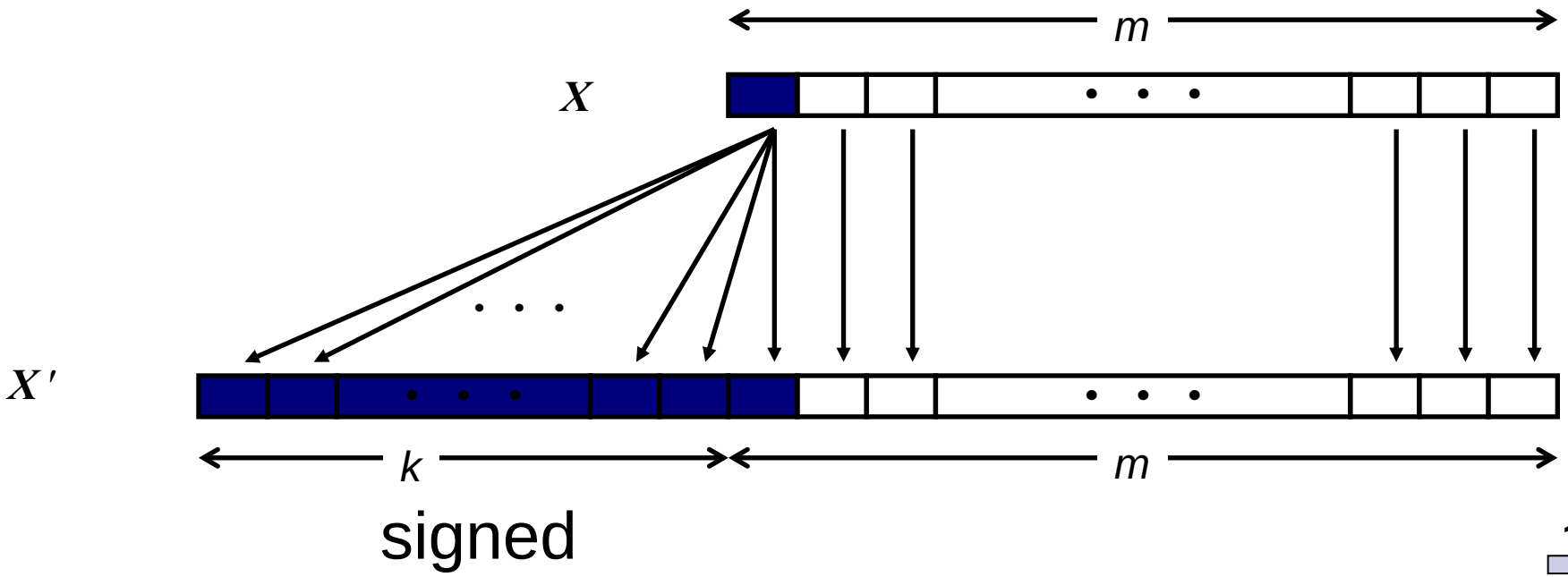
lw rB, im-const(rA) : **rB** ← MEM**[rA + im-const]**

stw (store 32-bit word into memory)

sw rB, im-const (rA) : **rB** → MEM**[rA + im-const]**

Unsigned/Signed Extension

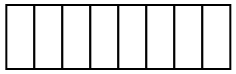




Data Types in Instructions

MIPS registers hold 32-bit (4-byte) words.

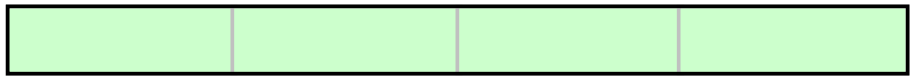
Other common data sizes include byte and halfword.



Byte = 8 bits - example: **lb / lbu**
- load byte extend signed/unsigned

Halfword = 2 bytes example: - **lh / lhu**
- load halfword extend signed/unsigned

Word = 4 bytes - example: - **lw** - load word



u-unsigned extension io - bypass cashe

Nios utilizises instead of l prefix ld + io

ALU Instructions

operation	R-format	I-format
add	`add / addu`	`addi / addiu`
subtract	`sub / subu`	`subi / subiu`
multiply divide	`mult / multu` `div / divu`	
AND	`and`	`andi`
OR	`or`	`ori`
XOR	`xor`	`xori`
NOR	`nor`	

`addi` `rB` ← `rA + se(`*`number`*`)`

Logic instructions AND, OR, XOR, NOR do not use se = sign-extension

Load Construction

Operation	Construction	Example
reg2 ← reg1	add reg2, reg1, $0	add $10, $11, $0
reg1 ← s_value	addi reg1, $0, s_value	addi $10, $0, -100
reg1 ← u_value	ori reg1, $0, u_value	ori $10, $0, 0x1243
reg1--	addi reg1, reg1, -1	addi $8, $8, -1
reg1++	addi reg1, reg1, 1	addi $9, $9, 1
reg1 ← big_value	lui reg1, $0, big_value_upper ori reg1, $0, big_value_lower	lui $7, $0, 0x1234 ori $7,$0, 0x4567
reg1 ← label	la reg1, label *(compiled as reg1 ← big_value)*	label1: la $1, label1

s_value - 16bit signed value
u_value - 16bot unsigned value

Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue
- `beq rs, rt, LA`
  - if (rs == rt) branch to instruction labeled LA;
- `bne rs, rt, LA`
  - if (rs != rt) branch to instruction labeled LA;
- `j LA`
  - unconditional jump to instruction labeled LA

More Conditional Operations

- Set result to 1 if a condition is true
  - Otherwise, set to 0
- `slt rd, rs, rt`
  - if (rs < rt) rd = 1; else rd = 0;
- `slti rt, rs, im_constant`
  - if (rs < im_constant) rt = 1; else rt = 0;
- Use in combination with `beq`, `bne`

```
slt $t0, $s1, $s2  # if ($s1 < $s2)
bne $t0, $zero, L  #   branch to L
```

Signed versus Unsigned

- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`
- Example
  - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `slt  $t0, $s0, $s1  # signed`
    - –1 < +1 ⇒ $t0 = 1
  - `sltu $t0, $s0, $s1  # unsigned`
    - +4,294,967,295 > +1 ⇒ $t0 = 0